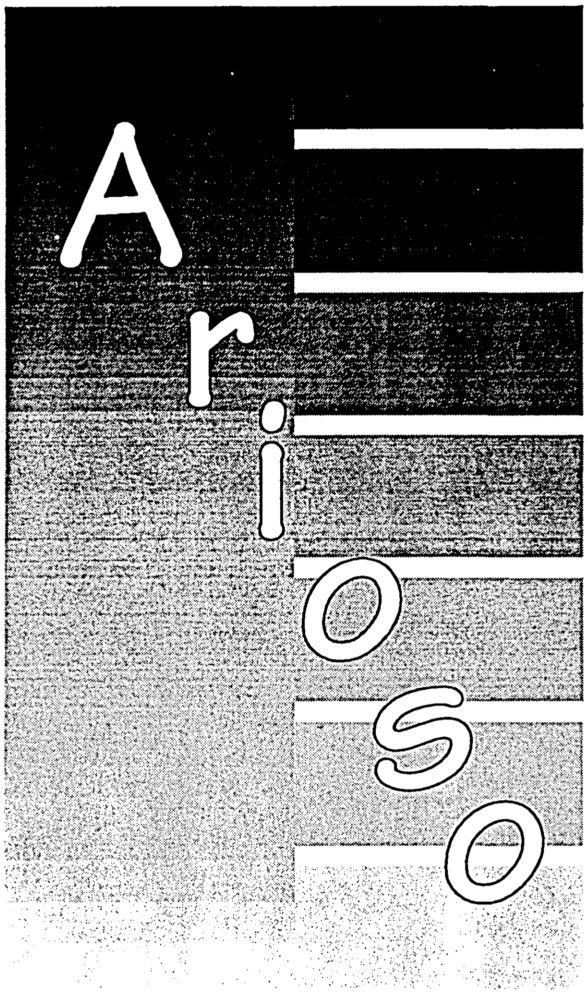


Macintosh I/O Systems Architecture version 1.0 1/29/92

Henry Kannapell
Scott Sarnikowski
Architectural Investigations & Modeling
Functional Technologies

Contents

Preface — About this Document

This document contains a description and discussion of the Arioso model of several Macintosh IO subsystems. The Arioso model is intended to be a conceptual model of IO subsystems that helps to improve the portability of low-level I/O software from one platform to another.

What's in this Document?

This document describes the important features and concepts of the Aioso I/O model for Macintosh computers. This document contains the following material:

- Chapter 1, "Introduction to Ariso" provides an overview of the Arioso model and discusses the motivation behind the Arioso Model. It also includes a description of a new set for diagramming symbols that capture the complexity of real-time algoritms.
- Chapter 2, "ADB - Apple Desktop Bus" documents the Arioso model for the ADB.
- Chapter 3, "Floppy" documents the Arioso model for the Macintosh Floppy I/O system.
- Chapter 4, "LocalTalk" documents the Arioso model for LocalTalk.
- Chapter 5, "Modem" documents the Arioso model for Modems.
- Chapter 6, "SCSI - Small Computer Systems Interface" documents the Arioso model for SCSI.
- Chapter 7, "Sound Out" documents that Arioso model for Sound Out.

This manual is structured so that the chapters that deal with a particular subsystem can standalone. So, if you are only interested in SCSI read only chapter 6. In order to get a backgound on the Arioso model, make sure that you read Chapter 1 prior to reading any of the later chapters.

Chapter 1. Introduction to Arioso

Henry Kannapell & Scott Sarnikowski

About This Chapter

This chapter presents an introduction to the Arioso model of the Macintosh IO Device System. The Arioso model is loosely based on the OSI model, specifying eight major layers for each IO device type.

While the CPU Design Centers have been able to keep producing new Macintoshes with new and innovative features, the cost of porting our three operating systems (Pink, Mac OS, and A/UX) grows with each new machine. The New Kernel of Mac OS will soon represent a fourth. Further, MAC portable products are redefining some of the traditional functions considered part of "every" Macintosh.

We set out to provide a framework for understanding and documenting the features and timing specifications required to support each of the Mac operating systems. This chapter describes the purpose of the project, defines the model we are using, explains each of the layers of the model in detail, and specifies a new form of flowcharting used in documenting the Arioso model of each IO device.

About Arioso

As the complexity and variety of computer systems has increased at Apple Computer, the need for simplifying the task of supporting multiple systems can no longer be ignored. Currently there is no coherent strategy for managing this problem.

Arioso is a technical piece of the solution to this problem. Arioso specifies a set of layer definitions to more easily support new platforms. Each layer, once supported, provides a foundation for all subsequent higher layers. The layers are constructed so that each is independent of the others with very well defined interfaces. The support of a new machine is simplified by selecting one of the layers for each I/O system and making it the standard interface between low-level and high-level software. This approach allows software and hardware below the chosen layer to be changed without requiring redesign of the higher-levels of software.

Although maximum flexibility is gained by implementing all of the Arioso layers, the definitions and specification of these layers should not be construed to mean that all the layers must be implemented. Instead, Arioso should be thought of as a generic conceptual model similar to the OSI model of data transmission. For example, developers may implement three model layers in a single layer to optimize performance. In most cases, there will be two or three layers that are documented for each I/O system.

Because Arioso is intended to be a generic conceptual model for IO devices and systems, the varied nature of the IO devices can lead to very different looking models for each IO device. In fact, not all of the layers may be specified in a particular system. For example, there may not be a network layer for some I/O systems, or FIFOs may not exist at a particular level.

In addition, the Arioso layers are defined so that they are not restricted to be either software or hardware implementations, nor are they restricted to a single processor. The layers are documented through a Command-Return-Result paradigm.. This paradigm defines three stages of execution for a command: Command, Return, and Result. During the command phase the command and any associated parameters are passed to the layer. During the return phase the thread of execution is returned to the caller in addition to any status or data. The time between the end of the command phase and the return can vary from immediate return to some time in the future, depending on implementation. Finally during the result phase the caller is notified of the completion of the command and any data or status can be returned at this point. There is a more detailed explanation of the Command-Return-Result paradigm in the Call Implementation section.

Why Arioso?

There are two system issues that complicate the process of supporting IO devices on new CPUs: ***hardware visibility*** and ***maximum tolerable latency***. Hardware visibility issues are concerned with the requirement of certain hardware functions to be addressable or visible by software. One example is the need to provide software visibility to detect when media is inserted into a floppy drive. Another example is the need to provide access to the DMA Request line of a SCSI chip. In each case, software must have access to properly implement the device drivers. At each layer of the Arioso model of the IO device, features like the ones mentioned above must be defined and specified.

Maximum tolerable latency issues are concerned with the amount of time a layer has between adjacent mandatory actions to support another layer in the model. These specifications provide the timing constraints imposed on a layer by the layer that it is supporting. One example is the time between the transmission of the last byte of a LocalTalk packet and the transmission of the abort sequence. The LocalTalk protocol requires that this time be no greater than 34.72µs. Therefore the maximum tolerable latency in the layer responsible for transmitting the last byte and the abort sequence is 34.72µs. Another example of a maximum tolerable latency specification is the duration and frequency that interrupts must be enabled for the SCSI manager. If either these specifications are violated, the SCSI manager will break.

In general, the Arioso model is defined so that as you move up the layers the dependance on the hardware is reduced. For maximum tolerable latency requirements, this means that higher layers have larger tolerable latencies, while lower layers have smaller tolerable latencies. For example, when working with hard disks, the tolerable latencies of the filesystem are considerably larger than the tolerable latencies of the SCSI manager. This can be demonstrated by the fact that at the application level you can stop a program with MacsBug and resume, where as stopping the SCSI manager may result in an abnormally terminated operation with no hopes of resuming.

Hardware visibility requirements show the same characteristics in an Arioso model. The higher the Arioso layer, the less dependent the layer is on hardware. Again using hard disks for example, the filesystem level shouldn't care whether the disk is an ESDI or SCSI disk, but at the device driver and lower levels these two different interfaces present different sets of hardware registers and status bits. Therefore lower levels need higher hardware visibility to interface to different register sets, while higher levels in the Arioso model should be independent of the hardware.

The specification of maximum tolerable latency and hardware visibility are summarized by the graphs shown in Figure 1. Each Arioso model will yield a set of graphs like the ones in Figure 1 that provide a simple and concise picture of an IO system.

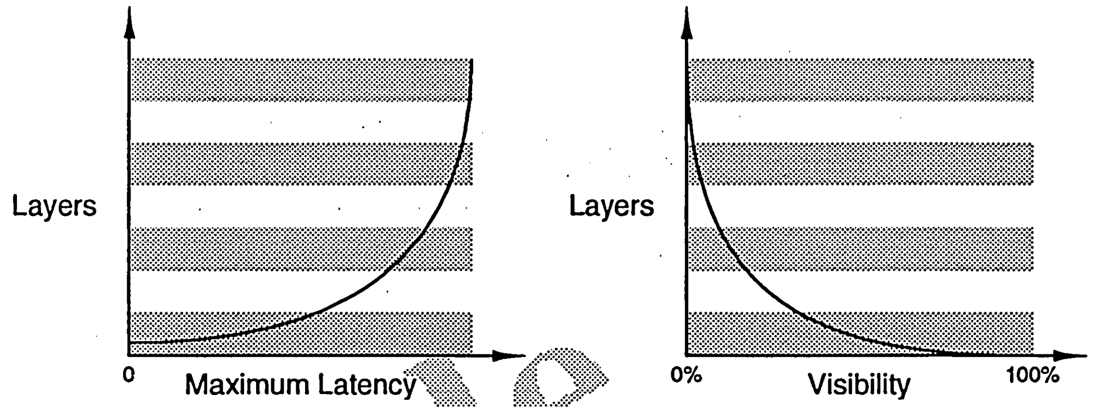


Figure 1. Examples of Maximum Latency and Visibility as a Function of Arioso Level

Arioso makes three major contributions to Apple development. First, Arioso provides a level of documentation and analysis that makes issues such as hardware visibility and maximum tolerable latency accessible to anyone in hardware or software involved in the development of new CPUs or new versions of system software. Second, Arioso provides the knowledge base necessary to intelligently partition software development between hi-level and low-level groups. Finally, Arioso provides a central repository of technical information about the Mac that does not exist in any of the traditional reference material.

Architectural Focus of Arioso

In order to understand the architectural focus of the Arioso project, we need to examine both Mac system hardware and Mac software software.

Mac System Hardware

Mac systems hardware can be seen as consisting of two major pieces (Figure 2.) The first major piece consists of structures that are traditionally thought of as tightly coupled to the core processor. Examples are: DMA, main memory, cache memory, memory busses, backpanel busses, MMUs, FPUs, and video frame buffers. All of these elements are best classified as memory reference and transformation elements and are referenced largely without any peripheral activity. In a sense, they only change the internal state of the computer, causing the internal memory of the computer to change from one state to another.

The second major piece of Mac system hardware is the I/O system. The IO system consists of several subsystems that are not as closely coupled to the core processor as the memory devices mentioned earlier.Examples of these subsystems are: SCSI peripheral interface chips, timers, sound circuits, keyboard circuits, video display circuits, communication circuits, and other such items.

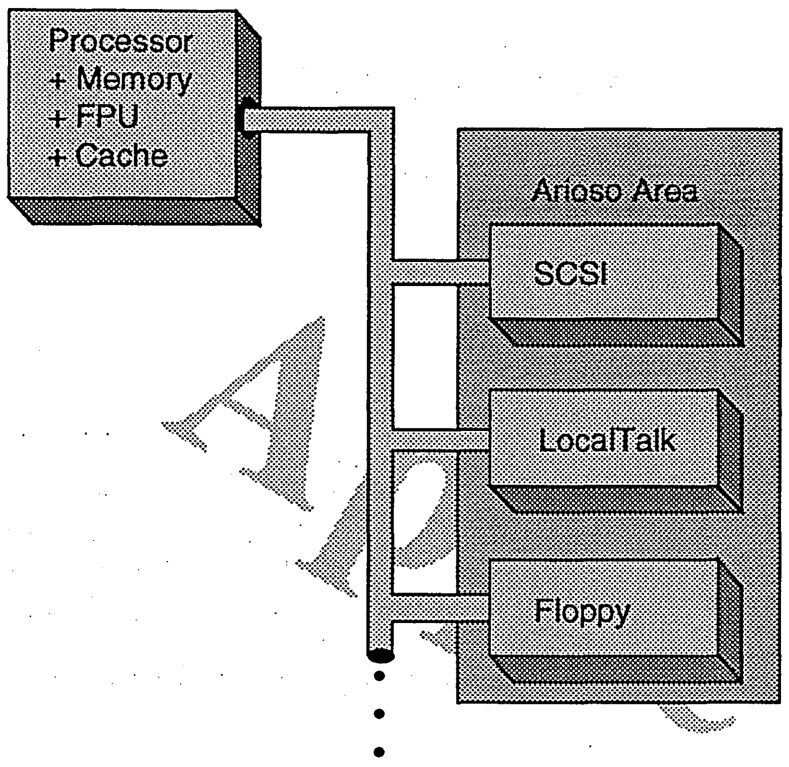


Figure 2. Processor and IO System

The subsystems within the IO system can be further divided into two pieces: ***IO Address Reference*** and IO Physical System. The I/O Address Reference is the portion of the I/O subsystem that is seen and addressed by processor bus cycles, and transfers data and commands to and from the memory reference system. The *I/O Physical System* consists of state machines, transducers, wires, cables, and electronic circuits that actually effect the environment.

Mac System Software

As with Mac system hardware, Mac system software can be seen as consisting of two major pieces. The first piece consists of memory management, application management, applications, file system, user interface, etc. In general, these pieces of software do not directly affect IO devices.

The second major piece of Mac system software consists of device drivers, or that portion of the code that manipulates the I/O devices. Mac system software utilizes Managers which represent sections of I/O systems.

Architectural Focus

Arioso's architectural focus is on the IO Address Reference portion of the Mac's IO system. Within this area, Arioso will address the layers within Mac system software that present the logical representation of the IO system to the rest of system software. In most cases the top of the Arioso model will correspond to the device driver of the IO system.

Arioso Layer Definition and Description

Figure 3 shows the layers within the Arioso model.

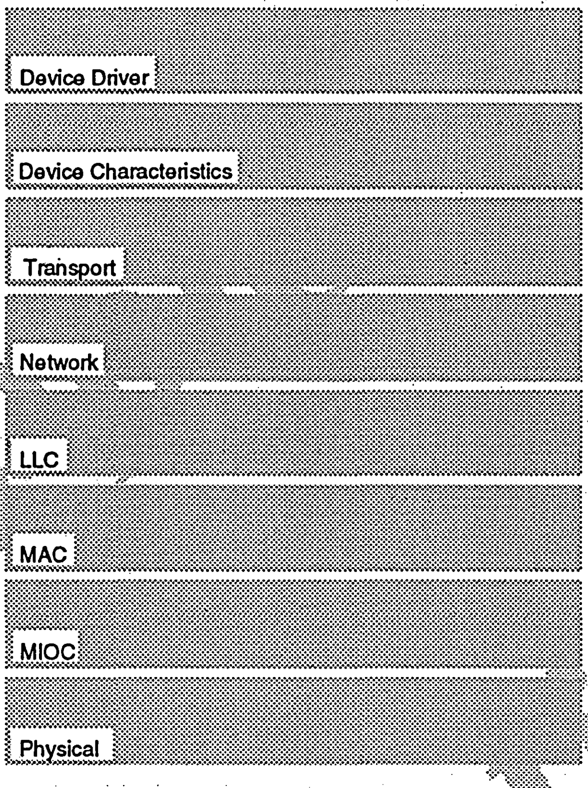


Figure 3. Arioso Model

Within the Arioso model, device drivers are divided into a number of layers. Each layer has an interface to the layer above it and an interface to the layer below it. The layers are described from the hardware devices at the bottom, up to the device driver interface (which is different between the different operating systems) at the top. As described earlier, the required response times and hardware independence of all of these systems gets larger as you move up the hierarchy chain. Layer 0 requires the fastest response time; the highest layer has the longest required response times and highest hardware independence.

All the layers have a similar structure (Figure 4). There is a control section which observes events and decides what actions to take next, and a framing section which implements those actions on the output side, and classifies the events on the input side. The control section is a state machine that manipulates tokens for its actions, and observes tokens for state changes. These tokens are then expanded (for actions) or classified (for inputs) by the framing section.

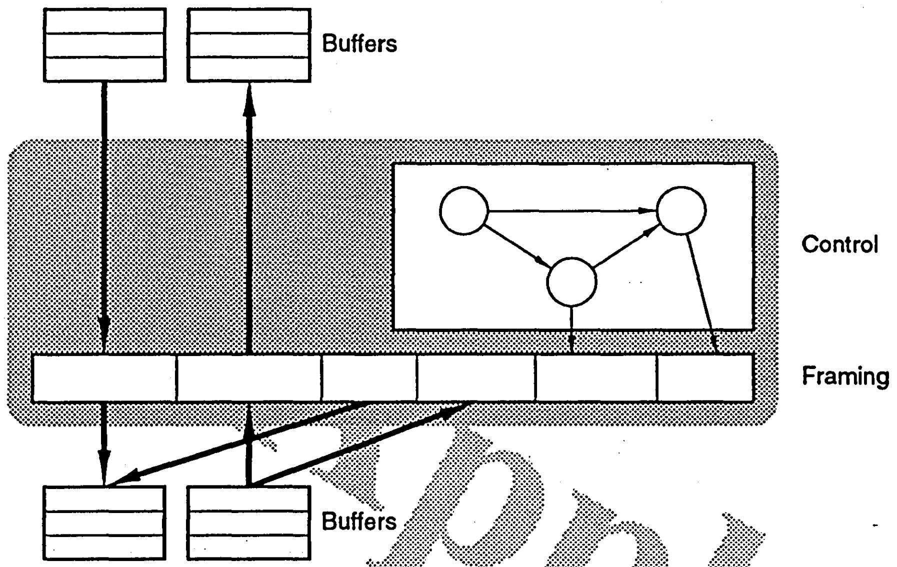


Figure 4. Diagram of an Arioso Layer

For example, for LocalTalk, the Mac layer sends an RTS packet (a four byte entity) when it prepares for transmission. The control section would generate a "Send RTS packet" token as an action. The framing section would then build an RTS packet of the four bytes and then queue them for transmission.

The protocol then expects a CTS packet (which is also four bytes). When four new bytes are received, they are classified as a CTS packet, and the framing section passes a "Received CTS packet" to the control section.

There are buffers for data in and data out, both for going up the layers and going down the layers. Any or all of these buffers can be FIFOs or queues, or only single element buffers. In some cases, the queues may not be ordered, such as in a disk access queue, in which the order of execution is determined by minimum seek time, not the time at which the entry was placed in the queue.

These data buffers may contain either references to data or the actual data itself. If each layer used actual data, there would be a copy operation between every layer. Normally there is only one or two copies for each data unit to be transmitted.

Each of the layers is now described in more detail.

Layer 0 — Physical

At the Physical layer, data is read or written directly to a buffer that marks the last boundary that bus cycles could see. For a parallel to serial converter, for example, the parallel buffer is the last boundary. Often the act of writing or reading to a buffer will trigger other activities to take place. In most modern peripherals, this layer is actually inside a hardware component, and a FIFO is the actual CPU interface.

Layer 1 — MIOC Layer

The Media Input/Output Control Layer addresses the media, and is responsible for transmitting and receiving data and status from the media.

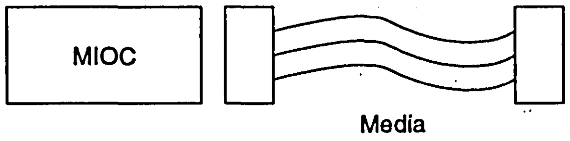


Figure 5. MIOC Layer

The MIOC layer is the first layer that looks at received data packets. This layer can send packets, receive, analyze, and classify packets, detect errors, and do other sorts of device maintenance. This layer is often thought of as the "lowest layer" from a software programming point of view.

The error messages are then passed up to the next layer. The MIOC layer provides independence of hardware for the purpose of determining errors. It does not matter how the errors are actually detected, only that the supported error messages be forwarded up to the next layer.

Basically, this layer is responsible for controlling the operations of the interface device and the media. The MIOC layer does not require a device to be a connected, since the interaction is only with the media and the interface device. Alternately, there is only one address supported in this layer; it is the address of the media.

All of the data bytes being transferred at this layer are sequences of bytes with no particular meaning to the MIOC layer. Most of the framing at this layer is hardware signalling instead of extra data bytes.

The problems of segmenting data, either because of virtual memory or because of scattered sources or destinations should be resolved by this layer. Data input is thus either a single contiguous buffer or a list of contiguous buffers to be sent as one unit.

The MIOC layer is normally considered part of the MAC layer, which is defined below. It is broken out separately because it is this layer that is primarily affected by new CPU designs.

Layer 2 — MAC Layer

The Media Access Control layer is half of the Data Link Layer, and is used in the same sense as the OSI protocols. The MAC Layer is the first layer at which a conversation takes place between the CPU and another peripheral device. Multiple logical connections may exist, but all devices that are part of these connections are directly connected to the media.

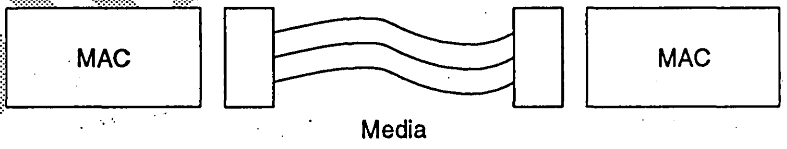


Figure 6. MAC Layer

This is the first level at which error recovery to an attached device can take place. The errors are detected by the MIOC layer or the framing section of the MAC layer and processed by the MAC control layer. The MAC layer can initiate packet transmissions.

Another way of stating this is to say that there is exactly one address per attached peripheral. A particular peripheral may be able to address 8 or 16 or 256 different devices, but each of the addresses are unique to a particular device.

Each of these connections looks like a distinct entity to the upper layers of the model. In the SCSI example, there appear to be a maximum of 8 different devices to the upper layers of the model. In a sense, the attached peripherals are represented to the upper layers by the operations of the MAC layer. The only characteristics that are represented are those required to maintain a connection and transfer data. The difference between a scanner and a printer, for example, are not visible at this layer.

Layer 3 — LLC layer

The Logical Link Control layer sits on top of the MAC layer. Several protocols allow one to talk to multiple logical entities within a physical device. The LLC protocol controls the connections and behaviors between these two logical entities. There may be multiple entities within any given I/O device. For example, SCSI allows one to have up to 8 Logical Units within any given device, while having only one media address. Most network protocols allow one to address different protocol stacks within a computer, such as a TCP/IP stack and an AppleTalk stack at the same ethernet media address. These capabilities are managed by the LLC layer.

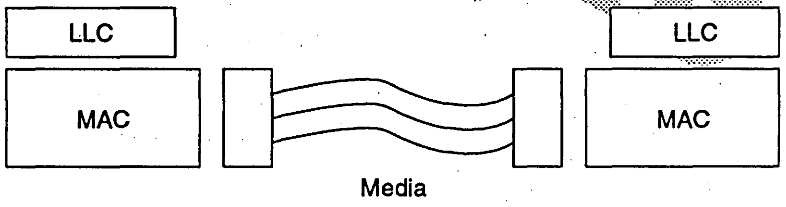


Figure 7. LLC Layer

To the upper layers of the model, each of these different logical units is addressed by this layer. It appears to the upper layers as though a connection is established to multiple different entities, all running asynchronously to each other.

Layer 4 — Network Logical Device

The Network Logical Device allows one to address logical entities that are beyond the scope of the current media. Typically a connection is maintained between peer devices on each of the media between the source and the destination. Bridges between two logical nodes for two different media connections on the same device accomplish the forwarding between media.

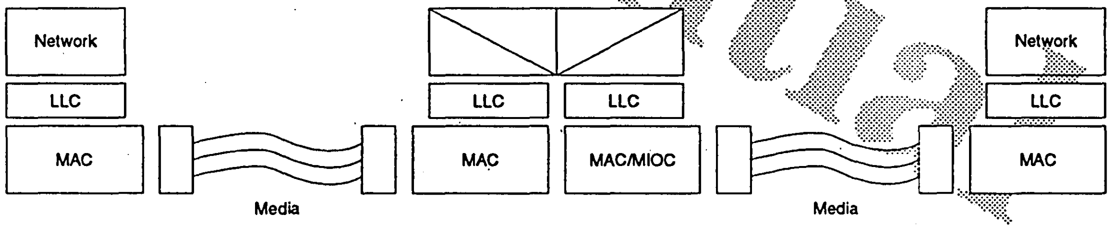


Figure 8. Network Layer

This layer is recursively defined. After bridging between media to form a network, a new layer may be introduced that bridges between networks to form an internet protocol. These internets may then be bridged again to form a second level internet protocol.

Normal practice only uses at most one internet protocol. Many devices, such as SCSI, don't have either an internet or a network capability built into them.

Layer 5 — Transport Layer

A subsystem may have multiple physical links that implement a class of I/O; there may be two SCSI busses, two serial devices, etc. They each possibly have network addresses that may be redundant. The transport layer provides an addressing umbrella that gives some type of logical naming to each of the active devices to maintain uniqueness to the system software. This layer also routes requests between the upper layer software and the destination media connection.

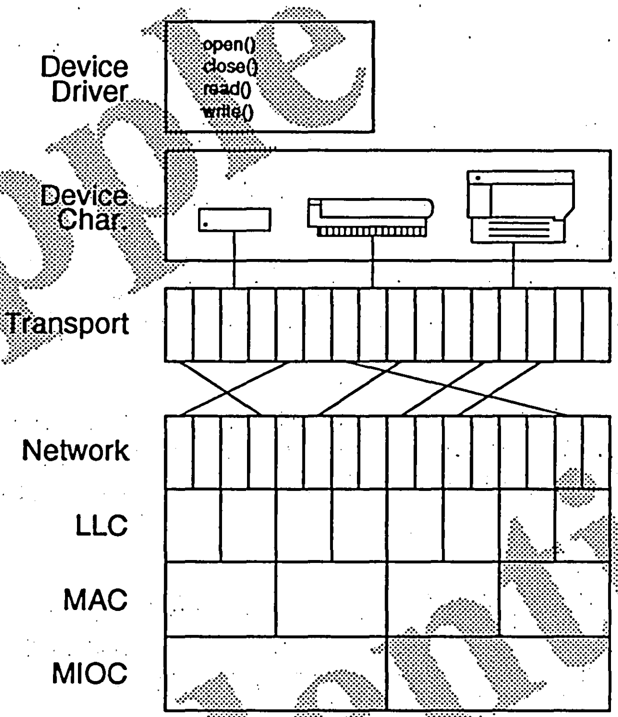


Figure 9. Addressing Hierarchy of Arioso

This layer is often responsible for guaranteed end to end delivery of information. For data communication protocols, this layer will try to send a message to the addressed device some number of times before giving up and classifying the device as unreachable.

Layer 6 — Device Characteristics

All of the previous layers have focused on delivery of data and information between one node and another. Within a device driver, there is usually a logical representation of the actual entity at the other end. Disk drives are assumed to have different characteristics than scanners; modems are different than MIDI synthesizers. These characteristics depend on the device, not on the operating system.

This layer captures the unique elements of any particular device. For instance for a hard disk, we may read, write, format, and check if the media is inserted into the device, whether it is mounted, etc... The actions do not depend on whether it is a SCSI hard disk, or an AppleShare volume.

Layer 7 — Device Driver

The final layer addressed by Arioso is the device driver interface. This interface is different for every operating system. Apple currently maintains at least four of these interfaces, even though the underlying devices are the same.

Most systems use a file paradigm. One can open the device driver, read and write to it, close it when done, and send device-specific control commands to it.

There may be several device driver layers within the device driver. A streams device driver in unix has several protocol layers that can be placed on top of the base device driver, but below the final device driver. Also, device drivers can call the services of other device drivers, as the MacOS AppleTalk device drivers do.

In this project, we will only be concerned with the base device drivers. We may find that there is no Transport layer in the LocalTalk driver, for instance.

Call implementation

The actual implementation of the model interface could be hardware, software, or a different processor. How can these all be described in a consistent manner, so that hardware/software tradeoffs do not disrupt the model?

We chose a Command-Return-Result paradigm for this purpose. Possible implementations of these interfaces are shown here. For synchronous commands, the return and the result occur at the same time. For asynchronous commands, the return is immediate; a completion routine will invoke the result.

Hardware

A hardware "command" is usually a single bus cycle. The peripheral devices observes when it is written to, and uses that as a signal to initiate some activity. The return takes place at the end of the bus cycle. If other bus masters delay the beginning of the bus cycle, the duration of the call is said to increase. The call processing time is measured from the beginning of the attempt to read an address to the beginning of the next attempt. Completion routines are implemented by interrupts, which typically invoke interrupt handlers.

An example is a SCSI chip command register write. There is a single bus cycle to write the command, the return occurs when the bus cycle is over, and result occurs when the device interrupt signals that the command is complete.

Software

A software command is either a normal subroutine call, a jump table jump or the beginning of a macro. The end of the call is said to occur when the next instruction after the call begins.

The completion routine is called by a lower layer using an address given to it by the calling module. The completion routine is also known as a software interrupt (SWI).

Coprocessor

A coprocessor may implement a lower layer. In this case, the call is the sending of a message of some kind to the alternate processor. If the caller issues the command synchronously, the call time is measured from the call till the time that the process wakes up and moves to the next instruction. If the caller does not block, the time is measured from the start of the sending of the message until the message transfer is completed, and the processor can go on to its next task.

Completion routines may be implemented by the coprocessor, or the coprocessor may send a signal to the main processor, and signal it to execute the completion routine.

Flowcharting

This sections presents a description of a set of flowcharting symbols suitable for describing multithreaded I/O drivers. It has symbols for blocking waits on I/O, and for passing control from one layer to another in the layered model of I/O systems.

The Arioso project needed a way to document the subtle timing characteristics and control flow that is found in I/O drivers. Since so much of the control flow is tied up in interrupt handlers and I/O waits, this posed a difficult problem.

We set out find a set of symbols that could describe these code and machine operations that would at once convey the basic information, avoid excessive detail, identify blocking I/O waits and explicitly show interactions between Arioso layers.

Standard flowcharting symbols have no special symbols for code blocking or for interactions between layers. Therefore we have defined 8 symbols in addition to the standard block and decision symbols common in standard flowcharts.

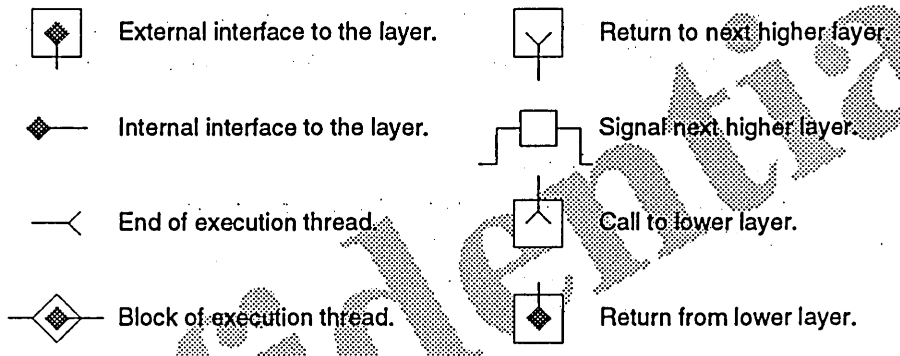

External interface to the layer.
Return to next higher layer.
Internal interface to the layer.
Signal next higher layer.
End of execution thread.
Call to lower layer.
Block of execution thread.
Return from lower layer.

Figure 10. Arioso Flowcharting Symbols

Description of the Symbols

- **External Interface to the Layer**

  This identifies a call that upper layers can make to this layer. It is shown as a vertical interface to emphasize the fact that the thread of execution comes down from a higher layer.

- **Internal Interface in the Layer**

  This identifies a control transfer point that may be reached by an interrupt or a software interrupt affecting the task scheduler. Thus an interrupt may cause execution to suddenly change from wherever it was to the point identified by this symbol.

- **End of Execution Thread**

  This identifies a control thread end. At this point the task effective ends its life. A new call or an interrupt or other asynchronous event are required to generate a new execution thread in the layer.

- **Block of Execution Thread**

  This identifies a pause in the execution. This is an explicit wait on I/O. There are a variety of ways that this can be implemented. In a multitasking system, it can be a block on I/O. In a single threaded system, it can be a busy wait on a status bit. It can also be implemented as several variants in which polling is done for a short while to catch responses that happen quickly, and for those responses that take longer than this period of time, a block is done.

- **Return to Next Higher Layer**

  After a service call to an external interface to a layer, control will return at some point back to the upper layer. This symbol indicates when this will happen.

- **Signal Next Higher Layer**

  If a completion routine, software interrupt, process restart or other signaling is to be sent to a higher layer as a result of I/O being completed, this symbol is used.

- **Call to Lower Layer**

  If a layer requests the services of a layer, it effectively transfers control to that layer. At some future point in time, control will return to the calling layer. This symbol shows the transfer of control to a lower layer.

- **Return from Lower Layer**

  If a layer requests the services of a layer, eventually it will return control back to the calling layer. This symbol is used to indicate that this has taken place.

Flow Charting Example

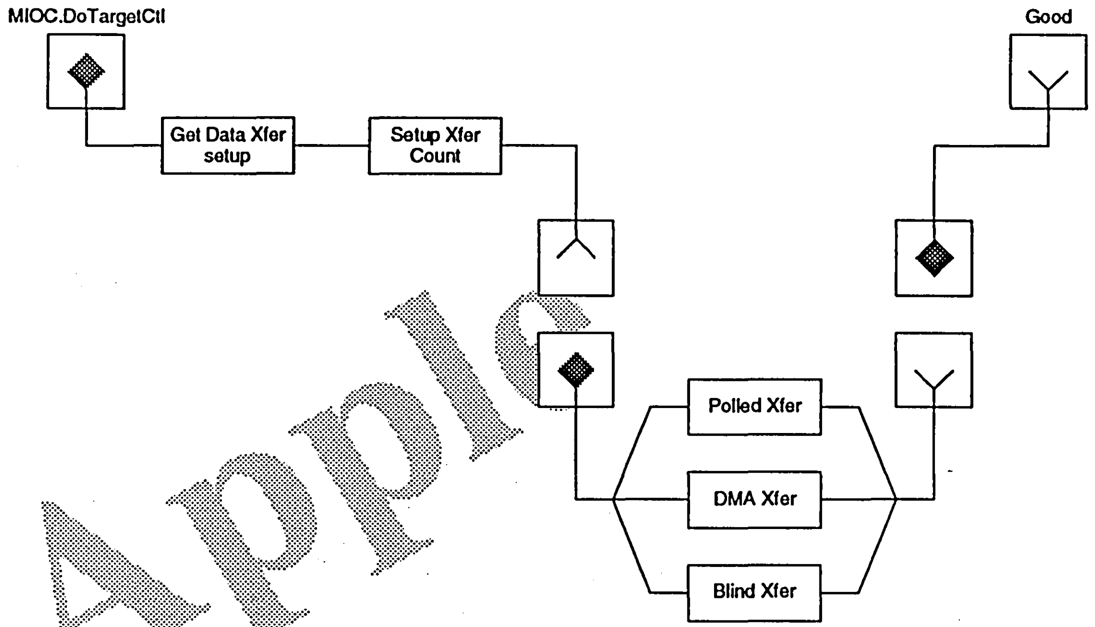


Figure 11. Example of symbols

In the example shown above, the MIOC.DoTargetCtl is an external interface to the layer. Control can transfer to this point from a service call from a higher layer. Upon receipt of the thread of control, the GET DATA XFER SETUP block is run, followed by the SETUP XFER COUNT block. Blocks can represent collections of symbols; so effectively each block will be described by another diagram. For simplicity, often these internal block diagrams will not be shown.

The layer then makes a call to a lower layer for data transfer services, using an external interface of the lower layer. If will execute one of the three data transfer routines synchronously, and after completion will return to the calling layer.

The upper layer now has control again and having finished its service to the original calling layer, returns and ends the description.

Chapter 2

Apple Desktop Bus

Version 1.0

Henry Kannapell

About This Chapter

This chapter describes a proposed ADB layering and interface for the ADB subsystem of the Macintosh. The chapter begins with an overview and then covers details of ADB. This specification divides ADB based device drivers into interface layers.

The ADB described here is a model; the actual calls and interface functions are not necessarily the actual ones that are used in the current or future ADB code. This document is a logical breakdown of the ADB protocol for Apple implementations. The goal is to define elements of the protocol that can become CPU independent.

Thus, DON'T THINK THAT ADB HAS TO HAVE LOTS OF LAYERS. It should probably have three layers (compared to the 6 or 7 it currently has). These should be the device driver (Mouse, keyboard), The Transport and Mac layers (the hardware independent portion of the ADB Manager) and the MIOC layer (hardware dependent data transmission and reception).

The HAL (Hardware Abstraction Layer) being considered in the RISC group is essentially the same as the MIOC layer.

Glossary

ADB terminology is defined in this section.

'ADBS'	• special resource that contains ADB device drivers.
'KCHR'	• ASCII translation; maps virtual keycodes to ASCII characters
'KMAP'	• Keymapping resource; maps keycodes to virtual keycodes
ADB	• Apple Desktop Bus
Autopolling	• Continuous retransmission of some innocuous command to allow devices to post SRQ's

Device Handler	• an 8 bit number that defines the behavior of registers 0 through 3 on a device.
Device ID	• The bus address of an ADB device
Layer	• Normally a consistent collection of subroutines, traps, or macros that provide the only interface necessary to a subsystem.
PIC	• Peripheral Interface Controller, 6502 used to control I/O devices
SRQ	• Service Request, a device signal to request polling from the ADB master
SRQ Autopolling	• After an SRQ has been detected, the system goes into a round robin search through the active ADB devices to service the one that posted the SRQ
VIA	• Versatile Interface Adapter. A I/O port expander with a simple shift register circuit

About ADB

The Apple Desktop Bus connects up to 16 low-speed primarily input devices to the Macintosh. Each device connected to the bus can contain up to four variable sized registers, whose contents can be read from or written to by the ADB master. Each register may contain 2 to 8 bytes of information.

There is assumed to be exactly one master of the ADB network, which is usually the Macintosh. All other devices accept data or send data based on commands from the master. There is a mechanism for a global service request broadcast that will request polling from the master. This in an unidentified request. The master responds by polling all devices for input.

ADB has no inherent mechanism for error control. Even though it is a physical medium, there is no provision for being able to detect transmission errors or losses. For keyboards and mice, it does not matter that information is occasionally lost or altered. For other types of transmission over this media, higher level error handling protocols must be used to ensure reliable delivery.

One of the goals of the Arioso project is to identify layers that can be implemented by the design centers instead of by system software. For the ADB system, the most probable split is the MIOC layer.

Implementation Tree

Now let's consider the current Macintosh ADB system. There are several interfaces to ADB code modules. These are related to the different implementations of the ADB. A calling graph of the different implementations is shown in the Appendix.

The original one is the "GI" (General Instruments) circuit. This used a 4 bit microcontroller/transceiver to modulate the bit stream to the ADB format, and would automatically repeat the last transmission. The shift register of the VIA was used to serialize data and present it to the transceiver.

The next design was the 6502 based PIC design found on the IIfx. This used the 6502 to control much of the protocol, and used shared static RAM to communicate with the 68030. A message passing architecture was designed at this point to support the processor to processor communication.

Another design is the one for the Macintosh portable computer. In this design, a keyboard matrix was interfaced to a Mitsubishi M50740 microcontroller, which encoded the keyboard and maintained status as an ADB node. This was then connected to the general purpose ADB connector, which connected to the power manager IC, another microcontroller. The Power Manager actually implemented the ADB master protocol. This power manager connected through a parallel interface to a VIA, which could then be read by the host 68000 processor. This was another message passing architecture, distinct from the PIC chip implementation.

The next design was the Egret based designs that use a 6805 microcontroller to modulate the ADB signal in addition to performing power management functions and I/O bit space expansion. This still used the VIA shift registers to transfer data to the 6805 in packet form. These packets were then transferred over the ADB to the destination node. This is yet another message based protocol.

A minor variation of the Egret design was created with the Cuda chip, as slightly different I/O bits were controlled.

Finally, a larger power manager was created with the PG&E chip for DB-Lite. This chip handles power management for the system, I/O port expansion, modem processing as well as ADB processing.

An implementation tree of the current and planned SCSI designs is shown below. This shows the type of ADB controllers versus the types of CPUs.

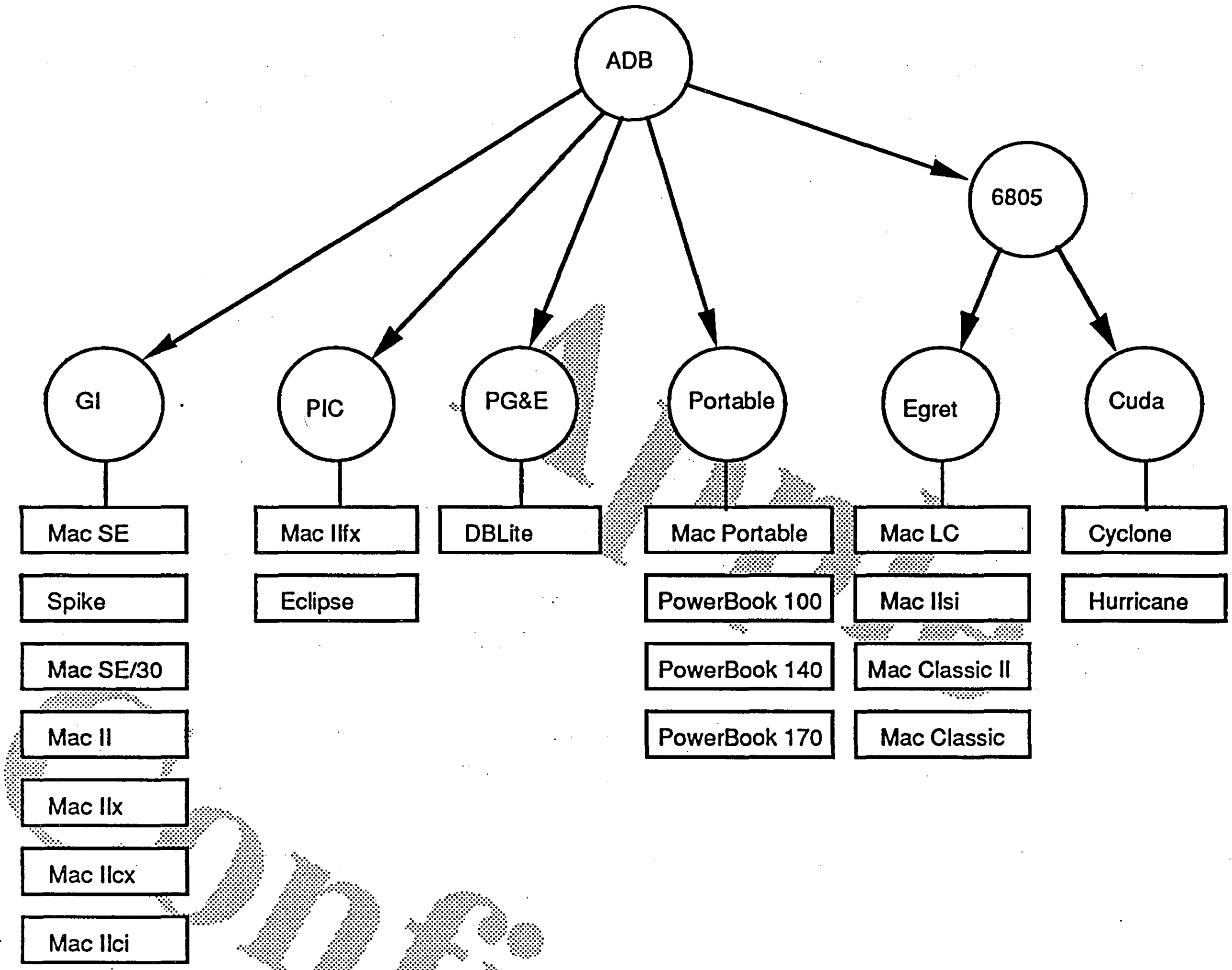


Figure 2-1 ADB Implementation tree

Arioso Model of ADB

The Arioso layers of the ADB system are defined in the remainder of this chapter. Each layer has an interface to the layer above it and an interface to the layer below it. The way that a layer is realized is called an *implementation.*

The layers are described from the I/O devices at the bottom, up to the device driver interface at the top.

Note that an entity in the device characteristics layer, such as the keyboard, can attach to more than one ADB device. This will happen in particular for the Norsi Keyboard.

Also note that there can be several different device driver clients of a single device characteristic entity. This can happen if special keys wish to be sent out as if they are their own device. This occurs for instance with sound modifier keys, such as a key for volume control.

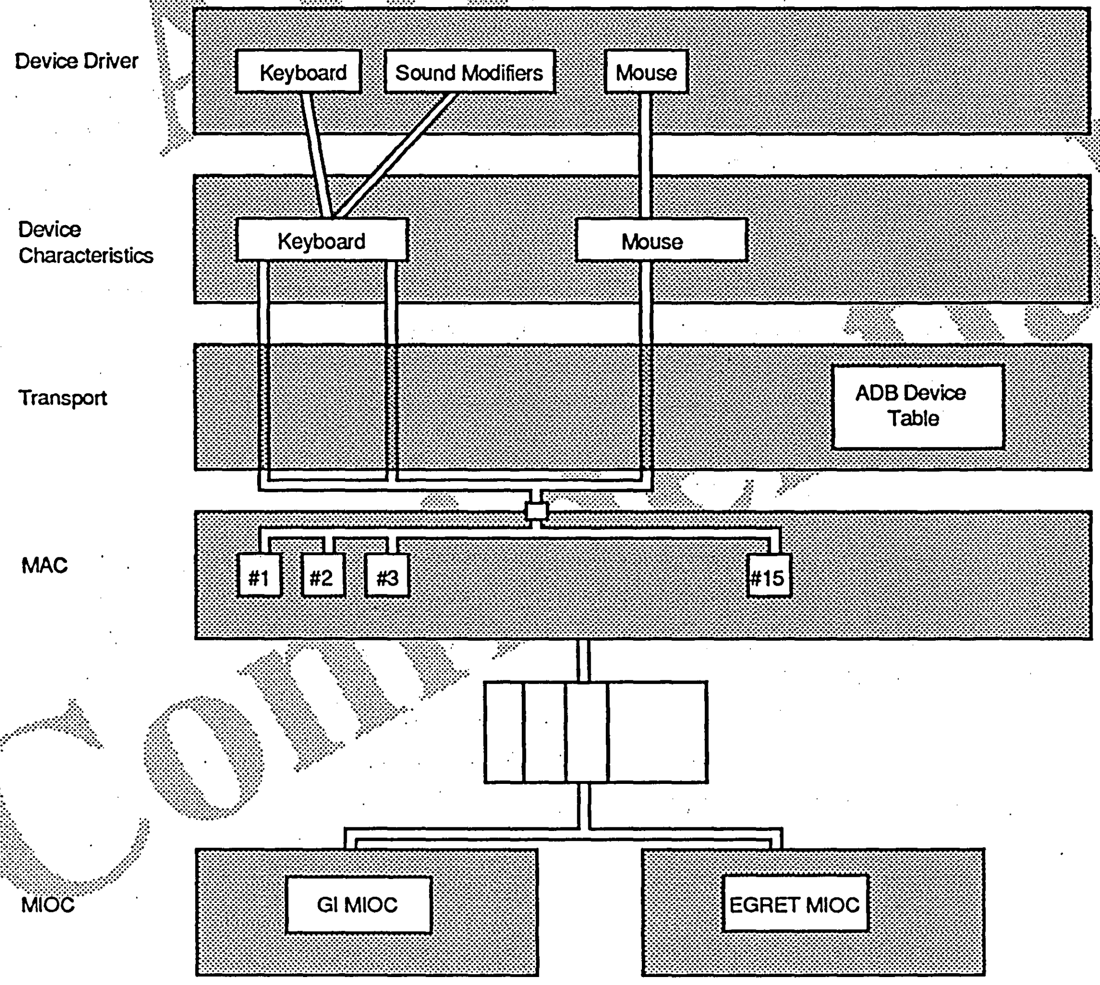


Figure 2-2. ADB layering model

Physical layer Description

At the bottom of the device driver, there is the physical signalling that arbitrates for the bus, selects the target, and transfers information. This information is documented in the *Guide to the Macintosh Family Hardware.*

ADB MIOC layer Description

The MIOC layer is responsible for sending formatted ADB data packets down the ADB Bus and receiving packets from the bus. There are no unexpected ADB data packets that come in to the system; it is a strict Master-Slave polling relationship. Slave devices can issue a service request however; this is a special physical layer coding that results in a perceivable indication to the Macintosh.

The Macintosh can send out data packets of variable length, with a maximum of 9 bytes. The Macintosh can receive either 8 byte data packets in response to a request for data. Any transmission over the bus can be modified to include Service Requests.

The ADB MIOC is the point of departure for most of the implementations. In the case of the GI , EGRET, Power Manager, and IOP chips, the full ADB packet is sent over the message passing protocol unique to each of these systems. Any responses are returned. The hardware only Condor implementation would also use the MIOC as the interface point. All ADB code above the MIOC should be common.

Currently a buffer queue is maintained between the MAC and the MIOC layers. MAC frames are written to the queue, and then control returns to the caller. There is a hardware dependent "StartReq" and "ReqDone" for each different implementation. These are similar to the MIOC framing layer calls defined in this document.

ADB Mac Layer Description

The MAC layer maintains ADB transactions across the bus by sending commands and receiving data. Some ADB transactions are single ended; there is no acknowledgement and no return data for some of the packet types. Only the TALK command will have data returned to it. Any transaction can have a Service Request indicated at the end of the packet. Any number of Service Requests may be associated with the same actual request.

The MAC layer sends data packets to and from the devices on the bus. They take as parameters the address of the device and the data packet if any to be sent to it. There are four kinds of commands, the same four that ADB defines - RESET, TALK, LISTEN, and FLUSH.

ADB LLC Layer

Any ADB device may have up to 4 registers that can be used as the target for talk and listen commands. These amount to identifiable points within the ADB device, and thus are part of the LLC layer. Normally only two of these registers are used. Register 0 is used for data transmission and reception. Register 3 is used for configuration and control information. Registers 1 and 2 may be used by an individual implementation for any reason. Apple keyboards use Register 2 to identify the current state of the modifier keys, which are the command, control, shift, etc. keys. Newer versions of the Apple keyboard drivers do not use this information, but rather synthesize it from keystroke histories.

While technically the network layer exists, from a practical standpoint it does not. If registers 0,1,2 and 3 could be spread to multiple physical devices (quite possible, of course), a network layer definition might make sense. In reality, connections are only made to one physical device at a time - all registers are associated with the same device.

Thus the registers will be treated as simply part of the device driver protocol, similar to special control frames for distinguishing commands from data in traditional networking protocols.

ADB Network layer

There is no network layer defined or used in any ADB implementation.

ADB Transport layer

Currently there is only one ADB bus per machine. Putting the previous layers together, we see that we can represent up to 16 devices on the bus (including the Macintosh), each with 4 internal registers. Each device that is open on this bus can send or receive data with the master on the ADB bus. In addition, each device has a separate command and control channel that is used to manage the device. The interface then looks like 16 main ports, each with two subports.

The transport layer maintains a map of all currently connected devices. Most ADB devices operate in an input only mode - they present a continuous stream of data to the Macintosh. Control information is sporadically sent to the device.

There is no mechanism for dynamically adding devices to the system. Dynamic addition poses extreme difficulties because of the addressing architecture of ADB. Since any new device may have the same address as a currently operating device, there is no obvious way of identifying the fact that a new device has been added. In addition, ADB circuits normally draw power from the ADB bus, and are not normally designed for hot insertion.

Clearly the system can respond to a request to identify all devices that exist on the bus, but devices cannot be added dynamically.

ADB Device Characteristics Layer Description

The DEVICE CHARACTERISTICS layer defines the useful attributes of the device being addressed. This layer is not specific to a particular connection media; instead it is a logical representation of the attached device.

The primary types of ADB devices are keyboards and mice. In addition, various security devices have been implemented on the ADB, and there is even a 9600 baud fax modem which works over ADB.

Typically these characteristics are embodied in a driver that is loaded at system boot time. The system will search for 'ADBS' type resources and will load them and associate them with physical ADB devices. A description, though incomplete, of this may be found in *Inside Macintosh, vol 5.*

This driver interface is subject to review and possible change at the current time.

ADB Device Driver Layer Description

The DEVICE DRIVER Layer takes the ADB specific device driver interface and passes control to the ADB driver itself, the 'ADBS' code referred to earlier. The normal output of this device driver is event records which are posted to the event queue for processing by the application. Note that this device driver interface is entirely separate and unrelated to the operating system device driver interface. This is a form of connection oriented data service in which the data travels in only one direction.

In the future this may be a standard driver interface. This is the interface for any input devices to the Macintosh. Thus, if a completed different sound modifier system (knobs or something like that) is implemented, one would expect the device driver interface to remain the same, and the rest of the system to be unable to determine that a new circuit is present.

Addressing Structure

The addressing structure of the ADB I/O system can be viewed as a tree either from the root to the leaves or from the leaves to the root. The difference is in whether one considers physical devices or the logical representations of these devices. Arioso views each layer as representing the devices at the lower layers. Thus it views the I/O system from the leaves to the root, with the actual media connection being the root. For those who wish to view the system as physical devices, the media connection is the root, and each device, and register are leaves. The correspondence between these two views is shown in figure xx.

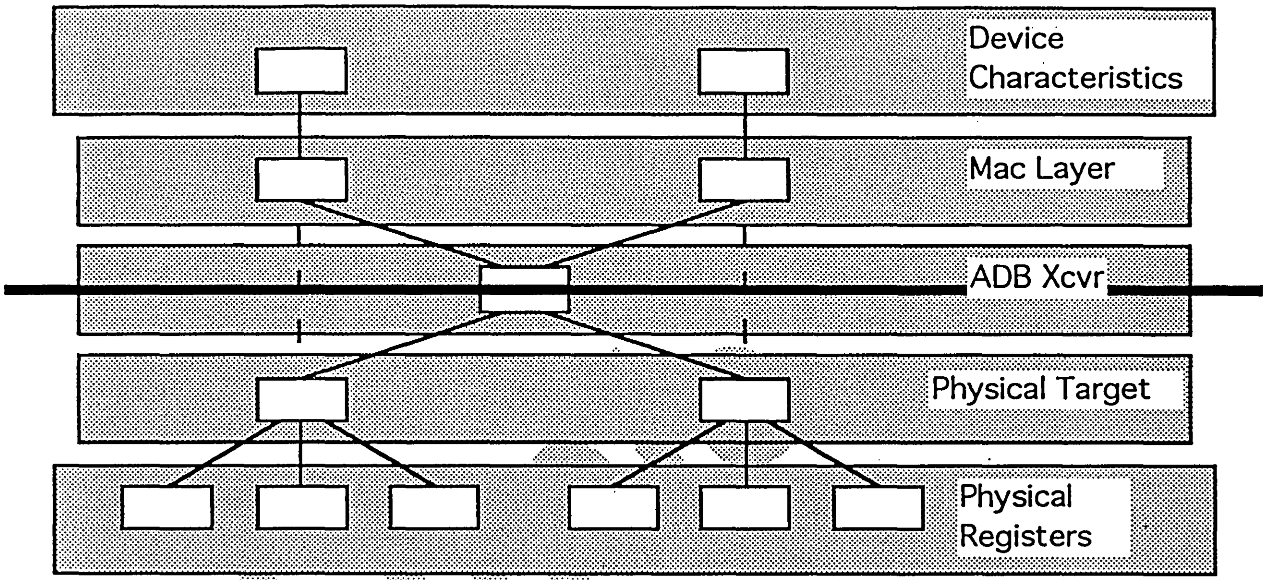


Figure 2-3: Addressing structure of the ADB I/O System

Addressing in the system is done with the first byte of the ADB packet. The primary addressing is done with the first four bits, which can select one of 16 targets. The second level addressing can address one of 4 registers. Notice that the figure for second level addressing shows the complete address, i.e. includes the device address.

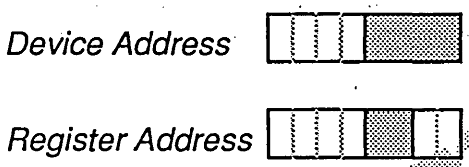


Figure 2-4: Addressing terms of ADB

The specification describes the ADB system in three ways. First, the *semantics* of the main interface calls is the description of what the call does, or what "functionality" it has. This is described at the Operations section of each call and in the written description at the beginning of each section. Since there is no formal method for describing this currently accepted by the industry, this section will be described by several properties of the call. State diagrams may be used to further illustrate the operation of a layer.

Second, the *syntax* of the interface functions specifies what services can be requested and what parameters are used to implement the call. This is described in the call header of each subroutine. The parameters will be shown in set notation to indicate that the order of the parameters and their exact type are not important to this specification. They *are* important to the implementation specification.

Third, the *timing* of each call is the description of how long the call takes minimum and maximum, and what underlying timing is required for the subsystem below it. This is shown in the timing requirements at the end of each layer definition. There are two important aspects to timing. First, what is the time that must be met or guaranteed for the system to work correctly? Second, what is the time that it take a particular implementation to perform a task. The first is a set of constraints that the implementation must meet. The second is a measure of the performance of the system.

Device Access Layer in Detail

The device access layer describes the low level access to the ADB chips themselves. While many calls might be defined, only the access to the ADB indication and collusion detection are included here.

Device Access Calls

DA.ServiceRequest() **Indication**

Indication	A device in the ADB device space requests service. It may be more than one device who requests this service. Multiple indications may be made for a single service request. Service request is assumed to be globally visible up to the transport layer

DA.Collusion() **Indication**

Indication	The ADB master cannot detect a collusion. However, the ADB devices must be able to detect a collusion as part of the automatic address reassignment algorithm.

DA.GlobalReset() **Request**

Command	Issue a Global Reset signal on the Bus. This will force all devices to a initial power on state.
Return	Return to Caller after queuing the packet
Result	{Success}. When completed, the call returns success

MIOC Layer in Detail - General Instruments Circuit

1. MIOC Control Layer

This description will be based on the General Instruments ADB transceiver chip and via combination. This is a serial shift register with a serial modem 4 bit microcontroller.

The Media Input/Output Control Layer addresses the media and attempts to transfer data bytes out over the media interface, and receive them from the media interface. There are two basic forms of this transfer - data that is sent out only, and data that is sent out that will expect return data from the device.

Currently the ADB specification restricts data transfer to 8 bytes and one byte for the header; thus there is a maximum of 9 bytes that can be sent. In the second case, a single byte transmission may be followed by up to 8 bytes being received.

Devices may append a Service Request at any stop bit in the transmission of data. Two things to note are 1) since several stop bits exist in a frame, there may be several Service Request indications in a single transmission, and 2) devices cannot issue service requests unless transmissions exist. Therefore, some form of autopolling must take place to allow devices to add service requests. Autopolling is addressed in the transport section.

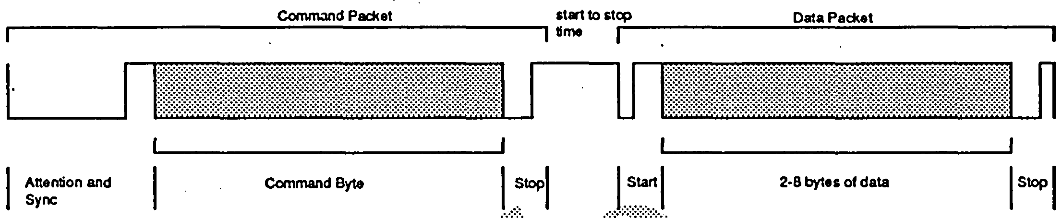


Figure 2-5. ADB data packet

From the MIOC's standpoint, there are two transmission slots; a command slot and a data slot. The Macintosh always transmits something in the command byte. For two of the commands, FLUSH and RESET, that is all that is sent. There is no data associated with these packet types.

For the LISTEN packet, 2 to 8 bytes of data will follow the command byte. These bytes are sent by the Macintosh. Note that there is no start or stop bits between the bytes of the data packet. Flow control between the VIA and the ADB microprocessor is managed by the state bits that are set by the VIA.

For the TALK packet, 2 to 8 bytes of data are returned by the device. The same timing constraints hold here also. The stop signal is used to define the end of transmission. Thus a variable number of bytes can be received. In the current implementations, there is no timeout if the stop signal never occurs; thus the ADB device can stall waiting for a signal that never comes.

There are two state bits that are used to signal the ADB transceiver. They are: ADB.ST[0:1]. These are used for handshaking control and byte identification between the CPU and the ADB microcontroller. They are not part of any actual data transmission. Their values are shown below:

ST1	ST0	Transaction state
0	0	0: Start a new command
0	1	1: transfer data byte (even)
1	0	2: transfer data byte (odd)
1	1	3: Idle

An interrupt that is generated by the ADB transceiver whenever a service request is posted by a ADB device.

Finally there is an ADB clock and data signal for sending serial data between the VIA and the ADB transceiver. This is clocked by the ADB transceiver.

There is a queue of pending ADB requests between the MAC and the MIOC layers. The StartQueueProcessing call will initiate the processing of these. The head of the queue is removed and the transmission initiated, and the resulting data returned (if any). The queue is then checked to see if it is empty. If so, the process stops; otherwise, the next element is taken and the processing continues.

MIOC Control Layer Calls

MIOC.StartQueueProcessing() — Request

Command	Begin taking elements from the Queue and issuing the ADB commands
Return	Return to Caller
Result	{Success}.

MIOC.QueueEmpty() — Indication

Indication	The Queue has become empty. The Mac must issue a StartQueueProcessing in order to restart the Queue processing.

2. MIOC Framing Layer

The MIOC Framing layer will take the calls and sequences from the MIOC layer and convert them into actual bus sequences. The MIOC has decided what will be done, and then calls the services of the MIOC Framing to actually accomplish them.

MIOC Framing Layer Calls

MIOC.F.GetNextPacket() -> ([ADBBufPtr], Length) — Request

Command	Get the next packet from the packet queue.
Return	Return to Caller
Result	{Success, Empty}.
Operation	get the next element from the Queue. Either it returns a packet or empty because there is no more packets.

MIOC.F.WritePacket([ADBBufPtr], Length) — Request

Command	write the current packet to the ADB transceiver - will call a sequence of WriteCommand, and WriteData calls.
Return	Return to Caller
Result	{Success}.
Operation	send the data to the ADB transmitter. This will in turn call WriteCommand and WriteData.

MIOC.F.ReadResponse([ADBBufPtr], Length) — Request

Command	get the current data resulting from a TALK command.
Return	Return to Caller
Result	{Success , Timeout}.
Operation	Either the data was successfully read or a timeout took place.

MIOC.F.SetState({Command,Even Byte, Odd Byte, Idle}) **Request**

Command	Set the transfer state to one of the above. This controls the protocol between the VIA and the ADB receiver.
Return	Return to Caller
Result	{Success}. There is only success .

MIOC.F.WriteCommand([Byte]) **Request**

Command	write the current command to the VIA for transmission to the ADB modem.
Return	Return to Caller
Result	{Success}.
Operation	Write a byte into the VIA shift register. The byte will be shifted out when the state is set to command. Used internally by MIOC Framing.

MIOC.F.WriteData([Byte]) **Request**

Command	write the current byte to the via shift register for transmission
Return	Return to Caller
Result	{Success}.
Operation	Write a byte into the VIA shift register. The byte will be shifted out when the state is set to command. Used internally by MIOC Framing.

MIOC.F.ReadData([Byte]) **Request**

Command	get the current byte from the via shift register . This will wait for a byte to be sent from the ADB transceiver. If it doesn't occur within the time window, then a timeout is returned.
Return	Return to Caller
Result	{Success , Timeout}.
Operation	After some indication that a byte exists in the shift register, read it out or timeout. Used internally by MIOC Framing.

MIOC.F.AbortCommand() **Request**

Command	Stop the transfer of the current command by setting the transfer state to Command.
Return	Return to Caller
Result	{Success}. The current command is aborted

MIOC Timing Requirements

Dynamic Characteristics

Symbol	Parameter	Min	Max	Units	Test Conditions
tWCID	Write command max interrupts disabled	--	20	us	
tDDAD	data byte to next data byte on ADB	70	130	us	
tSP	time to send a packet composed of N bytes		2165+800 *N	us	8.565 msecs for 8 bytes
tTO	time to detect timeout		2.025	ms	time to send a command and then find no response.
tDDSR	data byte to next data byte on VIA		–	ns	no restriction

MIOC Layer in Detail - Egret

In the original design, the MIOC framing level was the point of departure for the various hardware specific portions of the ADB manager. This document considers the MIOC control layer to be the point of departure. This is primarily because the Egret can receive results of explicit TALK packets asynchronously, where the other versions of the interface receive them synchronously. Also, the Egret can have a collusion when trying to send a packet that other implementations don't have. Finally, Egret does SRQ polling automatically, where the GI has to SRQ poll manually. At the Queue level, everything is the same.

1. MIOC Control Layer

This description will be based on the Egret ADB circuit and via combination. This is a serial shift register with a serial modem 8 bit microcontroller.

The behavior of this layer is similar to the previous description. The difference is that when a packet is taken from the queue, it is passed to the Egret instead of to the GI chip. Both are microcontrollers that will do autopolling. Egret, however, will do autopolling.

Sending data to the Egret is somewhat more complicated, as collusions can occur that the MIOC must recover from; also Egret can have other clients besides ADB. The latter point means that ADB commands must be reformatted to include header information. See the Packet ADBu spec for a description.

The header for Egret is a single byte that identifies the data packet as an ADB command. Commands are sent to the Egret, and data is asynchronously returned from Egret after processing. There should be only one outstanding ADB request at a time.

There are three state bits that are used to control the transfer to Egret. They are: VIA_FULL, SYS_SESSION, XCVR_SESSION. They are defined in the Packet ADB micro ERS.

There is an interrupt that is generated by Egret whenever an asynchronous packet is ready to be sent from Egret.

Finally there is an ADB clock and data signal for sending serial data between the VIA and the Egret. This is clocked by Egret.

There is a queue of pending ADB requests between the MAC and the MIOC layers. The StartQueueProcessing call will initiate the processing of these. The head of the queue is removed and the transmission initiated, and the resulting data returned (if any). The queue is then checked to see if it is empty. If so, the process stops; otherwise, the next element is taken and the processing continues.

MIOC Control Layer Calls

MIOC.StartQueueProcessing() **Request**

Command	Begin taking elements from the Queue and issuing the ADB commands
Return	Return to Caller
Result	{Success}.

MIOC.QueueEmpty() **Indication**

Indication	The Queue has become empty. The Mac must issue a StartQueueProcessing in order to restart the Queue processing.

2. MIOC Framing Layer

The MIOC Framing layer will take the calls and sequences from the MIOC layer and convert them into actual bus sequences. The MIOC has decided which elements will be done, and then calls the services of the MIOC Framing to actually accomplish them.

MIOC Framing Layer Calls

MIOC.F.WritePacket([ADBBufPtr], Length) **Request**

Command	Write the current packet to the ADB transceiver
Return	Return to Caller
Result	{Success, Collusion}.
Operation	Send the data to the ADB transmitter. Either it is successfully sent, or a collusion is indicated and it must try again.

MIOC.F.ReadPacket([ADBBufPtr], Length) **Request**

Command	Get the current packet from Egret. It is assumed that Egret has already signalled the Macintosh that a data packet is available.
Return	Return to Caller
Result	{Success, Timeout}.
Operation	Either the data was successfully read or the system failed and timeout.

MIOC.F.AbortCommand() **Request**

Command	Stop the transfer of the current command by setting VIA_FULL and SYS_SESSION off.
Return	Return to Caller
Result	{Success}. The current command is aborted

MIOC Timing Requirements

Dynamic Characteristics

There are currently no timing constraints on the system. Interrupts do not have to be turned off.

MAC Layer in Detail

1. MAC Control Layer

The Mac layer creates support for addressing different devices on the ADB bus. There are four interfaces here: RESET, which will reset all devices on the bus, LISTEN, which will send some data packet to the specified device, TALK, which will cause the device to output data, and FLUSH which does a soft reset of the ADB device, clearing all data buffers. There is still a notion of a single connection taking place at any time here. One can TALK to any node in the ADB space, but only one at a time.

There are four basic packet formats, which are shown below.

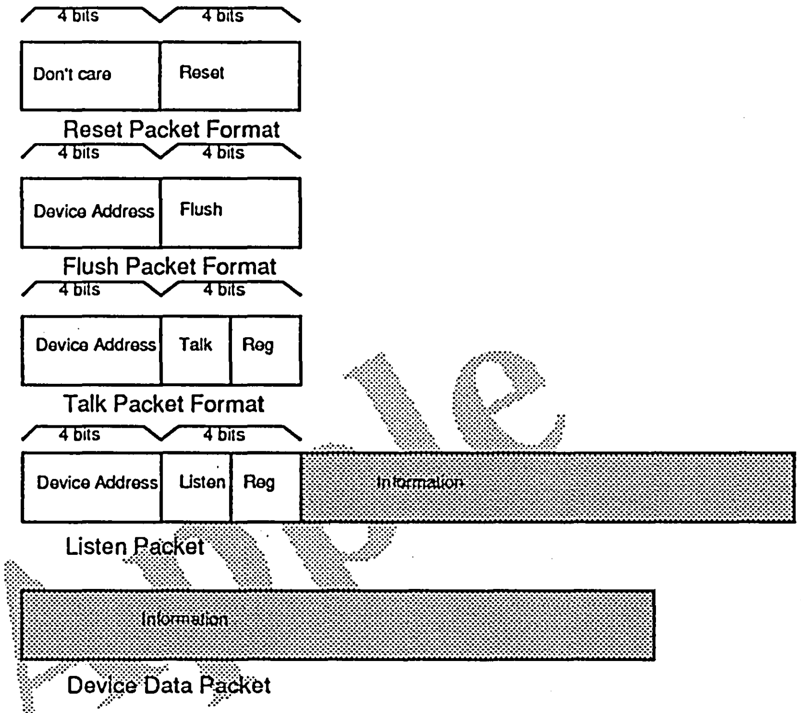


Figure 2-6. ADB packet formats

There are four packet formats defined at this level.

The RESET Packet uses one byte composed of 4 unused bits followed by a 4 bit reset indication.

The FLUSH packet has a 4 bit address field, and FLUSH type.

The TALK packet has a 4 bit address field, a 2 bit TALK type, and 2 bits for the register to be addressed. A Device Data Packet can be expected to follow this command.

The LISTEN packet has a 4 bit address field, a 2 bit LISTEN type, and 2 bits for the register to be addressed.

The last packet type is the Device Data Packet, which has no header. It is strictly a stream of data bytes, given in response to the Talk command if there is available data.

Reset Sequence

In order to reset all of the devices on the ADB bus, a broadcast reset packet is sent out. The framing layer actually creates the bytes for the packet and the MIOC is then called to send it out.

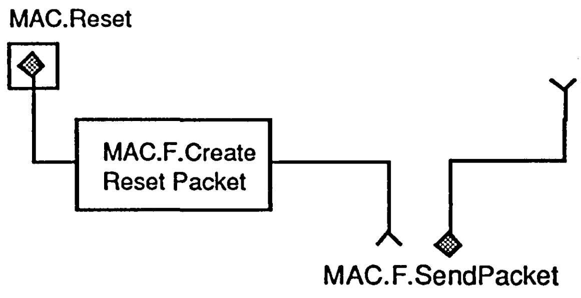


figure 2-7. Mac layer reset sequence

Flush sequence

In order to soft reset an individual device on the ADB bus, a flush packet is created based on the device ID, and sent out using the MIOC.SendPacket call

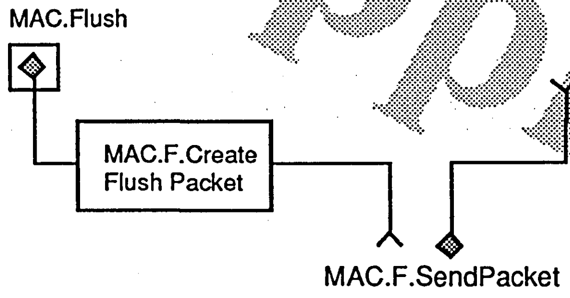


figure 2-8. Mac layer flush sequence

Talk sequence

The Talk sequence takes the information in the command, formats a talk packet, and enqueues it into the Queue between the MAC and MIOC layers. Included in the Queue entry is an address to put the data and count into, should any data exist.

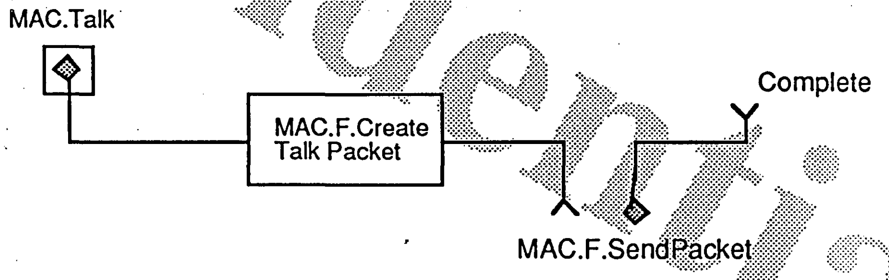


figure 2-9. Mac layer Talk sequence

Listen sequence

The Listen sequence takes data from the command, formats it, and places it in the queue between the Mac and the MIOC layers. The Command then returns as complete.

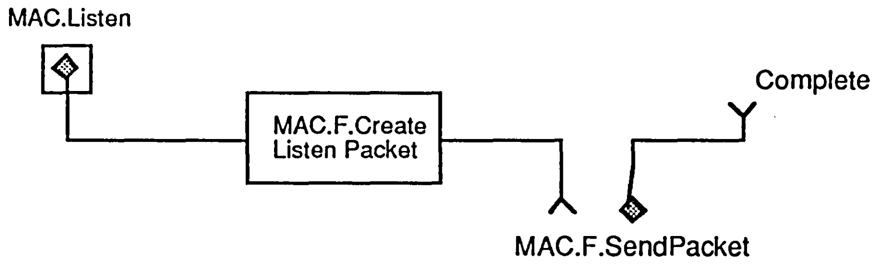


figure 2-10. Mac layer Listen sequence

MAC.TALK([TargetID], [RegNum], [BufPtr]) **Request**

Command	Issue a talk command and wait for the resulting data to appear. Put it in the buffer pointed to by BufPtr. A maximum of 8 bytes can be returned.
Return	Return to Caller
Result	{Complete,Timeout}. The command will either end with a timeout or with the buffer containing data.

MAC.LISTEN([TargetID], [RegNum], [BufPtr]) **Request**

Command	Issue a listen command and sends the data in the buffer. A maximum of 8 bytes can be sent. There is some interest in making this number target than 8 to enhance throughput. The MIOC might then break the command into 8 byte pieces.
Return	Return to Caller
Result	{Complete}. The command will be enqueued for transmission, and is assumed to eventually be transmitted.

MAC.FLUSH([TargetID]) **Request**

Command	request that the target clear itself
Return	Return to Caller
Result	{Done} The sequence is sent out over ADB to the target and done is returned. There is no check to see whether it was received or not, or whether it was received correctly. This command is device dependant; normally it clears internal buffers and drops all pending data.

MAC.RESET() **Request**

Command	request that all devices on the bus clear themselves.
Return	Return to Caller
Result	{Done} The sequence is sent out over ADB as a broadcast to all devices and done is returned. There is no check to see whether it was received or not, or whether it was received correctly. This command is device dependant; normally it clears internal buffers and drops all pending data.

2. MAC Framing Layer

The framing layer for the MAC layer will expand the different packet types by adding an address field, and possibly some packet type bits. There is one call here for each type of packet.

MAC.F.CreateReset() **Request**

Command	Form a RESET packet
Return	Return to Caller
Result	{Done}

MAC.F.CreateFlush([TargetID]) **Request**

Command	Form a FLUSH packet
Return	Return to Caller
Result	{Done}

MAC.F.CreateTalk([TargetID], [RegNum], [DataPtr]) **Request**

Command	Form a TALK packet, with a pointer to the destination of the data.
Return	Return to Caller
Result	{Done}

MAC.F.CreateListen([TargetID], [RegNum], [DataPtr]) **Request**

Command	Form a LISTEN packet, appending the data in DataPtr
Return	Return to Caller
Result	{Done}

MAC.F.SendPacket([ADBBufPtr], Length) **Request**

Command	Enqueue a packet for transmission over the ADB bus. The first byte is assumed to be a command. The length will dictate how long the transfer will be. This is a write only transfer; thus it will cover LISTEN, RESET, FLUSH, and the first half of the TALK commands.
Return	Return to Caller after queuing the packet
Result	{Success, Fail} + {ServiceRequest}. There is either success or an unexpected status.

Transport Layer

1. Transport Control Layer

There is normally only one ADB circuit per machine, so there is no routing function that needs to be done. There is no reliable data transmission provisions, so all transmissions can be considered datagrams, that is connectionless data transfer. The primary function of the transport layer in ADB is to map the physical devices and registers that exist on the bus with logical entities, so that communication paths are opened to the devices and automatic mapping to device drivers is done.

To facilitate this mapping of physical devices to logical points, some form of autopolling is done. This could be done by the transport layer, that would repeatedly issue TALK commands to the last addressed device. Current implementations use the ADB transceivers themselves. These automatically autopoll whenever no explicit requests have been issued to the transceiver for a time specified by tAP (see timing characteristics).

Since it could also be done manually (in some super low cost implementation) the autopolling is defined in the transport layer.

After a service request is received, polling of all currently enabled devices will begin. Each is issued a TALK Register 0 command, till the one that actually had the data is found. The device that posted the service request is expected to keep asserting Service Request until it receives the TALK Register 0 command.

Each Node appears as a two way stream of data. Data can be written to or read from any register in the ADB address space.

When a service request is posted, the transport layer will poll each of the active devices and return their data to the clients of the transport layer.

Transport Control Layer Calls

TRANSPORT.CreateDeviceTable() — Request

Command	issue TALK commands to register 3 for each ADB address. Then execute the remapping procedure defined in *Guide to the Macintosh Family Hardware.*
Return	Return to Caller. Create the device table to reflect the device mappings
Result	{Complete}.

TRANSPORT.Open(ADBDevice) —> Reference — Request

Command	Open a path to the defined ADB device . After this call, data may be sent to the device, or the device may asynchronously return data. A reference number is returned, which is the handle to the device from then on.
Return	Return to Caller
Result	{Complete, NoDevice}. Either it was successful, or no connection was possible because there is no such device.

TRANSPORT.LostDevice() —> Reference — Indication

Indication	The Device has become unreachable, that is will no longer respond to talk register 3 commands.

TRANSPORT.GetAllAttachedDevices() —> {DevTable} — Request

Command	Return a table of all of the currently attached devices. This information can then be used for a set of Open calls to connect to the devices.
Return	Return to Caller
Result	{Success}. This is a table that has a map of all of the attached ADB devices

TRANSPORT.Write(RefNum, [DataPtr]) — Request

Command	Write the chunk of data to the ADB device and register defined by RefNum. Currently there are only 8 bytes that may be sent at a time. This may be changed for greater efficiency in the future.
Return	Return to Caller
Result	{Success}.

TRANSPORT.DataAvailable(RefNum) — Indication

Indication	Data is available from the RefNum channel.

TRANSPORT.Read(RefNum, [DataPtr]) — Request

Command	Read the chunk of data from the ADB device and register defined by RefNum. Currently there are only 8 bytes that may be moved at a time. This may be changed for greater efficiency in the future.

Return	Return to Caller
Result	{Success}.

TRANSPORT.SetAutoPolling(Rate,Device Map) **Request**

Command	Set the default rate that the autopolling will run at. Also define the devices that will be autopolled in the device map. Note that the GI chip cannot change the autopolling rate.
Return	Return to Caller
Result	{Success,Fail}. Fail if polling rate can be achieved, otherwise succeed.
Operation	Rate is from {0 to 255}. 0 means minimum (1 msec), 1 means 1 msec, and N means N msecs.

2. Transport Framing Layer

The ADB protocol is simple enough that there is no Transport framing layer. The control layer makes calls directly to the MAC layer.

Transport Timing Requirements

Dynamic Characteristics

Symbol	Parameter	Min	Max	Units	Test Conditions
tAP	time between autopolls	--	11	msecs	worst case about 100 keystrokes/ second
tSRQ	time between successive SRQ polls	500	--	us	

Device Characteristics Layer in Detail

1. Device Characteristics Control Layer (Keyboards)

All of the previous layers have focused on delivery of data and information between one node and another. Within a device driver, there is usually a logical representation of the actual entity at the other end. Disk drives are assumed to have different characteristics than scanners; modems are different than MIDI synthesizers. These characteristics depend on the device, not on the operating system. Usually one transport stream is normally mapped to a single device. In the case of the Norsi keyboard, two different streams are attached to a single device. Also, it is possible to have more than one device attached to a single stream or group of streams. An example might be sound modifier keys on a keyboard that are part of the keyboard ADB device but are passed to their own device driver.

For keyboards in the current system, incoming data is taken out of the stream, and translated into a virtual keycode using a 'KMAP' resource in the system file. The state of the modifier keys is checked, and then an event record is created. This event record is then dumped into the event queue using the _POSTEVENT trap, where applications pick it up.

Device Characteristics Control Layer Calls

DC.Open(RefNum) **Request**

Command	Open paths to the ADB device specified by RefNum. This call connects the device characteristics module to the transport stream.
Return	Return to Caller
Result	{Complete, Error}. Either this was successful or not

2. Device Characteristics Framing

The device characteristics level will translate the calls to it to a device dependent but medium independent set of commands. The Device characteristics framing level will convert these commands to the actual bytes required to be transmitted, and then will call the transport layer to send them.

The Apple extended keyboard is used as an example.

Device Characteristics Framing Layer Calls

DC.F.OnLeds([RefNum],{NumLock,CapsLock,ScrollLock}) **Request**

Command	turn on one or more LEDs on the keyboard
Return	Return to Caller
Result	{Success}

DC.F.OffLeds([RefNum],{NumLock,CapsLock,ScrollLock}) **Request**

Command	turn off one or more LEDs on the keyboard
Return	Return to Caller
Result	{Success}

DC.F.SetDeviceHandler([DevNum],Hndlr) **Request**

Command	Set a new device handler number in the Keyboard
Return	Return to Caller
Result	{Success}

DC.F.GetKey([DevNum]) ->KeyCode+Modifiers **Request**

Command	Get the next key code and modifier bits. If no key is available, then wait synchronously until one is available.
Return	Return to Caller
Result	{Success}

DC.F.ResetKeyMap([DevNum]) **Request**

Command	Reset the key map to the initial state.
Return	Return to Caller

Result {Success}

Device Driver in Detail

Currently the Start Manager will search the system file for ADBS resources. For each of these that if finds, it treats the resource as an ADB device driver. The ID number of the ADBS resource matches the ADB device.

The ADBS device driver interface is described in Inside Macintosh.

Acknowledgements

Special thanks to Gary Davidian, Steve Smith, Gus Andrade, Gary Rensberger, and Wayne Meretsky for their help and information.

References

More information on the ADB System of the Macintosh can be found in the following books and articles.

[1] Guide to the Macintosh Family Hardware. Addison-Wesley 1990

[2] Inside Macintosh vol 1-5. Addison-Wesley 1986

[3] Inside Macintosh vol 6. Apple Computer, Inc.

[4] Macintosh Tech note #160 : Key Mapping

[5] Packet ADBu ERS, Ray Montagne.

Appendix

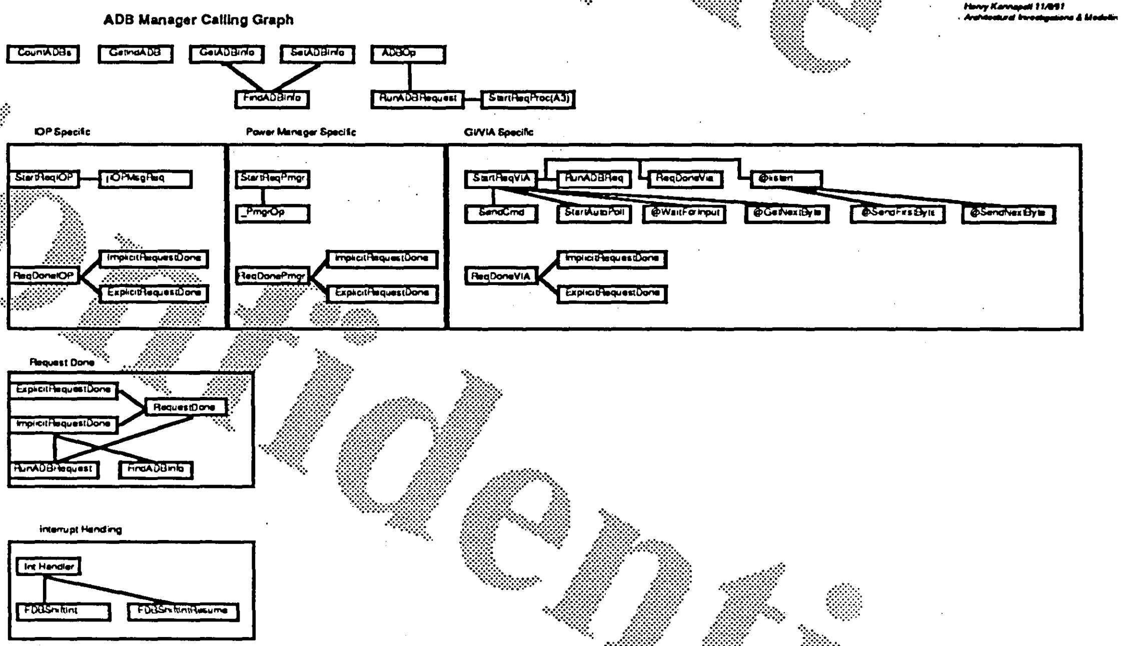
Current ADB Manager Call Graph.

A Call Graph is a method of exposing structure in software. Each block in the diagram represents a body of code.

Each horizontal link is a jump at the same level, that is it jumps without calling any subroutines.

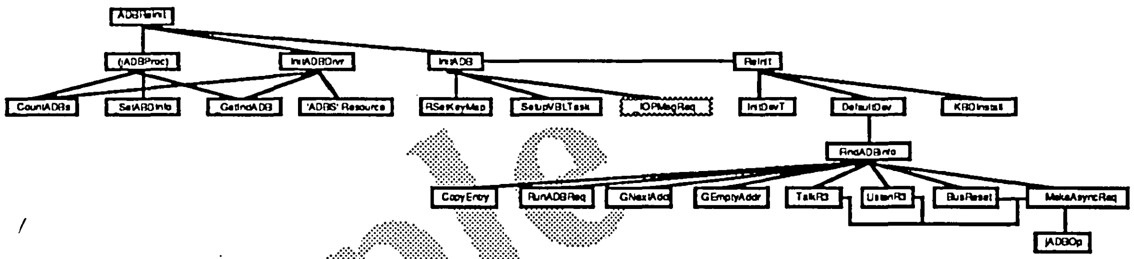
Each vertical link is a subroutine or a trap that saves the current state and will return upon completion. The names of the blocks correspond to either the module names of the code or the label of a section of code. A label is required to identify a section when a jump takes place.

Note that for start request there is a different mechanism for each I/O implementation. The power manager uses subroutine calls, the IOPs use jumps, and the VIA/GI type uses in line code with subroutines for supporting pieces of code.



ADB Manager Calling Graph

ADB Manager Initialization Routines



Chapter 3

Floppy

version 1.0

Scott Sarnikowski

About This Chapter

This chapter describes the architecture of the floppy driver.

Glossary

GCR — Group Code Recording. Encoding format for 400K and 800K media. This technique encodes 3 bytes of data into 4 to limit the run length of the data written to disk.

MFM — Multi-Frequency Modulation. Encoding format for 720K, 1.4M, and 2.8M media. This technique encodes data "1's" as transitions and inserts transitions between adjacent "0's" to limit the run length of data written to disk. Unlike GCR, MFM has no restrictions on data patterns.

Mark Byte — Special byte used in MFM to provide synchronization for data recovery. Mark bytes violate the MFM coding standard by dropping a clock.

NRZI encoding — Non Return to Zero Invert on 1's. Encoding technique in which data both "0's" and "1's" cause transitions. MFM is an example of an NRZI code.

Nibblizing — Process used in the GCR encoding technique. The nibblizing process encodes 3 bytes of data into 4 bytes to guarantee no more than 2 bit times between transitions.

Denibblizing — Data recovery for GCR data. The denibblizing process decodes 4 bytes of data into 3 bytes.

Perpendicular Recording — Recording technology used on 2.8MByte media.

Sector — Smallest unit of data storage. For Apple floppy disks, a sector contains 512 bytes.

About Floppy

The Mac supports two floppy standards: Multi-Frequency Modulation (MFM) and Group Code Recording (GCR.) Apple has used Group Code Recording (GCR) since the time of the Apple II. GCR gets its name from the fact that the encoding process works with a group of bytes rather than a single byte. To improve the reliability of data recovery, the GCR encoding process encodes data so that there are no more than 2 bit times between transitions in the serial bit stream. The data transfer rate for GCR is 489.6Kbits/sec.

The second floppy format is MFM. MFM has been an "industry standard" since the introduction of the IBM PC in 1981. Apple did not support MFM until the introduction of the Mac IIx and SE. Unlike the GCR technique, there are no limitations on data patterns. The MFM technique encodes all "1's" as transitions and inserts transitions between adjacent "0's" in the serial bit stream. MFM implementations in Mac systems transfer data at 500Kbits/second (720K and 1.4M) and 1MBits/second (2.8M). The data rates are the same for industry standard drives, except that the transfer rate of 720K is 250KBits/second.

Besides the non-standard GCR data encoding technique, all Apple floppy drives have a proprietary drive interface. Unlike the standard 36 pin interface, the Apple floppy drive interface is 20 pins and has only one output from the drive, Read Data. All status and read data are mux'ed on the read data line. This has considerable benefits with connector size, but significantly complicates the interface over industry standard drives. The industry standard interface has dedicated lines for status.

All floppy implementations in Macintosh systems to date have either used the IWM or SWIM design. The most notable characteristic of both of these designs is that the hardware only implements the serial/parallel and parallel/serial conversions. With the low level hardware support of the IWM and SWIM, System Software is responsible for all dynamic control. A new part, currently under development (NEC 72070), considerably raises the level of hardware support for the floppy interface, eliminating the severe real-time constraints of the IWM/SWIM interface.

Arioso Model of Floppy

The Arioso model of Floppy contains six layers: Device Driver, Device Characteristics, Transport, MAC, MIOC, and Physical. Figure 3-1 shows the Arioso layering for Floppy.

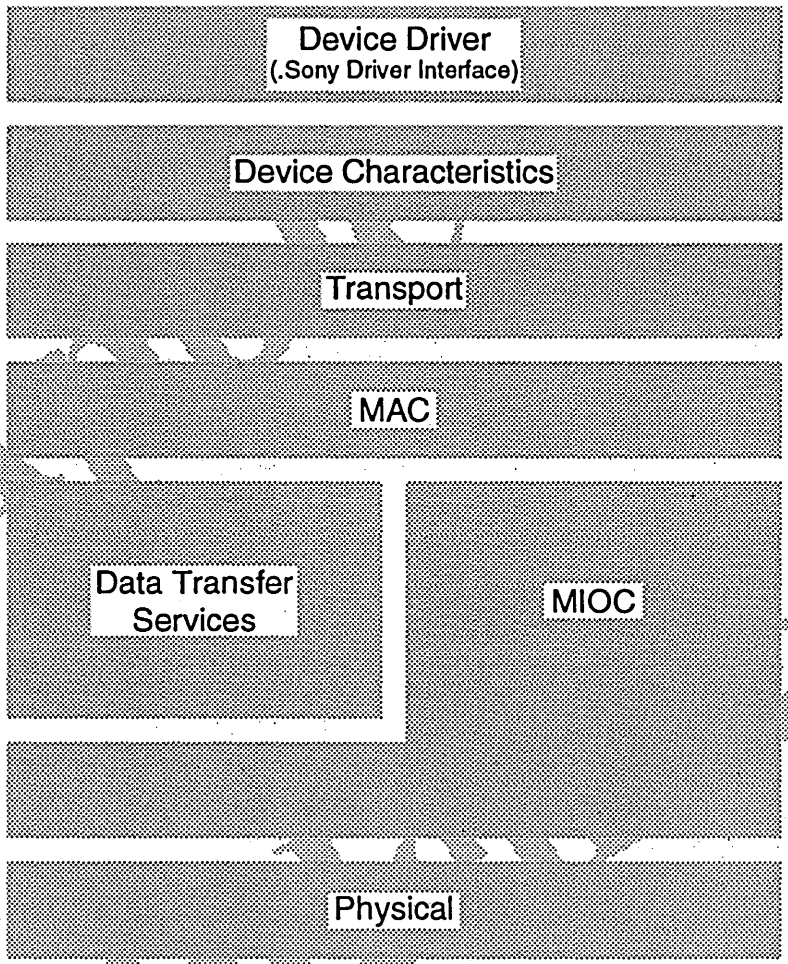


Figure 3-1. Arioso Partitioning of Floppy

Device Driver Layer

The device driver layer is responsible for taking external read, write and control requests and mapping them to device specific equivalents. One of the functions implemented by the device driver layer is to examine the characteristics of the device and map the generic data requests into the device specific data request. For floppy, this means that the device driver converts from a byte offset reference to a head, track, and sector reference on the actual media. The external interface to the device driver layer is specified in the .Sony Floppy Disk/HD-20 Driver External Software Specification (Christensen, 1990).

Device Characteristics Layer

The device characteristics layer is responsible for maintaining device specific information necessary to map the logical data requests for the external interface to the device specific references. This layer maintains information such as media format, density, sector length, etc.

Transport Layer

The transport layer is responsible for mapping requests to the appropriate physical device. In the levels above the transport layer, all references to a device are with logical device numbers. The transport layer maintains the tables necessary to map the logical references to physical references needed to select the proper devices.

MAC Layer Description

The MAC layer s responsible for implementing the low level functions of the Floppy driver. The MAC layer implements commands such as read a sector. At the MAC layer, the read a sector command consists of calls to the MIOC layer to find the sector, read it, and check for errors. In general, the MAC maintains all bi-directional communications with the device.

MIOC Layer Description

The MIOC layer is responsible for implementing the atomic commands necessary to implement the commands of the MAC. In this way, the MIOC is responsible for coordinating the hardware interface for all unidirectional operations. The MIOC has no knowledge of the coordination needed between the commands that it implements. The MIOC is also responsible for error detection, but is not responsible for understanding what to do with the information. A typical command at the MIOC layer is search for header. In this command the MIOC will read the data from the disk until it locates the specified header. The MIOC then alerts the MAC. This call can either be a part of a read or write command, but the MIOC doesn't know the difference.

Physical Layer Description

The physical layer is responsible for the interface with the device. This layer implements all data recovery and device control. Arioso defines the physical layer general enough to permit the interface from the MIOC to be common to many different physical interfaces.

Device Characteristics Layer in Detail

The device characteristics layer is responsible for maintaining the information necessary to map from the device independent byte offsets of the device driver calls to the absolute locations on the device. To achieve this, the device characteristics layer has knowledge of the data format (GCR or MFM), sector size, and disk capacity. With this information the device characteristics layers can convert the byte offsets to the physical head, cylinder, and sector. This layer uses only logical address.

The disk.inserted event starts the only function of the device characteristic layer. When the device characteristics layer senses the disk_inserted event, it calls the Transport layer to determine the device dependent information.

Transport Layer in Detail

The sole responsibility of the Transport layer is to map from the logical references to physical references.

MAC Layer in Detail

The MAC layer is responsible for all bi-directional traffic between higher levels and the device. The Arioso model defines Bi-directional traffic as communications requiring coordination between reading and writing operations to the device. The three operations that the MAC implements are format a track, read a sector, and write a sector. All discussions below will use MFM for convenience, and can be extended to not only cover GCR but any other encoding standard as well

Format a track

The flow diagram for the Format a Track command begins in figure 3-2.

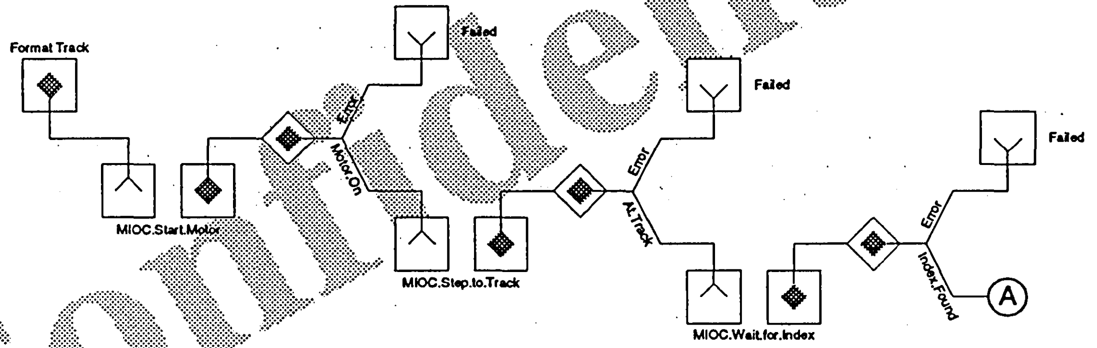


Figure 3-2. Format a Track

The first step in formatting a track is to start the motor. The command starts the motor with a MIOC.Start.Motor call. After the command returns, the execution flow can block until the MIOC returns an indication. If the motor failed to start, the command records an error and terminates. The command then positions the head to the correct track with a MIOC.Step.to.Track. Implicit to the call is the knowledge of where the head is so that at this level there is no need to keep track of the head location. After the call to the MIOC, the command can block waiting on the outcome of the command. If the step fails, the command terminates with an error indication. Once the head is properly positioned, the command then issues the MIOC.Wait.for.Index call to find the beginning of the track. After the call, the command can block until an the MIOC issues an index.found indication. If the MIOC failed to find the index, the command fails and terminates with an error indication. If the index is found the command can continue with formating the track (Figure 3-3).

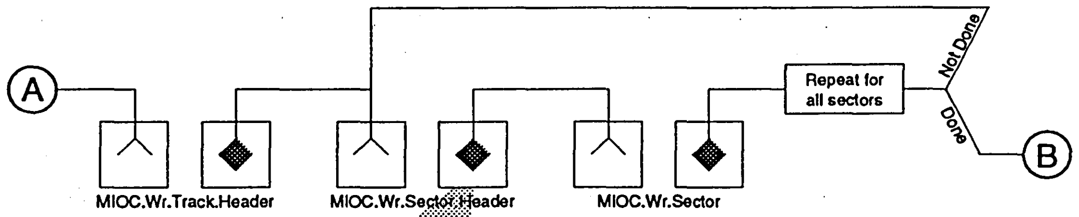


Figure 3-3. Format a Track (cont.)

The first step in writing the track format information is to write out the index address information. This operation must begin with very little delay so that it coincides with the index pulse. The command writes the index address information with the MIOC.Wr.Track.Header. Following the index address information, the command formats the individual sectors with two calls: MIOC.Wr.Sector.Header and MIOC.Wr.Sector. The MIOC.Wr.Sector.Header is responsible for writing the address mark, address header, and gaps. The MIOC.Wr.Sector is responsible for formatting the data in the sector as well as writing the data mark and the gaps. During the format operation, the command calls the MIOC.Wr.Sector with the format option, writing all "0's" in the sector data. Figure 3-4 shows the final steps of the format operation

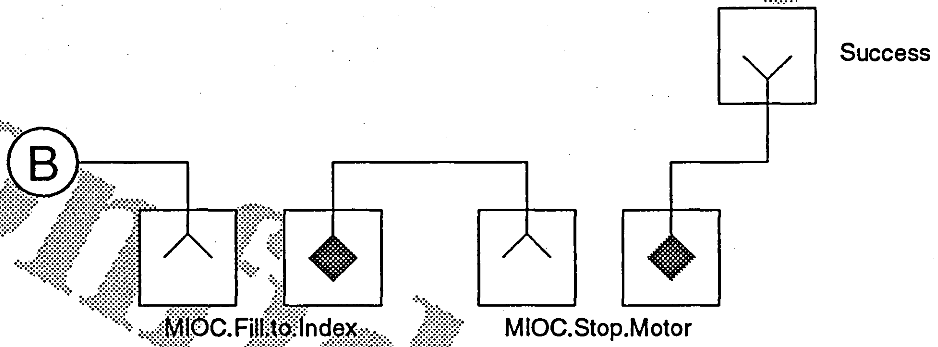


Figure 3-4. Format a Track (cont.)

To finish the format operation, the command fills the rest of the track with the MIOC.Fill.to.Index call. After the command has filled the remainder of the trac, it stops the motor with a MIOC.Stop.Motor call.

Read a Sector

The flow diagram for the Read a Sector command begins in Figure 3-5.

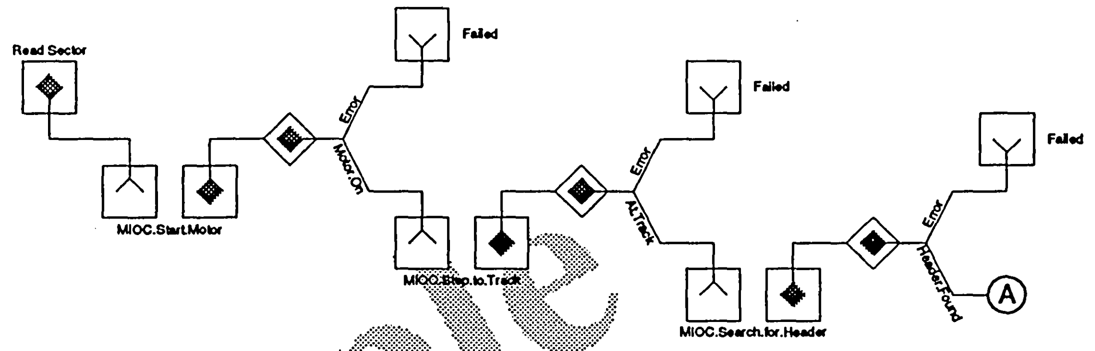


Figure 3-5: Read a Sector

As with Format a Track, the first step in the Read a Sector is to turn the motor on with a MIOC.Start.Motor command. Once the motor has successfully started, the command then positions the head with the MIOC.Step.to.Track command. After the command positions the head, it then locates the desired sector by issuing the MIOC.Search.for.Header. The flow can then block waiting on a header.found indication from the MIOC. If the MIOC.Search.for.Header command fails to find the requested sector, the command terminates with an error indication. After the command finds the specified sector, it moves on to read the sector (Figure 3-6).

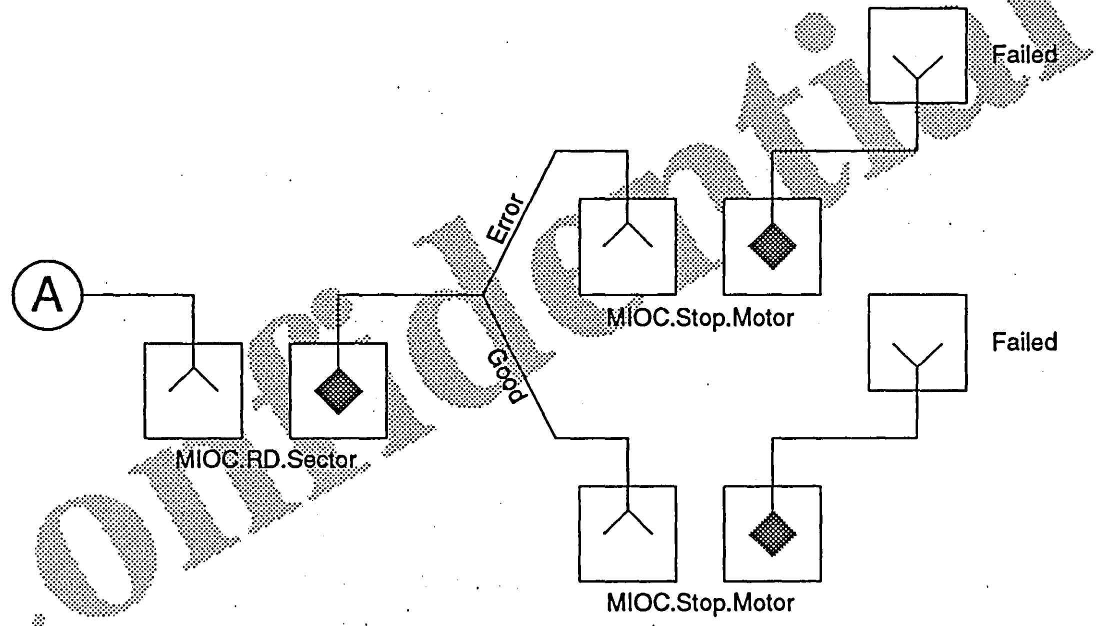


Figure 3-6. Read a Sector (cont.)

The MIOC.Rd.Sector call is responsible for reading the data, moving it into memory, and detecting any errors. If the call read the data without an error, it terminates with a good indication. If the call found an error, it returns with an error indication. Prior to the return, the command turns off the motor.

Write a Sector

The flow diagram for writing a sector begins in figure 3-7.

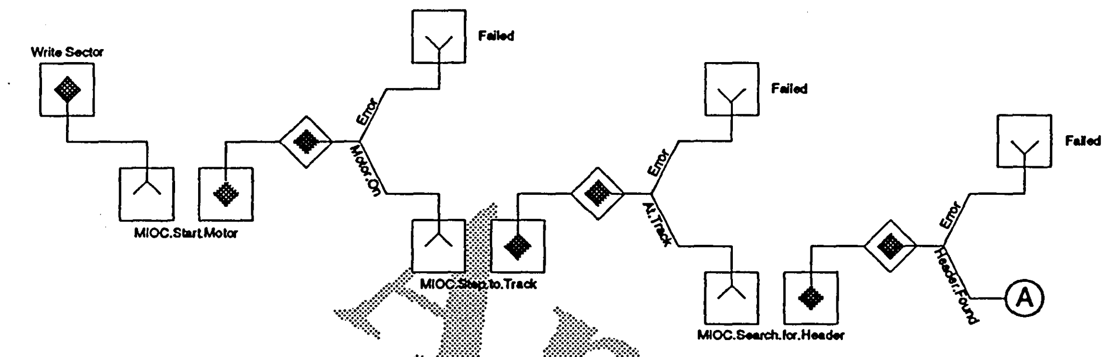


Figure 3-7. Write a Sector

The first portion of writing a sector is identical to reading a sector. In this part, the command starts the motor, positions the head, and then locates the sector. Any failure will cause the command to terminate with an error indication. Once the command finds the specified sector, it then writes the sector data (Figure 3-8)

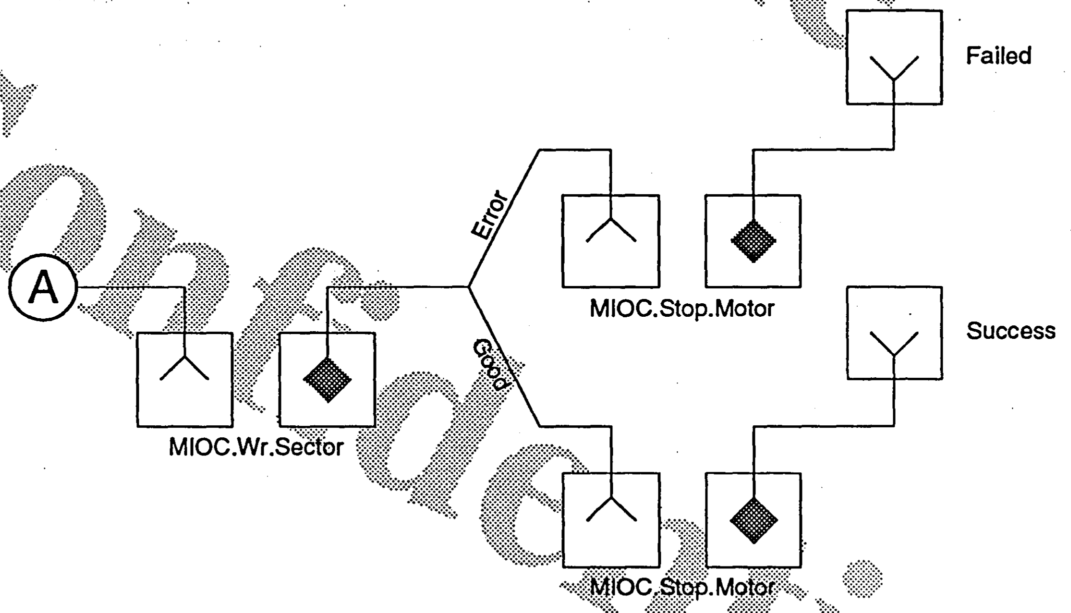


Figure 3-8. Write a Sector (cont.)

The command writes the sector data with the MIOC.Wr.Sector call. This MIOC call is responsible for calculating the checksum, checking for errors, and transferring all data. If the call writes the data without an error, the command terminates with a success indication. If the call finds an error, the command terminates with an error indication. Before completion, the command turns off the motor.

MAC Layer Calls

`MAC.Format.Track(track, head, no. of sectors)` Request

Command	Format the specified track with the specified number of sectors. The headers for each sector are supplied by the Data Transfer Services during the execution of the command.
Return	Return to caller.
Result	{Success, Failed}

`MAC.Wr.Sector(track, head, sector)` Request

Command	Write sector data on the specified track and sector.
Return	Return to caller.
Result	{Success, Failed}

`MAC.Rd.Sector(track, head, sector)` Request

Command	Read sector data on the specified track and sector.
Return	Return to caller.
Result	{Success, Failed}

MAC Timing Requirements

The most critical timing requirements of the MAC layer are the times between adjacent write and read commands. Table 1. shows these times for the MAC.

Table 1. Timing for MAC Dynamic Characteristics

Symbol	Parameter	Min	Max	Units	Test Conditions
t_{Hdr}	Header found to MIOC.Rd.Sector or MIOC.Wr.Sector	--	16	µs/byte	1.4MByte
			8	µs/byte	2.8MByte
t_{fmt}	Minimum between end of command to execution of next during format a track	--	16	µs/byte	1.4MByte
			8	µs/byte	2.8MByte

MIOC Layer in Detail

Fill to Index

Figure 3-9 shows the flow diagram for the Fill to Index command.

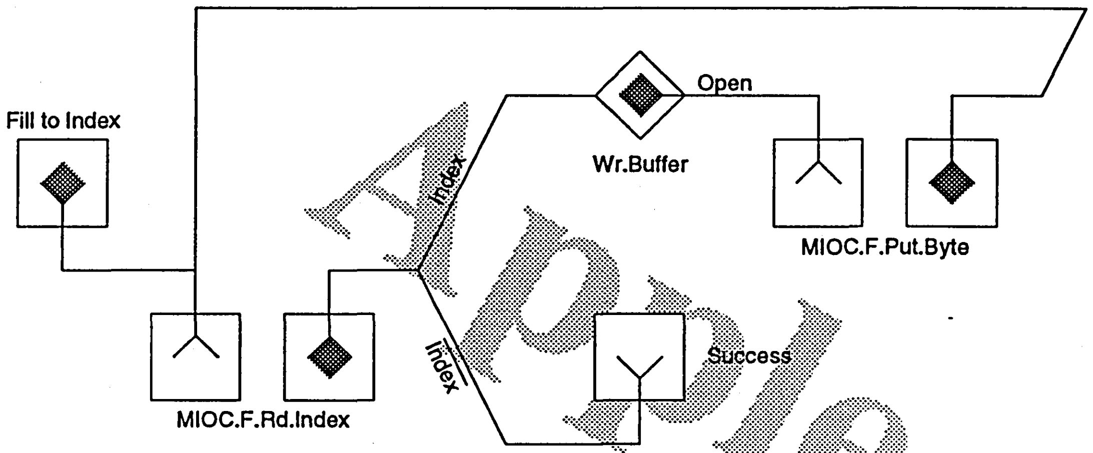


Figure 3-9. Fill to Index Command

The MAC uses the Fill to Index command to finish the format a track command. The Fill.to.Index command finishes a format operation by filling the remainder of the track with gap bytes. The command first checks for the index pulse by calling the MIOC framing layer with the MIOC.F.Rd.Index command. If the call does not detect the index pulse, the command writes out the gap byte. This continues until the command detects the index pulse. At this point the command terminates, and returns control to the MAC.

Read Sector

The flow diagram for the Read Sector command begins in Figure 3-10 below.

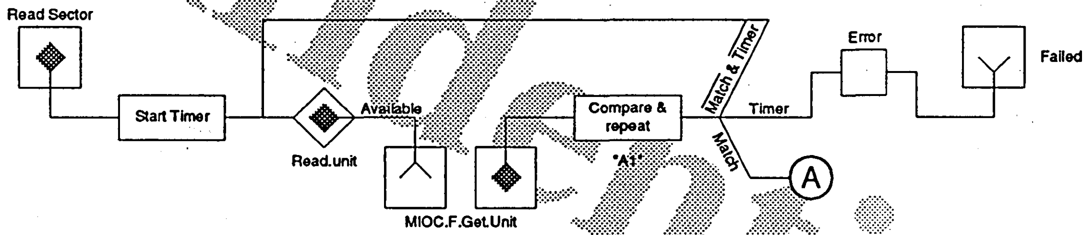


Figure 3-10. Read Sector Command

The first action of the Read Sector command is to start a deadman timer that is used for error checking while looking for the data mark. When reading the bytes, the command blocks until an indication that a byte is available is asserted by the physical layer. When the byte is available, the command uses the MIOC.F.Get.Unit call of the MIOC framing layer to read and then compare the byte. If the first data mark "A1" is found before the timer expires the command continues on to read the rest of the mark. If the timer expires before the mark is found, the command terminates with an error indication.

Following the first data mark, the command continues to read the next 3 available bytes (Figure 3-11). If they do not match the remainder of the data mark ("A1" "A1" "FB"), then the command terminates with an error. If the data mark is correct, the command continues on to read the data in the sector.

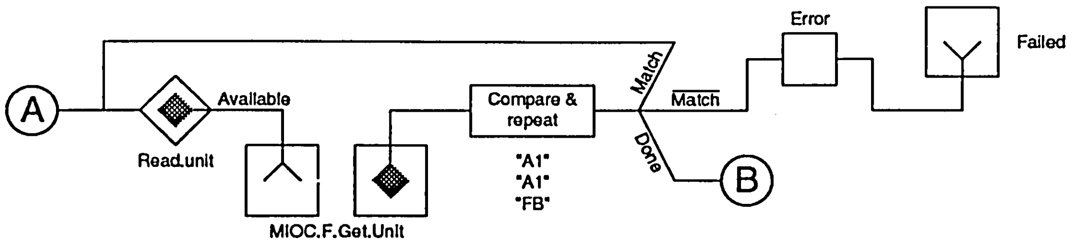


Figure 3-11. Read a Sector Command (cont.)

The process of reading the sector data requires that the command read data from the physical layer and coordinate with the Data Transfer services layer (Figure 3-12).

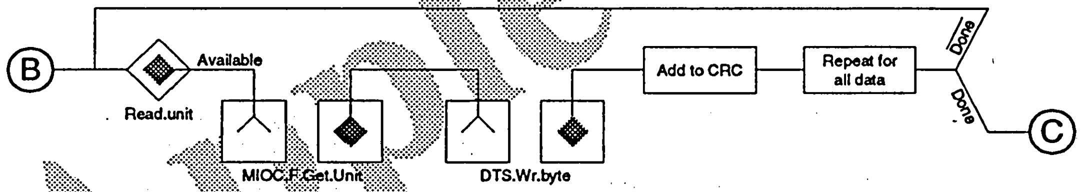


Figure 3-12. Read a Sector Command (cont.)

The command reads the data by first blocking until a byte is available. When a byte is available the command reads the byte with the MIOC.F.Get.Unit command. The command then calls the DTS layer with the DTS.Wr.Byte command. The action of the this command is to place the byte into memory and alleviates the MIOC of the details of the memory systems or DMA. After the byte is read, the byte is added to the CRC and the process is repeated for every byte of the sector.

When the final byte is read, the command moves on to read the CRC and check for errors (Figure 3-13).

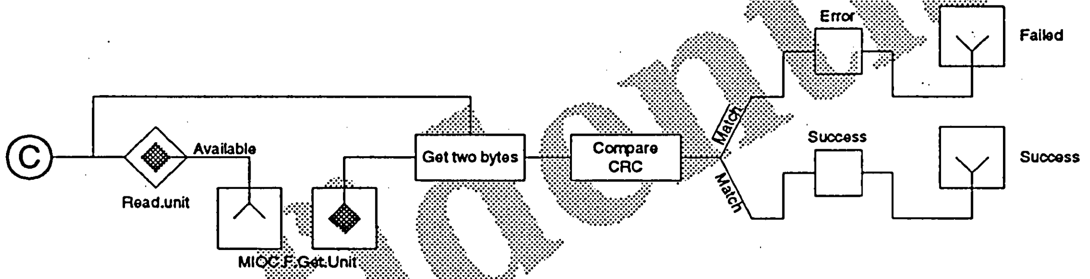


Figure 3-13. Read a Sector Command (cont.)

The command reads the two CRC bytes and then compares them with the calculated CRC. If they match, the command is terminated with a good indication. If the CRC does not compare, the command terminates with an error indication.

Start Motor

The Start Motor command is shown in Figure 3-14 below.

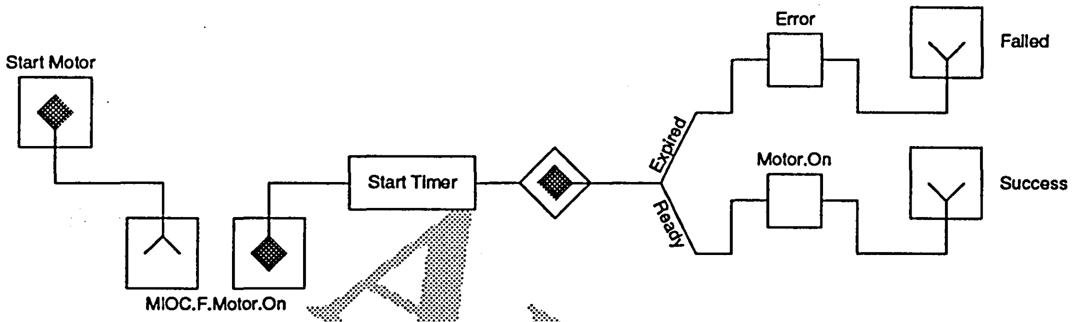


Figure 3-14. Start Motor Command

The Start Motor command is used to start up the drive motor. The first step is to issue the MIOC.F.Motor.On command. After this call, the command then starts a deadman timer and blocks. If the ready indication from the physical layer is not received before the timer expires, the command terminates with a failure indication. If the ready indication is asserted, the command terminates with a complete indication.

Wait for Index

The Wait for Index command is shown in Figure 3-15 below.

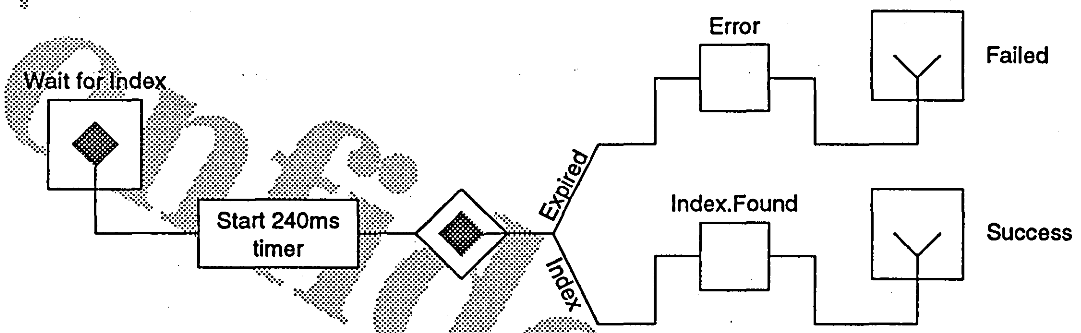


Figure 3-15. Wait for Index

The Wait for Index command is used to properly position the head to begin a format command. The first step of the command is to start a 240ms timer. The command then blocks until either the command expires or the index pulse is encountered. The command is then terminated with the appropriate indication.

Search for Header

The Search for Header command begins in figure 3-16 below.

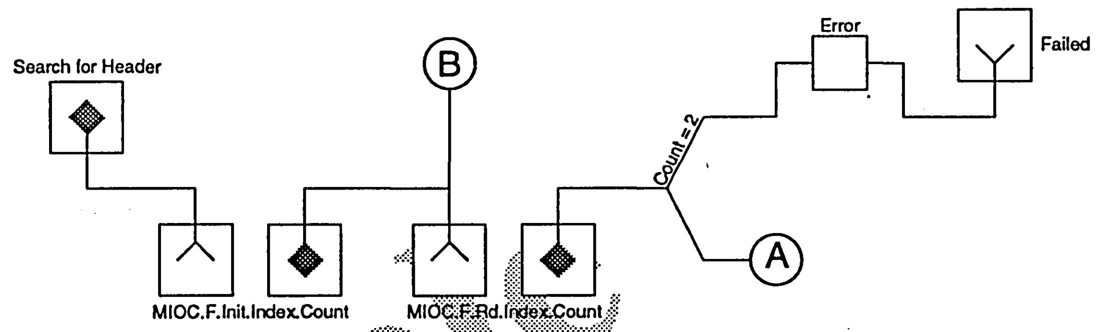


Figure 3-16. Search for Header Command

The first step in the Search for Header command is to initialize the index counter in the physical layer. This counter is set to tell if the search operation has gone over two revolutions. The command then enters a loop in which the index count is checked each time an index mark is not found. At this point, if the number of index counts is 2 or greater the command terminates with an error indication, otherwise the command continues on as shown in Figure 3-17.

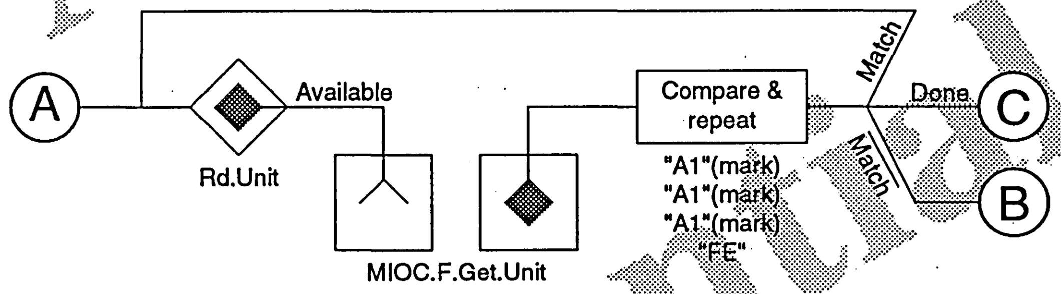


Figure 3-17. Search for Header Command (cont.)

When reading a byte, the command blocks until a byte is available to be read from the physical layer. When a byte is available, the command reads the byte and compares the byte against the address mark. If the byte does not match a byte in the address mark, the command returns to the beginning to continue the search process. If the command does find a valid match for all the bytes in the mark, the command continues on to read the address header (Figure 3-18).

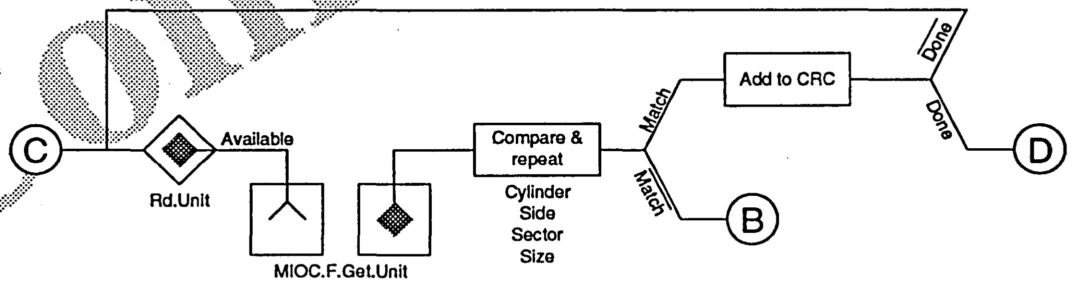


Figure 3-18. Search for Header Command (cont.)

When reading the address header, each of the 4 bytes are compared against those passed in the call. If any of the bytes do not match, the command returns to the beginning to continue the search. If all the bytes match the command continues on to check the CRC (Figure 3-19).

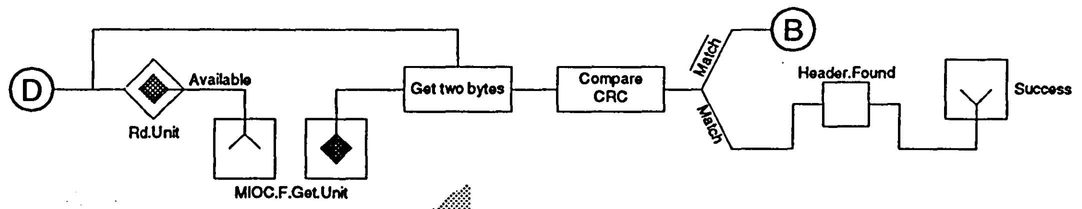


Figure 3-19. Search for Header Command (cont.)

After the command reads the two byte CRC, it is compared with the computed CRC. If the two do not match, the command returns to the beginning to continue the search. If the CRC is good, the command terminates with a success indication.

Step to Track

The Step to Track command is used to position the head over the required track. The flow diagram is shown in Figure 3-20 below.

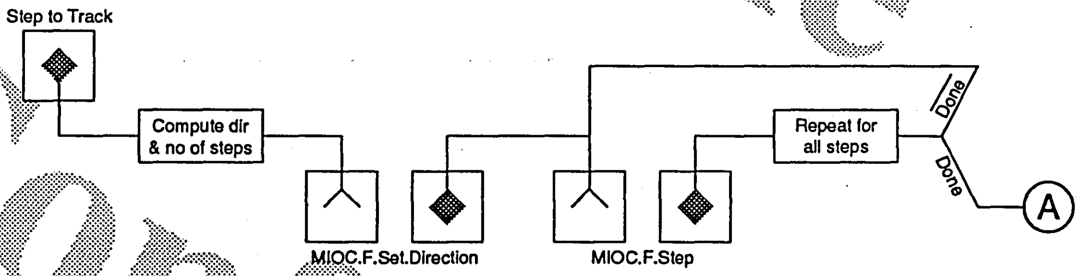


Figure 3-20. Step to Track

The first step in the command is to compute the number of the steps and direction to position the head. Once this is done, the command sets the direction with the MIOC.F.Set Direction command. When that is complete, the command then issues the all the steps with the MIOC.F.Step command. After all the steps have been executed, the command then checks the ready status of the drive with the MIOC.F.Rd.Ready. (Figure 3-21).

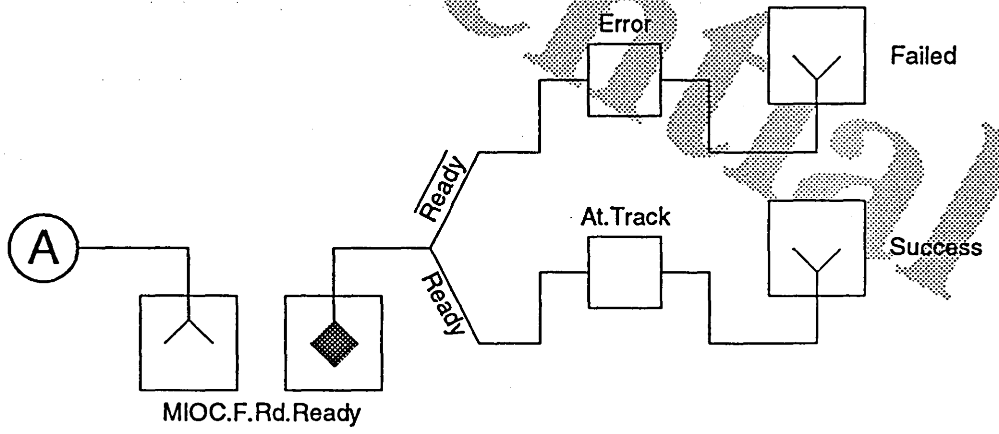


Figure 3-21. Step to Track (cont.)

If the ready status of the drive is not asserted the command terminates with an error indication. If the ready status is asserted, the command terminates with a success indication.

Write Sector

The Write Sector command is used to write the contents of the sector. When the command is entered, it is assumed that the head is positioned after the appropriate address header. The flow diagram of the Write Sector command is shown in Figure 3-22.

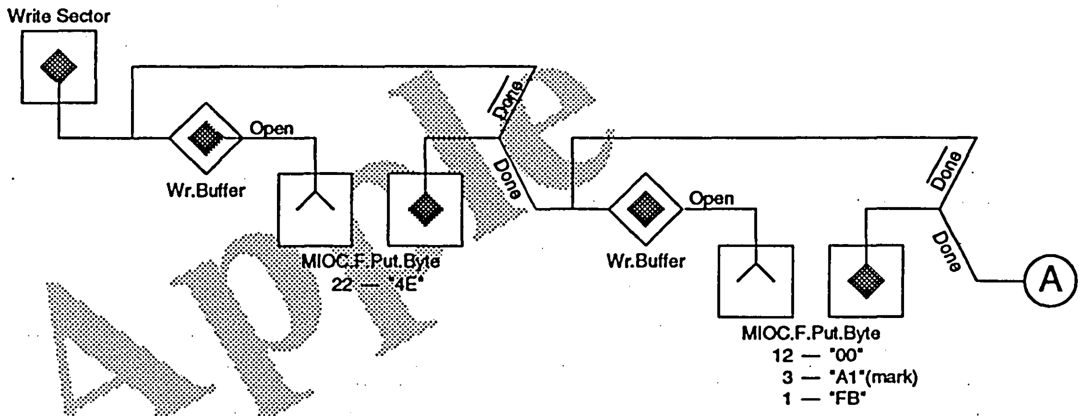


Figure 3-22. Write Sector Command

The first step in writing a sector is to rewrite the intra-sector gap. For 1.4MByte MFM, the intra-sector gap is composed of 22,"4E" data bytes. The execution of the command blocks waiting for access to the physical layer. When the physical layer can accept a new byte, the command writes out the byte with a MIOC.F.Put.Byte command. After all the bytes have been written, the command writes out the address mark. Following the address mark the command continues on to write out the sector data (Figure 3-23).

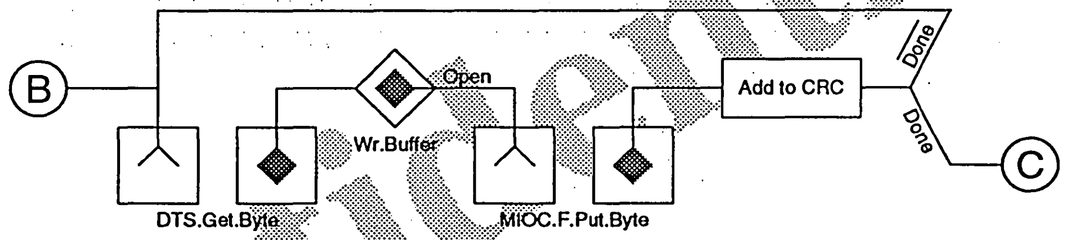


Figure 3-23. Write Sector Command (cont.)

To write out the sector data the MIOC coordinates the action of the Physical and Data Transfer layers. Each byte to be written is first read from the Data Transfer Services layer with DTS.Get.Byte command. Once the byte has been passed, the command blocks waiting for the indication that room available in the physical layer. When room is available, the command writes out the byte with the MIOC.F.Wr.Byte call. After each byte is written, the byte is accumulated in the CRC.

When all the bytes have been transferred and the CRC accumulated, the command finishes by writing out the CRC and the inter-sector gap (Figure 3-24).

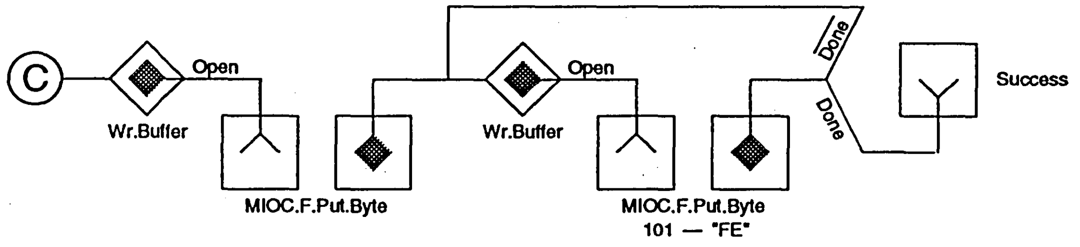


Figure 3-24. Write Sector Command (cont.)

Finally, the command terminates with a success indication.

Write Sector Header

The Write Sector Header command is responsible for formatting and writing out the sector address. The flow diagram for the command begins in Figure 3-25.

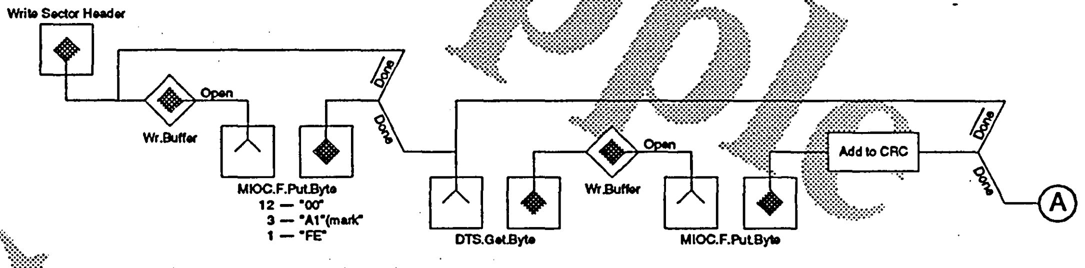


Figure 3-25. Write Sector Header

The Write Sector Header command begins by writing out the address mark. Following the address mark, the command call the DTS to get the address header bytes. After the byte has been read from the DTS by the DTS.Get.Byte, the command blocks waiting for access to the Physical Layer. When the Physical Layer has room, the byte is written with the MIOC.F.Put.Byte call. The byte is also added to the CRC. After all the bytes in the address header have been written, the CRC is written and the command terminates (Figure 3-26).

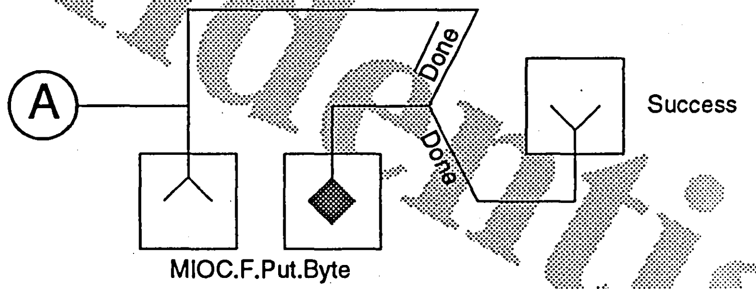


Figure 3-26. Write Sector Header

Write Track Header

The Write Track Header command is used to format and write the Track Header that coincides with the index pulse. The flow diagram of this command is shown in Figure 3-27.

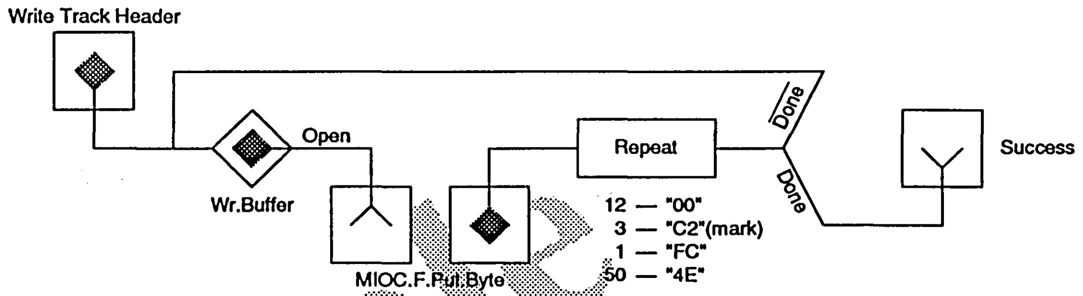


Figure 3-27. Write Track Header Command

Prior to writing out the track header information, the command blocks until a room is available in the Physical layer. When room is available, the command writes out the byte with the MIOC.F.Put.Byte call. This is repeated until the entire header is written. The command then terminates with a success indication.

MIOC Layer Calls

MIOC.Fill.to.Index — Request

Command	Fill from current head position to index pulse with byte pattern
Return	Return to caller.
Result	{Success, Failed}

MIOC.Rd.Sector — Request

Command	Read data from identified sec tor. Data is read and deposited in memory.
Return	Return to caller.
Result	{Success, Failed}

MIOC.Start.Motor — Request

Command	Start up the motor.
Return	Return to caller.
Result	{Success, Failed}

MIOC.Wait.For.Index — Request

Command	Wait for the index pulse. This command is used to synchronize the head with the start of the track.
Return	Return to caller.
Result	{Success, Failed}

MIOC.Search.for.Header — Request

Command	Find the specified header.
Return	Return to caller.
Result	{Success, Failed}

MIOC.Step.to.Track — Request

Command	Step the head to the specified track.
Return	Return to caller.
Result	{Success, Failed}

MIOC.Wr.Sector — Request

Command	Write sector data. The data is transferred from memory.
Return	Return to caller.
Result	{Success, Failed}

MIOC.Wr.Sector.Header — Request

Command	Write the sector address header.
Return	Return to caller.
Result	{Success, Failed}

MIOC.Wr.Track.Header — Request

Command	Write the track header.
Return	Return to caller.
Result	{Success, Failed}

MIOC Timing Requirements

The most important timing requirements of the MIOC layer are the times between indications and the calls to write or read data from the Physical layer. Table 2. shows these times for the MIOC.

Table 2. Timing for MIOC Dynamic Characteristics

Symbol	Parameter	Min	Max	Units	Test Conditions
t_{wr}	Wr.Buffer.Open indication to MIOC.F.Put.Byte	--	16	µs	1.4MByte
			8	µs	2.8MByte
trd	Rd.Unit.Available indication to MIOC.F.Get.Unit	--	16	µs	1.4MByte
			8	µs	2.8MByte
t_{DTS}	DTS.Get.Byte completion to MIOC.F.Put.Byte	--	16	µs	1.4MByte
			8	µs	2.8MByte

MIOC Framing Layer

The MIOC framing layer is responsible for generating the physical layer dependent actions required to implement simple functions such as get unit and put unit.

MIOC Framing Layer Calls

`MIOC.F.Rd.Index` Request

Command	Read the status of the Index pulse.
Return	Return to caller.
Result	{On, Off}

`MIOC.F.Put.Unit` Request

Command	Write unit to the Physical layer.
Return	Return to caller.
Result	{Success, Failed}

`MIOC.F.Get.Unit` Request

Command	Read unit from the Physical layer.
Return	Return to caller.
Result	{Success, Failed}

`MIOC.F.Motor.On` Request

Command	Turn on the device motor.
Return	Return to caller.
Result	{Success, Failed}

`MIOC.F.Get.Index.Count` Request

Command	Read the index counter.
Return	Return to caller.
Result	{Index count}

`MIOC.F.Init.Index.Count` Request

Command	Initialize the index counter.
Return	Return to caller.
Result	{Success, Failed}

`MIOC.F.Set.Dir` Request

Command	Set the head step direction.
Return	Return to caller.
Result	{Success, Failed}

`MIOC.F.Step` Request

Command	Issue a single step pulse to the device.
Return	Return to caller.
Result	{Success, Failed}

`MIOC.F.Rd.Ready` — `Request`

Command	Ready the ready status from the device.
Return	Return to caller.
Result	{Ready}

`Error` — `Indication`

Indication	Indication that the command has failed.

`Success` — `Indication`

Indication	Indication that the command has succeeded.

`Motor.On` — `Indication`

Indication	Indication that the motor has been successfully turned on.

`Index.Found` — `Indication`

Indication	Indication that the index pulse has been asserted within the specified time limit.

`Header.Found` — `Indication`

Indication	Indication that the requested header has been found.

`At.Track` — `Indication`

Indication	Indication that the head has been positioned over the requested track.

Physical Layer in Detail

The Physical layer consists of the minimal set of registers and circuits that could be addressed by a software implementation. Because of the different coding standards of MFM and GCR (and potentially more), and different drive interfaces, the Physical is thought of as being unique to the interface. This is done conceptually, but is not intended to preclude the development of multi-purpose implementations. The generic block diagram for the Physical layer is shown in Figure 3-28.

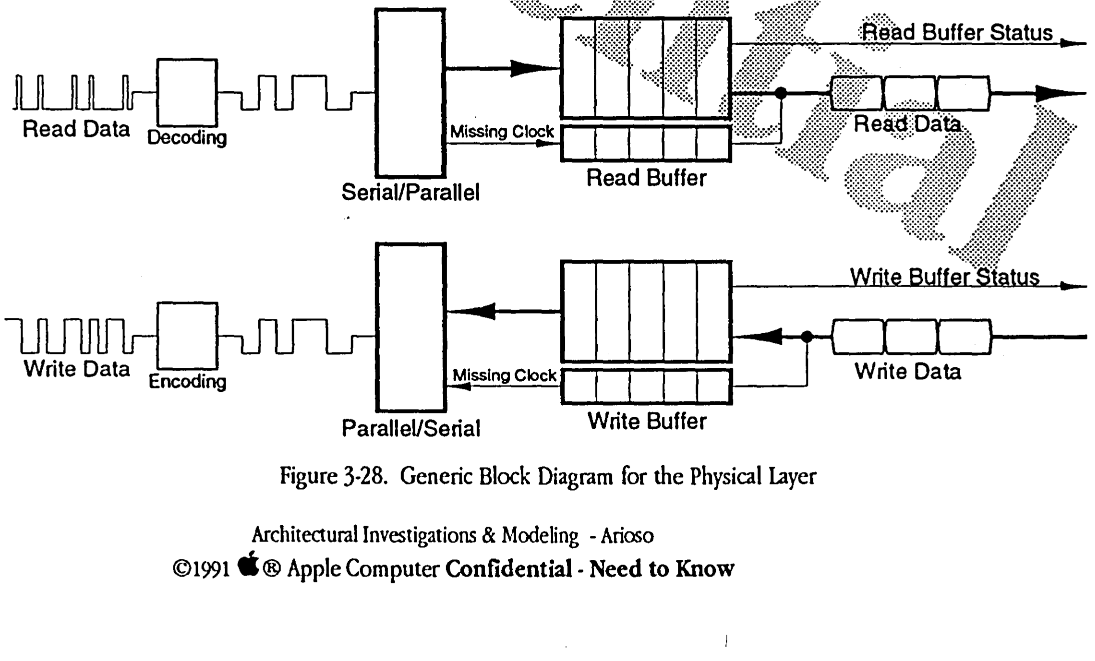


Figure 3-28. Generic Block Diagram for the Physical Layer

The main components of the generic implementation of the Physical layers are: decoding, encoding, Serial/Parallel conversion, and Parallel/Serial conversion.

The decoding block is responsible for decoding the serial bit stream prior to converting to parallel conversion. This is necessary for any NRZI code that has to recover the "0's" and "1's" differently.

The encoding block is responsible for the opposite operation. The encoding block takes the serial bit stream from the Parallel/Serial converter and encodes for standards like NRZI. This block could also be responsible for "1's" insertion if it were required by the interface.

The Serial/Parallel interface is responsible for recovering the data from the decoded serial bit stream. This block contains the circuitry necessary to separate clock from data and will detect coding violations. This is necessary for MFM, where a coding violation is used to delimit features of the format. The code violation is carried with the data as it is processed by the higher layers.

The Parallel/Serial interface is responsible for converting from parallel data from the higher layers to serial data to be sent to the encoder. One of the important features of the this block is to insert any special characters required by the coding format. In the case of MFM, this means that mark bytes are encoded with a missing clock transition.

Another aspect of the Physical layer is the translation of high level command calls to interface specific signals. The translation is dependent upon the floppy disk interface and a high level command is specified to improve portability.

Physical Layer Indications

Rd.Unit.Available — Indication

Indication	Indication that a unit of data is available from the Physical layer.

Wr.Buffer.Open — Indication

Indication	Indication that room is available in the write buffer of the Physical layer.

Index — Indication

Indication	Indication that the index pulse of the device is asserted.

Ready — Indication

Indication	Indication that the ready status of the device is asserted.

References

PAR Technical Report #4, "Effects of Real-Time Devices on Floppy Disk Performance", Scott Sarnikowski, June 4, 1990

Apple Floppy Disk Controller Specification - Version 4.0, Scott Sarnikowski, August 2, 1991

Macintosh Technical Note #272, "What Your Sony Drives for You", Rich Collyer, June 1990

SWIM Chip User's Reference - Rev 1.7, Steve Christensen, September 29, 1991

.Sony Floppy Disk/HD-20 Driver External Software Specification - Rev 2.2, Steve Christensen, January 9, 1990

SWIM2 ASIC ERS - Rev 2.0, Steve Smith, August 14, 1990

Chapter 4

LocalTalk

version 1.0

Scott Sarnikowski

About This Chapter

This chapter describes the LocalTalk layer of AppleTalk. Within the AppleTalk protocol architecture, shown in figure 4-1., LocalTalk provides one the data link layers.

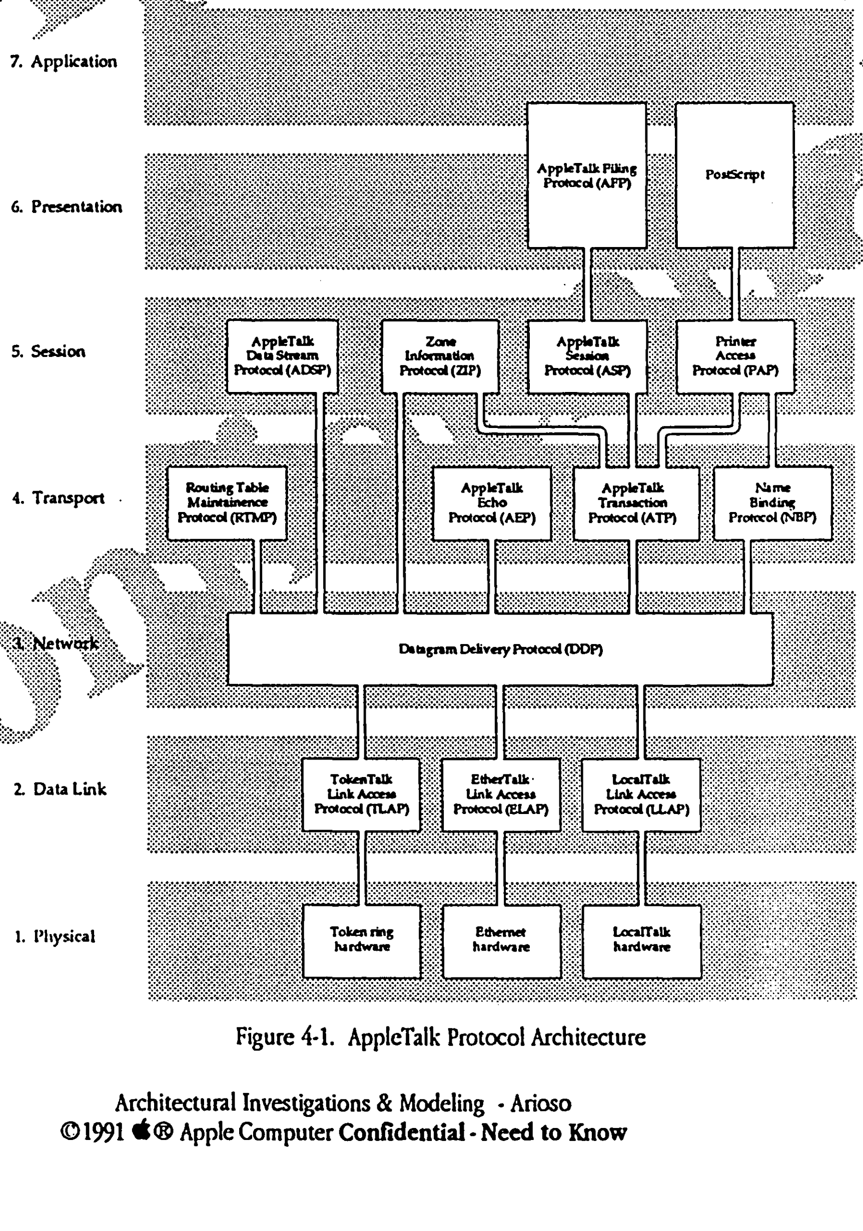

7. Application
6. Presentation
AppleTalk Filing Protocol (AFP)
PostScript
5. Session
AppleTalk Data Stream Protocol (ADSP)
Zone Information Protocol (ZIP)
AppleTalk Session Protocol (ASP)
Printer Access Protocol (PAP)
4. Transport
Routing Table Maintainence Protocol (RTMP)
AppleTalk Echo Protocol (AEP)
AppleTalk Transaction Protocol (ATP)
Name Binding Protocol (NBP)
3. Network
Datagram Delivery Protocol (DDP)
2. Data Link
TokenTalk Link Access Protocol (TLAP)
EtherTalk Link Access Protocol (ELAP)
LocalTalk Link Access Protocol (LLAP)
1. Physical
Token ring hardware
Ethernet hardware
LocalTalk hardware

Figure 4-1. AppleTalk Protocol Architecture

Glossary

Abort Bits — Part of the LocalTalk protocol that is used to "wake" up deferring senders. While in a special interrupt mode, the presence of the abort bits will cause the 85C30 to generate an interrupt when the packet on the line is complete.

Back-Off Count — Count maintained by the LAP to measure the loading on the link. Each time a collision occurs the Back-Off count is incremented.

CRC — Cyclic Redundancy Check, used to detect errors in transmission.

Defer — When a sender collides while attempting to send, the sender defers, waiting for the end of the packet. This allows other tasks to be executed while the current packet completes.

Dialog — Exchange of two or more packets that are interlocked over the link. All LocalTalk transactions occur in dialogs.

Flag — Standard SDLC transmission flag used to frame the data in the packet.

Interdialog Gap — Time between adjacent dialogs on the link. The length of the gap is 400µs to a random time dependent upon the loading of the network.

Inter-frame Gap — Time between packets within a LocalTalk dialogue, always 200µs.

Link — The physical connection between nodes on a LocalTalk network.

LLAP — LocalTalk Link Access Protocol. The LLAP is the lowest level in the AppleTalk protocol architecture.

PIC — Programmable Interface Controller. A hardware implementation of LocalTalk that uses a dedicated 6502 processor and shared memory to implement the LLAP.

Single-Ended Transmission — Packet transmission that is not interlock or check for successful completion.

SCC (85C30) — Serial Communications Controller, used in all LocalTalk implementation to provide the serial/parallel conversion.

Sync Pulse — Part of the LocalTalk protocol that is used to optimize the collision detection mechanism. The sync pulse consists of a transition followed by 2 or more bit times of no transmission. This will cause the Missing clock bit in the 85C30 to be assert, providing for a quick means of detecting activity on the link.

About LocalTalk

LocalTalk provides a relatively low-cost implementation through the use of the Zilog 85C30. Using the 85C30 as the hardware platform, LocalTalk can achieve transfer rates of 230.6 KBytes/sec.

The majority of CPU implementations of LocalTalk have relied up System Software to implement all of the functions of the LocalTalk Link Access Protocol (LLAP). With the release of the IIfx, the burden was taken off of the main processor by the Programmable Interface Controller (PIC) implementation. With the PIC, a dedicated 6502 processor was assigned the duty of implementing the LAP. For various reasons, a new approach has been developed to make the SCC LocalTalk functionality compatible with DMA. This approach allows a central DMA approach to the IO system of the Macintosh. For more details on this issue see PAR Tech Report #2, Effects of Reduced Processor Response on LocalTalk (Samikowski, 1990).

Arioso Model of LocalTalk

In analyzing AppleTalk in general, and LocalTalk specifically, with the Arioso model, the Arioso layers of most interest are the Media Access Control (MAC), Media Input/Output Control (MIOC), and the Physical layers. Figure 4-2. shows the LocalTalk Link Access protocol of AppleTalk partitioned according to the Arioso model.

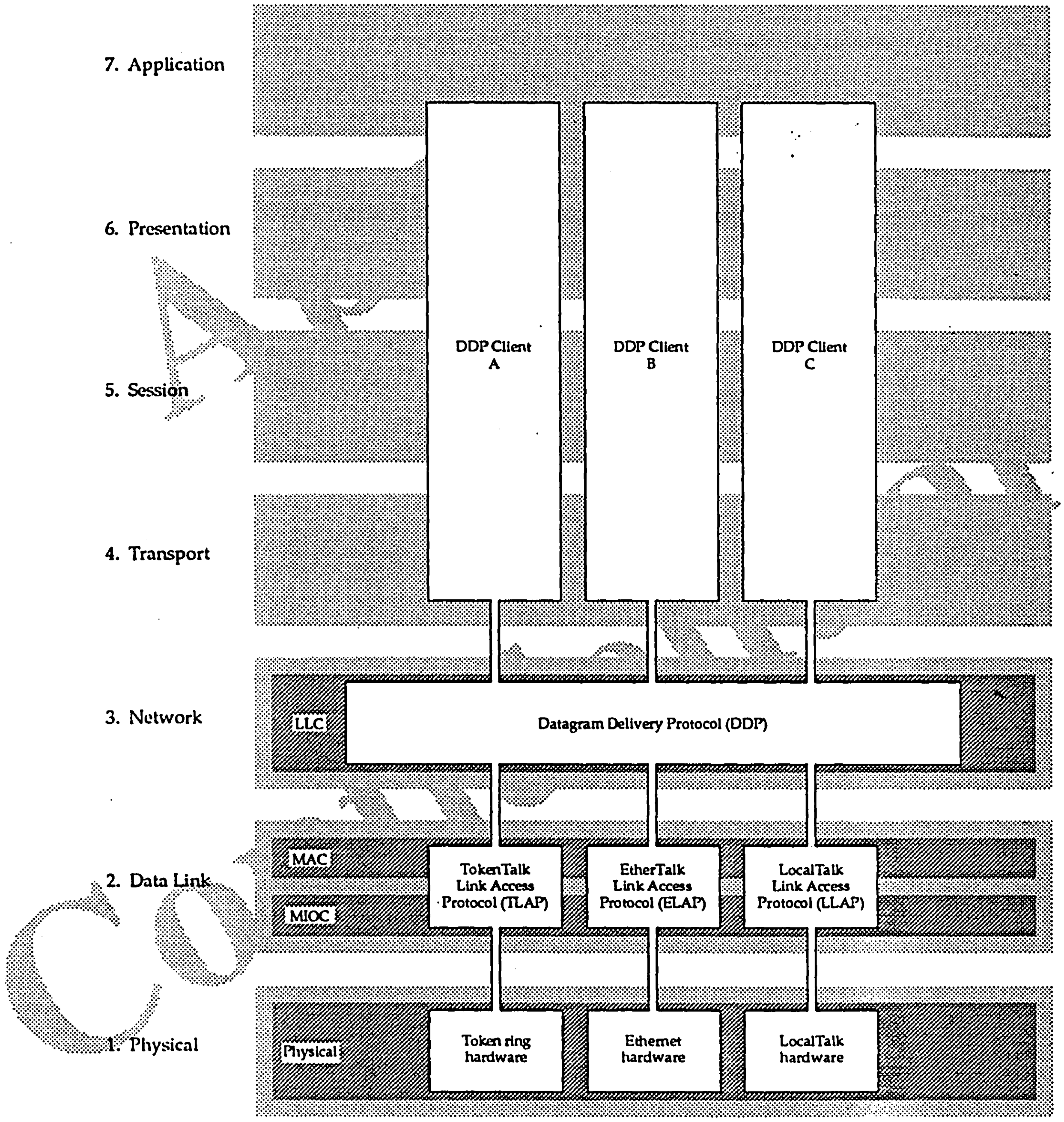


Figure 4-2. Arioso Partitioning of AppleTalk

Figure 4-3 below shows the detailed Arioso block diagram for LocalTalk. The highest level of the block diagram is the Logical Link Control (LLC). This level is identical to the Datagram Delivery Protocol of the

AppleTalk protocol architecture. Data from the LLC to the Media Access Control layer (MAC) is carried in the Frame FIFO. As described in the Arioso Architecture Chapter, each layer communicates with adjacent layers through the use of FIFOs. The FIFO between the layers are intended to be conceptual, that is, there is nothing implicit in the use of the term that requires data to be passed. It is within the architecture that the FIFOs can contain abstract references such as pointers, as well as data.

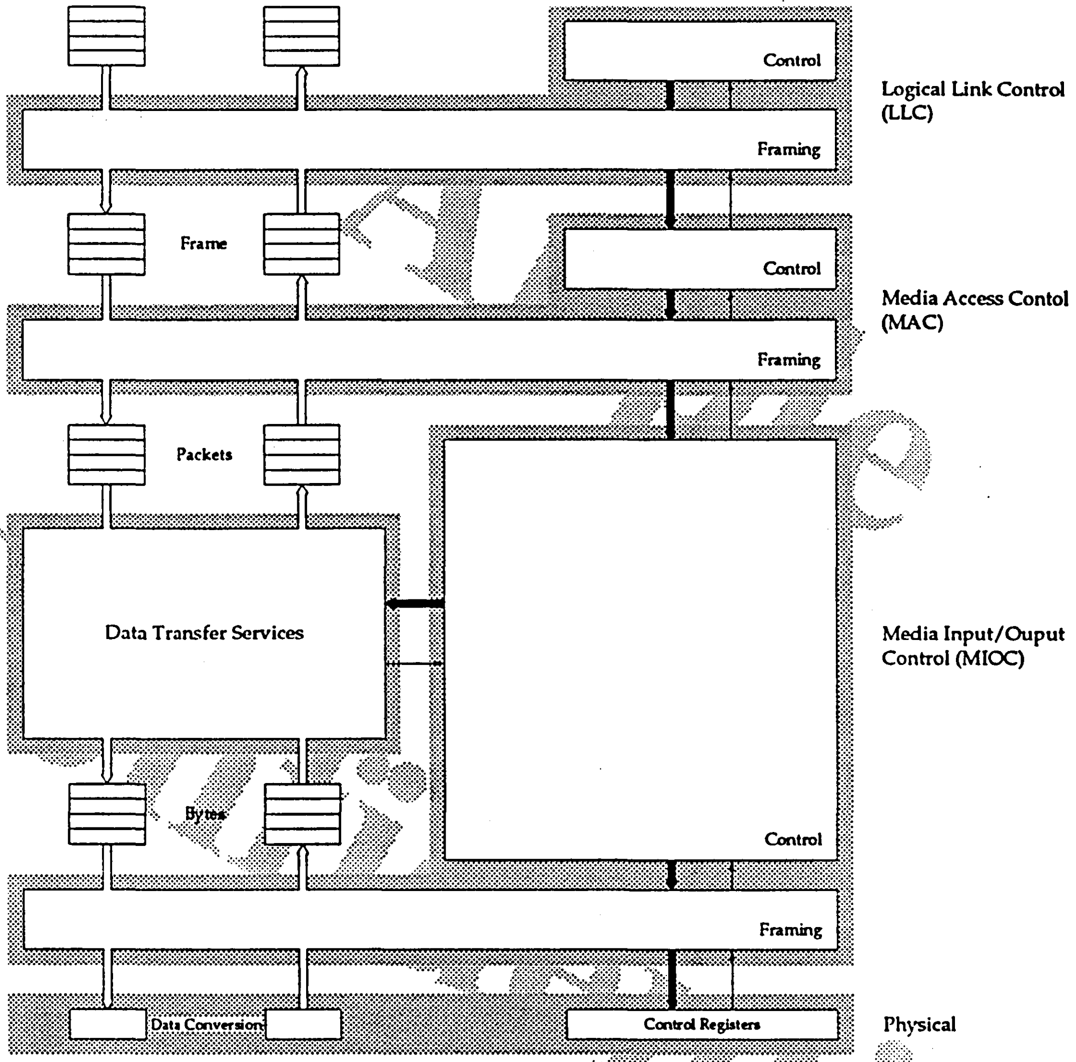


Figure 4-3. Detailed Arioso Block Diagram for LocalTalk

As shown in Figure 2., the majority of the LLAP is implemented with the MAC and MIOC layers defined in Arioso. The MAC receives data from the LLC on a transmit, and provides data to the LLC on a receive. The data structures that the MAC interacts with are the Frame FIFO, between the MAC and the LLC, and the Packet FIFO, between the MAC and the Data Transfer Services layer.

The Data Transfer Services layer represents structures that are required to translate the contents of the Packet FIFO into real data bytes that can be written to the hardware interface located in the Physical layer. As mentioned in the Arioso Architecture Chapter, the presence of the Data Transfer Services layer is to account for system characteristics such as Virtual Memory and DMA. In this detailed analysis of LocalTalk, the functional specifics of these two characteristics will not be documented directly, but what will be specified in the present chapter will be the constraints imposed on VM and DMA by the LocalTalk. For example, this chapter will not specify the functionality of the VM or DMA in the system, rather the detailed analysis will specify the timing requirements for data movement from the Packet FIFO to the Byte FIFO. Therefore, this chapter will attempt to specify the timing required to insure that the LocalTalk protocol is met on both the send and receive.

The Media Input/Output Control (MIOC) layer is responsible for moving data from the Packet FIFOs to the Byte FIFOs on a transmit, and from the Byte FIFOs to the Packet FIFOs on a receive. The MIOC is the only layer that directly interacts with the Data Transfer Services layer and therefore sets all the constraints for the Data Transfer Services layer. In moving data from the Packet FIFOs to the Byte FIFOs, the MIOC must maintain all timing relationships critical to the LocalTalk protocol.

The final stop for a transmission is the Physical layer. The Physical layer represents the absolute lowest level in the Arioso architecture. The Physical layer can be somewhat misleading because it is not simply a single hardware device. Because of the multiple implementations for the LocalTalk Link hardware, there is not a clear line that differentiates hardware functions from software functions that applies to all implementations. Instead of choosing a single implementation, the Physical level is considered to be a minimal set of functions that can perform very simple functions. These functions will be defined and discussed in the detailed description of the Physical layer.

LLC Layer Description

As shown in Figure 3, the LLC layer for LocalTalk in Arioso is identical to the DDP layer in the AppleTalk Protocol Architecture. For a complete description of the DDP layer see *Inside AppleTalk*.

MAC Layer Description

The MAC layer is responsible for maintaining all dialogues between a send and receiver. The DDP sees the MAC layer as providing full service delivery of data. The MAC layer is therefore responsible for acquiring a LocalTalk node ID, establishing a connection, and implementing the collision detection and avoidance mechanism.

When a node is powered-up the first action of the MAC is to acquire a node ID that is used to uniquely identify the system on the network. This node ID is only used by LocalTalk, with other IDs embedded in the packet that are used by different MAC clients. To acquire a node ID, the MAC layer must manage the ENQ/ACK dialog. This dialogue consists of the node randomly selecting node IDs and formatting an ENQ packet that is transmitted over the link. If another node out on the link has the ID contained in the ENQ packet, it will respond with an ACK packet. When detected by the MAC, it indicates that the node ID is taken. This process continues until the sending node chooses an ID that is not answered. This ID then becomes the source ID for all LocalTalk transmissions.

Figure 4-4 shows the general flow of the ENQ/ACK dialogue.

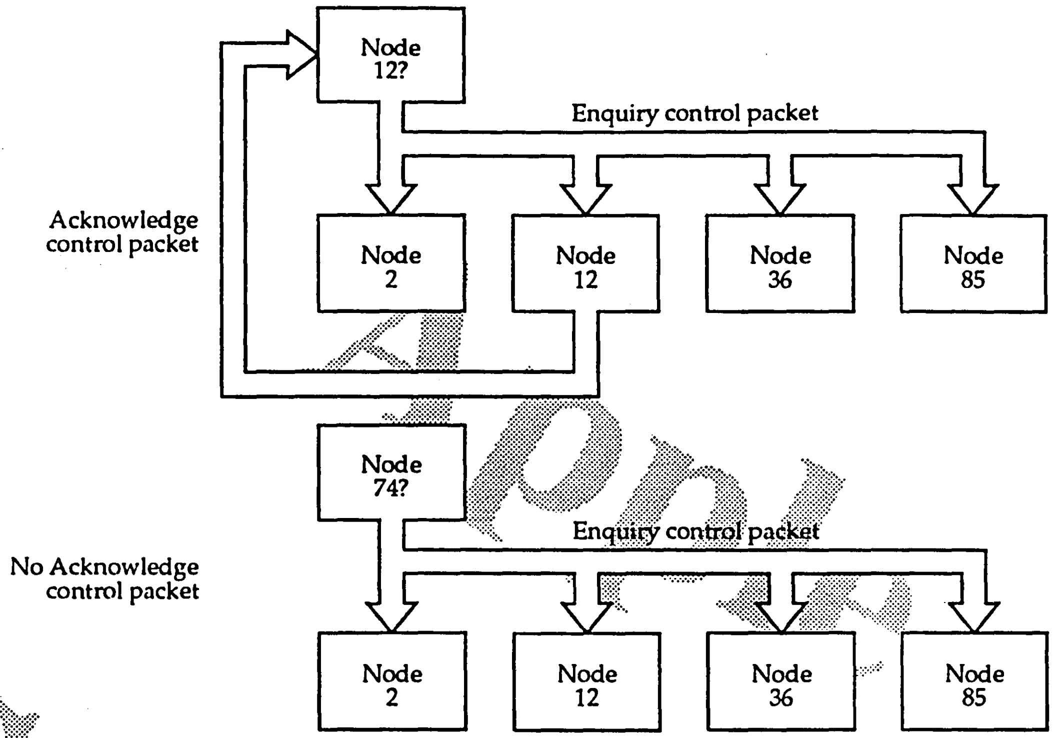


Figure 4-4. Process flow of the ENQ/ACK Dialogue

Once the node ID has been established, the node is free to transmit data over the link. To transmit over the link, the sending node must establish a connection with the receiver through the RTS/CTS dialogue. In the RTS/CTS dialogue, the sender transmits an RTS packet, and awaits a CTS packet from the intended receiver. If the sender does not receive the CTS within the allowable time, the sender assumes that a collision has occurred, and the transmission is attempted again later. If the CTS is received within the allowable time, the sender determines that it has access to the link, and sends the data packet. The transmission of the data packet, terminates the dialogue, and the link is free for another sender.

Figure 4-5 summarized the LLAP transmission dialogs.

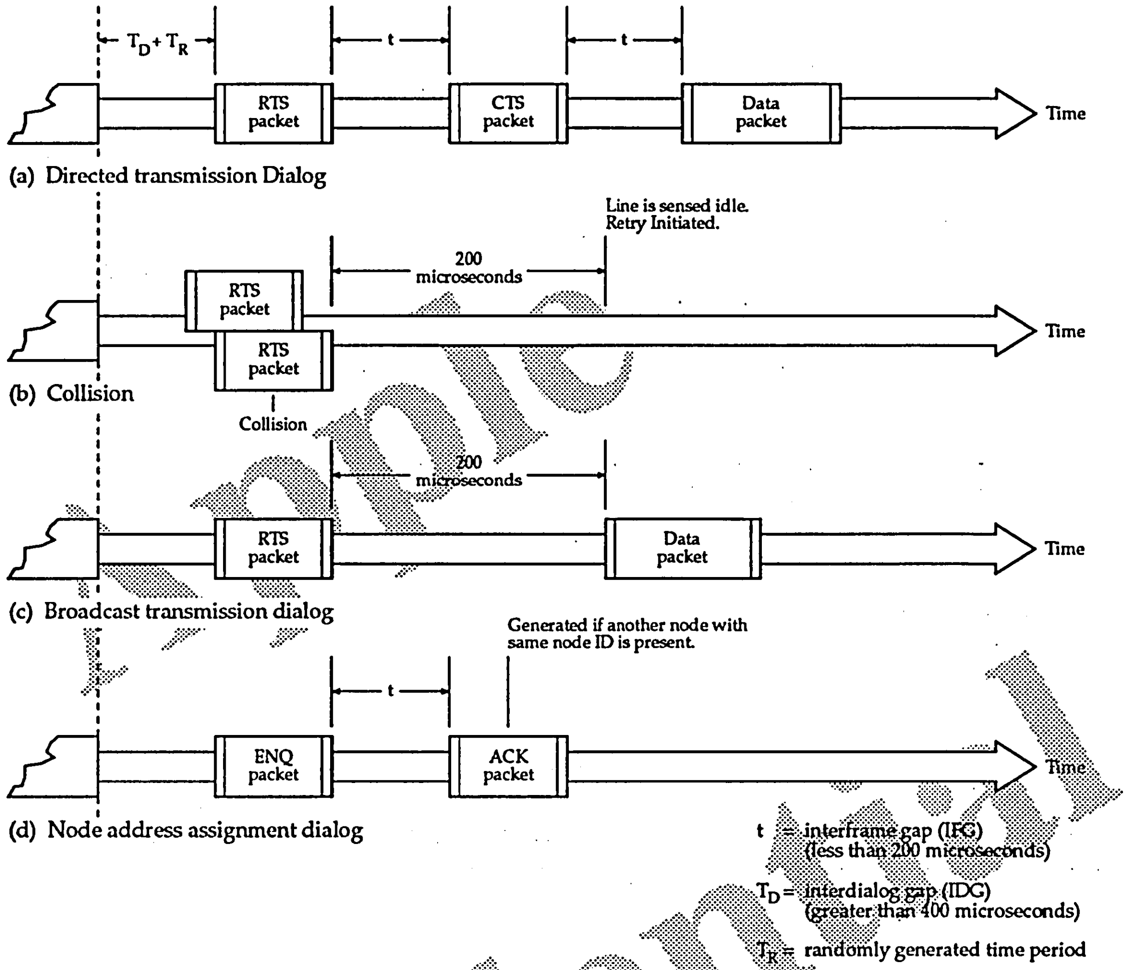

$T_D + T_R$
t
t
RTS packet
CTS packet
Data packet
Time
(a) Directed transmission Dialog
Line is sensed idle. Retry Initiated.
200 microseconds
RTS packet
RTS packet
Time
(b) Collision
Collision
200 microseconds
RTS packet
Data packet
Time
(c) Broadcast transmission dialog
Generated if another node with same node ID is present.
t
ENQ packet
ACK packet
Time
(d) Node address assignment dialog
t = interframe gap (IFG) (less than 200 microseconds)
T_D = interdialog gap (IDG) (greater than 400 microseconds)
T_R = randomly generated time period

Figure 4-5. Examples of LocalTalk Dialogs

Under the Arioso model, the MAC layer is not required to understand the intricacies of detecting a collision on the line. This is left for the MIOC layer. During a transmission, the calls to the MIOC will return a status indicating whether a collision has occurred. This information can be used by the MAC to sense the relative loading of the network and vary the inter-frame timing to optimize for collisions. It is therefore the responsibility of the MAC layer to make use of the collision information that is available from the MIOC.

Arioso provides for two methods for the MAC to receive a packet from the link. The first is through the Packet Available indication (discussed in detail in later sections). This indication is asserted when a complete packet has been received and stored in memory and therefore represents an after-the-fact notification.

The second method is to monitor the status of the packet FIFO with the calls documented in the section titled: MAC in Detail. This approach provides the advantage of being notified as data is filling the FIFO and could lead to more optimized implementations. On the other hand, making use of this method would significantly complicate the implementation of the MAC.

In the sections that follow, the first method will be documented in detail, while the FIFO paradigm calls will be provided for completeness only.

MIOC Layer Description

The MIOC layer is responsible for maintaining all single ended transmission. Unlike the MAC, the MIOC does not monitor or control the relationship between any of the packet in the LocalTalk dialogues. The primary function of the MIOC is to provide the hardware control to transmit or receive a single packet over the network. As such the MIOC is not concerned with validating responses.

The transmission of a packet is started by a call from the MIOC. When the call is initiated the MIOC adds additional information to the data formatted by the MAC and begins the data transmission. Prior to the transmission, the MIOC will check the state of the line to determine whether it is free. If it is free, the MIOC will take data and, with calls to the Physical layer, cause the data to be sent over the link. If the MIOC detects that the line is busy, it will terminate the send and indicate, through the result, that the call was terminated due to a collision.

While receiving a packet, the MIOC is responsible for filtering the packets according to their destination node number. The node ID was determined by the MAC and was given to the MIOC with a call from the MAC. All packets that are not destined for the node are ignored, and are not seen by the MAC. When a packet is available that is destined for the mode, the MIOC will transfer the data to the MAC and check for CRC.

Physical Layer Description

The Physical layer is responsible for the actual interfacing to the link. As currently defined in the Arioso model for LocalTalk, the Physical Layer does not correspond to any single hardware implementation for LocalTalk. The primary responsibility of the Physical layer is to provide serial to parallel and parallel to serial conversion, as well as classification of the bytes that are sent and received.

MAC Layer in Detail

As described earlier, the MAC layer is responsible for maintaining the dialogues that are used to acquire the LocalTalk node ID and transfer data. Each of these dialogues will be described, from the perspective of both transmit and receive, in detail below. Following the dialogue description, a detailed description of all the calls will be provided.

ENQ/ACK Dialogue (Transmit)

Upon initialization, a LocalTalk node will initiate a dialogue to acquire a valid node ID. A flow diagram for the transmitter is shown in figure 4-6.

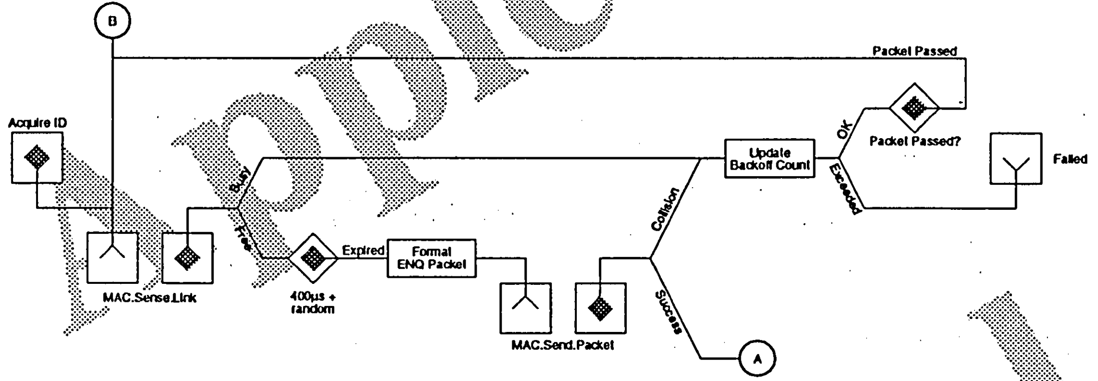


Figure 4-6. ENQ/ACK Dialogue

The first operation performed by the MAC is to sense the state of the link. This is performed through the MAC.Sense.Link call to the MIOC layer. The MIOC will return status indicating that the link is available or busy. If the link is busy, the MAC will update the back-off count and then defer. The process of deferring returns control to higher levels, waiting for the current packet on the line to complete. When the current packet has passed, represented by the Packet.Passed indication, the MAC will return to sample the line.

If the link is available, the MAC will start a timer that for 400µs plus a random component that is a function of the back-off count. At this point control can be passed to a higher level. When the timer expires, represented by the Timer.Expired indication, the MAC formats an ENQ packet with the MAC.Send.ENQ call to the protocol framing of the MAC. This call formats the packet type as ENQ and sets the destination to the node ID to be tried. The MAC then issues a MAC.Send.Packet call to the MIOC.

At the completion of the MAC.Send.Packet command, the call will return status to indicate whether the packet send was completed or if a collision occurred. If a collision occurred the MAC updates the back-off count and defers.

If the ENQ is successfully transmitted, the MAC initiates a 200µs timer. This portion of the process is shown in Figure 4-7. Once the timer has been initiated control can be transferred to other tasks. There are two possible ways to re-enter the MAC at this point. If the timer expires, as represented by the Timer.Expired indication, then the MAC assumes that the node addressed in the ENQ frame is free. The MAC takes this as its node ID, and writes it into the MIOC with a MAC.Write.ID call. The node ID is then used by the MIOC to filter packets that are on the link.

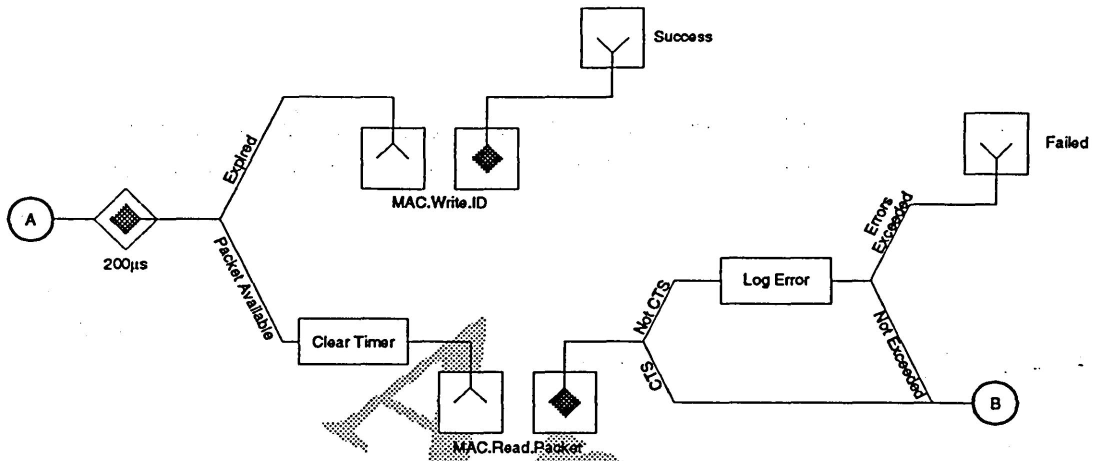


Figure 4-7. ENQ/ACK Dialogue (cont.)

If a Packet.Available indication is received, the MAC clears the current timer and then reads the packet with a MAC.Read.Packet call. As the packet is read, the protocol framing portion of the MAC classifies and checks the packet for errors. If the packet is an ACK packet, the MAC will choose another node ID and try the process again. If the packet is not a valid CTS packet, the error is logged and the process begins again.

RTS/CTS Dialogue (Transmit)

The RTS/CTS dialogue is the main vehicle for transmitting data from a sender to a receiver. A flow diagram for the RTS/CTS dialogue is shown in figure 4-8.

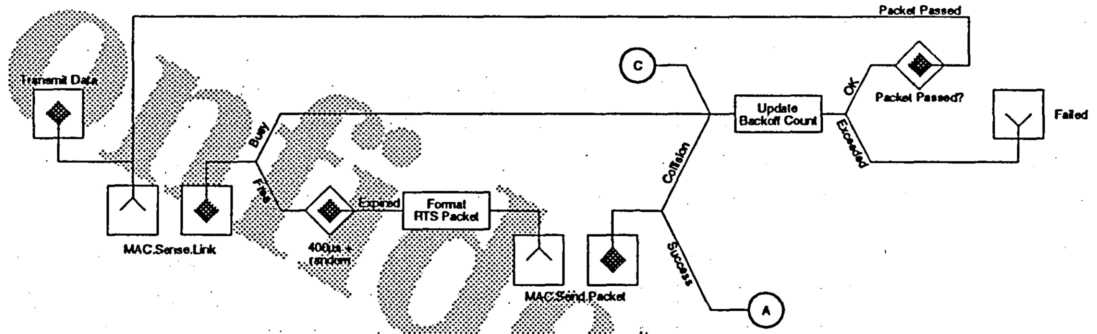


Figure 4-8. RTS/CTS Dialogue

The RTS/DTS dialogue is initiated by a MAC client to send data to an identified receiver. As in the ENQ/ACK dialogue, the MAC begins by sensing the link for activity. If the link is busy, the MAC will update the back-off count and then defer. If the line is available, the MAC will initiate a timer for 400ms plus a random component. When the timer has expired, represented by the Timer.Expired indication, control returns to the MAC. The MAC formats the RTS packet with the MAC.Send.RTS call to the protocol framing layer within the MAC. The MAC then issues the MAC.Send.Packet to the MIOC to transmit the packet. Once the send operation is terminated, the call will return status to indicate whether the RTS packet was successfully sent. If the call returns status that there was a collision, the MAC updates the Back-off count and then defers.

The flow for a successful RTS send continues in Figure 4-9.

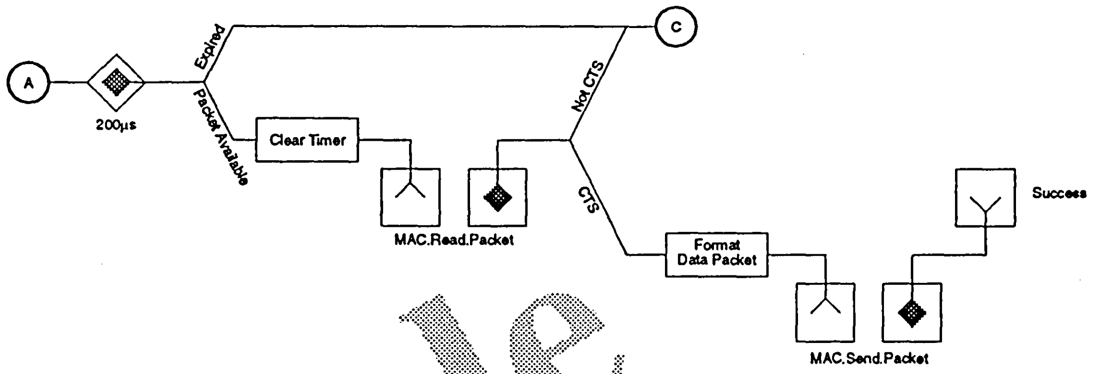


Figure 4-9. RTS/CTS Dialogue (cont.)

After the successful transmission of the RTS packet, the MAC initiates a 200ms timer. Once the timer has been initiated, control can be transferred to other tasks. There are two ways to re-enter the MAC at this point. If the timer expires, represented by the Timer.Expired indication, the MAC assumes that the send has collided, updates the back-off count, and then defers.

If a packet is received, represented by the Packet.Received indication, the MAC generates a MAC.Read.Packet call to the MIOC. The MAC.Read.Packet call returns the status and location of the packet. As the packet is read, the protocol framing layer in the MAC classifies the packet. If the packet is found not to be a valid CTS packet, then the MAC assumes a collision, updates the back-off count, and defers. If the packet was a valid CTS packet, the MAC determines that it can send the data packet.

To send the data packet, the MAC makes a MAC.Send.Data call to the protocol framing layer to properly set the values of the packet type and destination. Afterwards the MAC generates a MAC.Send.Packet call to the MIOC. At this point the MAC returns a Success code to its caller.

MAC As a Receiver

Unlike the previous dialogues where the MAC enters into the operation at the request of a client, the MAC can be invoked when packets are available from the MIOC. There are three possible instances when this can arise: receipt of an ENQ packet, receipt of a RTS packet, or receipt of a unclassified packet (not ENQ or RTS). Thanks to the services of the MIOC, the MAC will never have to deal with packets that are not destined for its node.

The flow diagram in Figure 4-10 shows the MAC as receiver.

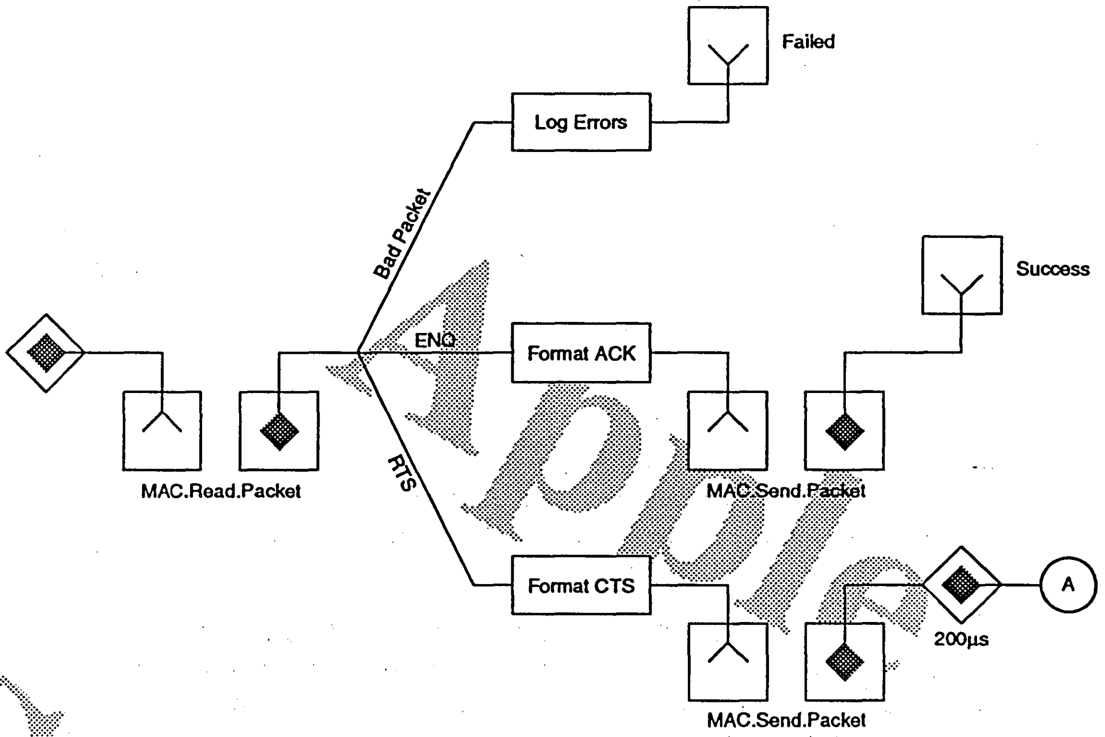


Figure 4-10. MAC Receive Flow

When acting as a receiver, the MAC is entered on a Packet.Available indication. In the MAC, the MAC issues a MAC.Read.Packet call to the MIOC to read in the packet. In the process of reading the packet, the Protocol Framing layer of the MAC classifies the packet and passes on any errors. If the packet is neither a valid ENQ or RTS packet, the error is logged and the MAC is terminated.

If the packet is an ENQ, the MAC must respond with an ACK packet within 200µs. The MAC formats the ACK packet and then sends it out with a MAC.Send.Packet call to the MIOC. At this point the MAC terminates without checking on the status of the send. In the current LocalTalk protocol, collision detection and avoidance are only performed during the exchange of the RTS/CTS dialogue.

If the packet is a RTS, the MAC must respond with a CTS packet within 200µs. The MAC formats the packet and then sends it out with the MAC.Send.Packet call to the MIOC. The MAC initiates a 200µs timer and can return control to another level. At this point there are two possible ways to re-enter the MAC (Figure 4-11).

If the MAC is re-entered with a timer expired indication then the operation is considered a failure and is terminated with a failed indication.

If the MAC is re-entered with a Packet.Available indication, the MAC generates a MAC.Read.Packet call to the MIOC to read the packet. As the packet is read, the Protocol Framing layer of the MAC classifies the packet and reports errors. If the packet is a valid data packet the operation is terminated with a success indication. If the packet is not a valid data packet, the error is logged and the operation is terminated with a failed indication.

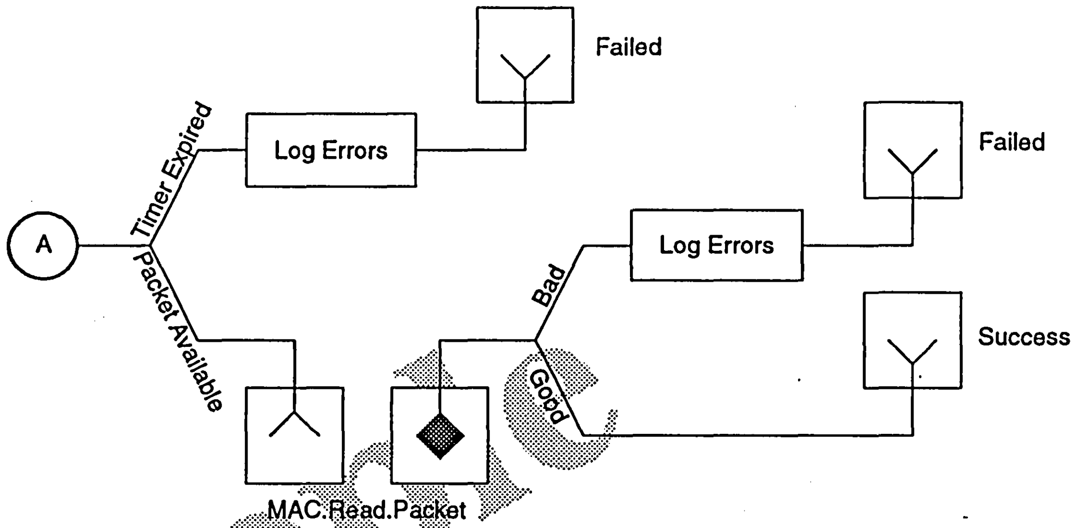


Figure 4-11. MAC Receive Flow (cont.)

MAC Layer Calls

`MAC.Send.RTS()` Request

Command	This command causes the Protocol Framing layer within the MAC to properly add the RTS packet type to the packet to be sent.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MAC.Send.CTS()` Request

Command	This command causes the Protocol Framing layer within the MAC to properly add the CTS packet type to the packet to be sent.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MAC.Send.Data()` Request

Command	This command causes the Protocol Framing layer within the MAC to properly add the Data packet type to the packet to be sent.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MAC.Send.ENQ()` Request

Command	This command causes the Protocol Framing layer within the MAC to properly add the ENQ packet type to the packet to be sent.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MAC.Send.ACK()` Request

Command	This command causes the Protocol Framing layer within the MAC to properly add the ACK packet type to the packet to be sent.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MAC.Write.ID([NodeID])` Request

Command	Write the LocalTalk node ID to the MIOC to filter unwanted packets.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MAC.Read.Packet()` — Request

Command	Call to get status and location of a received packet.
Return	Return to Caller
Result	[Location, {Good, Bad}] The result is the status of the read. A bad indication can arise from bad framing or bad CRC.

`MAC.Send.Packet()` — Request

Command	Trigger the process of sending a packet.
Return	Return to Caller
Result	{Collision, Success,} Success indicates that the packet was sent. The only way by which the packet can not be sent is if there was a collision on the link. This is indicated through the collision result.

`MAC.Get.Rcvr.Pkt.FIFO()` — Request

Command	Provides an alternate way to pass get data during a receive operation. The command is used to read data from the receive packet FIFO that is filled by the MIOC. This is available to simulate a FIFO paradigm for transferring data.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MAC.Get.Rcvr.Pkt.FIFO.Status()` — Request

Command	Get the status of the receiver packet FIFO. This is available to simulate a FIFO paradigm for transferring data.
Return	Return to Caller
Result	{Full, Empty} Result reflects the status of the receiver packet FIFO.

`MAC.Put.Xmit.Pkt.FIFO()` — Request

Command	Provides an alternate way to pass get data during a transmit operation. The command is used to put data in the transmit packet FIFO that will be used by the MIOC during a transmit. This is available to simulate a FIFO paradigm for transferring data.
Return	Return to Caller
Result	{Full, Empty} Result reflects the status of the receiver packet FIFO.

`MAC.Get.Xmit.Pkt.FIFO.Status()` — Request

Command	Get the status of the transmit packet FIFO. This is available to simulate a FIFO paradigm for transferring data.
Return	Return to Caller
Result	{Empty, Full} Result reflects the status of the transmit packet FIFO.

`MAC.Get.Xmit.Frame.FIFO()` — Request

Command	Provides an alternate way to pass data during a transmit operation. The command is used to get data from the transmit frame FIFO filled by the LLC layer. This is available to simulate a FIFO paradigm for transferring data.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MAC.Get.Xmit.Frame.FIFO.Status()` Request

Command	Get the status of the transmit frame FIFO. This is available to simulate a FIFO paradigm for transferring data.
Return	Return to Caller
Result	{Empty,Full}
	Result reflects the status of the command.

`MAC.Put.Rcvr.Frame.FIFO()` Request

Command	Provides an alternate way to pass data during a receive operation. The command is used to put data into the receive frame FIFO that is used by the LLC This is available to simulate a FIFO paradigm for transferring data.
Return	Return to Caller
Result	{Success,Failed}
	Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MAC.Get.Rcvr.Frame.FIFO.Status()` Request

Command	Get the status of the receive frame FIFO. This is available to simulate a FIFO paradigm for transferring data.
Return	Return to Caller
Result	{Empty, Full}
	Result reflects the status of the receiver frame FIFO

`MAC.Set.Timer([time])` Request

Command	Request to set timer to the value specified in µs.
Return	Return to Caller
Result	{Success,Failed}
	Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MAC.Clear.Timer()` Request

Command	Request to clear the current timer.
Return	Return to Caller
Result	{Success,Failed}
	Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MAC.Sense.Link()` Request

Command	Request to the MIOC to determine the status of the link.
Return	Return to Caller
Result	{Free, Busy}
	The result indicates whether the link is available or in use by another node.

`Packet.Passed()` Indication

Command	Indication that the packet previously on the line has completed transmission.
Return	Return to Caller
Result	{Packet.Passed}
	Indicates that the packet transmission has completed.

`Timer.Expired()` — Indication

Command	Indication that the current timer task has expired.
Return	Return to Caller
Result	{Expired} Indicates that the timer has expired.

`Packet.Available()` — Indication

Command	Indication that a packet is available to be read. This provides a paradigm to read packets in which it is implied that the packet is completely available in memory.
Return	Return to Caller
Result	{Packet.Available} Indicates that a packet is available to be read.

`MIOC.Indication()` — Indication

Command	Indication of a state change in the MIOC layer.
Return	Return to Caller
Result	{Command Code} Command code to indicate the state change in the MIOC layer.

`Rcvr.Pkt.FIFO.Indication()` — Indication

Command	Indication of a state change in the receiver packet FIFO. This is used to implement a FIFO paradigm for transferring data.
Return	Return to Caller
Result	{Command Code} Command code to indicate the state change in the receiver packet FIFO.

`Xmit.Pkt.FIFO.Indication()` — Indication

Command	Indication of a state change in the transmit packet FIFO.This is used to implement a FIFO paradigm for transferring data.
Return	Return to Caller
Result	{Command Code} Command code to indicate the state change in the transmit packet FIFO.

`Xmit.Frame.FIFO.Indication()` — Indication

Command	Indication of a state change in the transmit frame FIFO.This is used to implement a FIFO paradigm for transferring data.
Return	Return to Caller
Result	{Command Code} Command code to indicate the state change in the transmit frame FIFO.

`Rcvr.Frame.FIFO.Indication()` — Indication

Command	Indication of a state change in the receiver frame FIFO.This is used to implement a FIFO paradigm for transferring data.
Return	Return to Caller
Result	{Command Code} Command code to indicate the state change in the receiver frame FIFO.

MAC Timing Requirements

The most important timing requirements of the MAC layer are the times required to respond to a received packet during a dialog. All times are measured from a Packet.Available indication to the Send.Packet call to the MIOC. Table 4-1. shows these times for the MAC.

Table 4-1. Timing for MAC Dynamic Characteristics

Symbol	Parameter	Min	Max	Units	Test Conditions
t_{CTS}	Packet.Available to CTS MAC.Send.Packet	--	200	µs	
t_{Data}	Packet.Available to Data MAC.Send.Packet	--	200	µs	
t_{ACK}	Packet.Available to ACK MAC.Send.Packet	--	200	µs	

MIOC Layer in Detail

As described earlier, the MIOC is responsible for all single-ended transmissions. Unlike the MAC, all transmissions are open-ended, with no feedback on the success of the transmission. In addition to providing services to send and receive single packets, the MIOC provides the services to determine whether the link is busy or free.

Send Packet

During the transmission of a LocalTalk packet, the MIOC is responsible for adding the LocalTalk required framing characters (Figure 4-12). The actual implementations of these characters are provided in the Physical layer. The framing layer within the MIOC is responsible for properly formatting them for the Physical layer. Prior to the MIOC, the data portion of the packet (gray shaded in Figure 12) has been created and deposited in some storage medium.

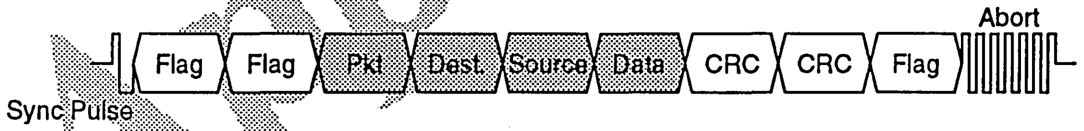


Figure 4-12. LocalTalk Packet Format

Upon command from the MAC layer the MIOC will initiate the send sequence. The first stages of the send sequence are shown in Figure 4-13.

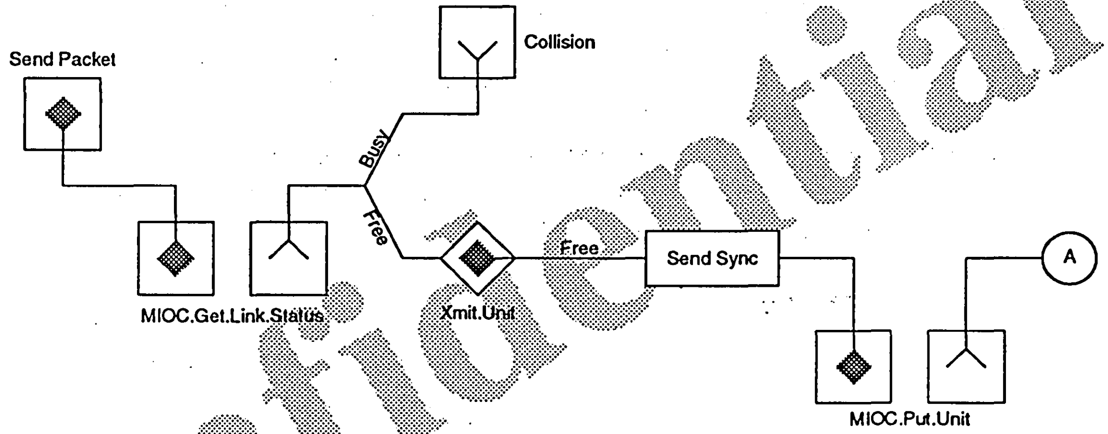


Figure 4-13. Send Packet

After receiving the MAC.Send.Packet command, the MIOC will check the status of the link by issuing the MIOC.Get.Link.Status command. If the link is busy, the MIOC will terminate with a status to inform the MAC that a collision has occurred.

If the link is free, the MIOC will wait on the Xmit.Unit.Free indication from the physical layer. At this point control can be passed to another task. Once the indication is detected, the MIOC will format the sync character of the packet by a call to the MIOC framing layer. The MIOC then transmits the sync character by issuing the MIOC.Put.Unit command.

After completion of the MIOC.Put.Unit command, the MIOC formats two flag characters and transmits them with two MIOC.Put.Unit commands (Figure 4-14).

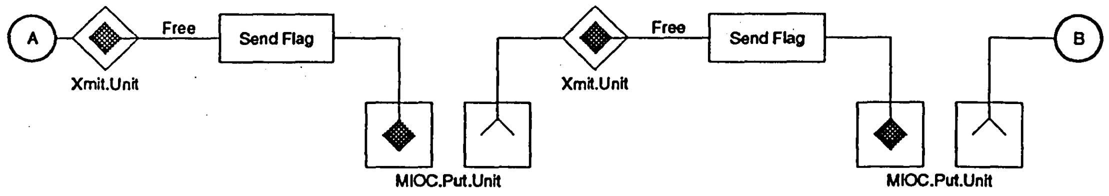


Figure 4-14. Send Packet (cont.)

After transmission of the sync character and the opening flags, the MIOC transmits the contents of the packet (Figure 4-15).

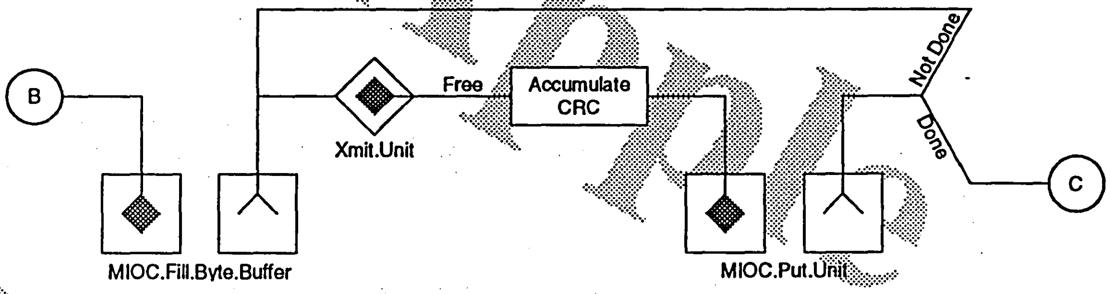


Figure 4-15. Send Packet (cont.)

The first step in transmitting the packet is to signal the Segmentation layer to fill the byte buffer, through the MIOC.Fill.Byte.Buffer command. The timing of the MIOC.Fill.Byte.Buffer call is implementation dependent and must take into account the characteristics of system features such as VM and DMA. For documentation purposes, this call is performed at the last conceivable moment immediately before the data is actually needed. In systems where this call could potentially require the action of VM and DMA, the call should be performed much earlier than shown. Regardless of when the call is actually performed, the Data Transfer Services layer must guaranteed that it can meet the timing required during the send.

The byte buffer is used by the MIOC to stage the data prior to sending the data out on the link. During the data transmission, each byte is accumulated into the CRC and then sent out over the link. As in the other examples, the MIOC waits for a Xmit.Unit.Free indication before issuing the MIOC.Put.Unit command. Depending upon the performance of the machine, this time could be used to give control to other tasks. This process continues until all the bytes of the data portion of the packet have been send. Prior to the MIOC layer the destination and source, as well as the packet type have been added to the data stream, allowing the MIOC to transfer the packet without knowledge of the packet type or format.

After the data has been completely sent, the MIOC transfers the two byte CRC that was computed during the data transfer (Figure 4-16). As in the earlier examples, each byte is transferred when the Xmit.Unit.Free indication is asserted by the Physical layer, by the MIOC.Xmit.Unit command.

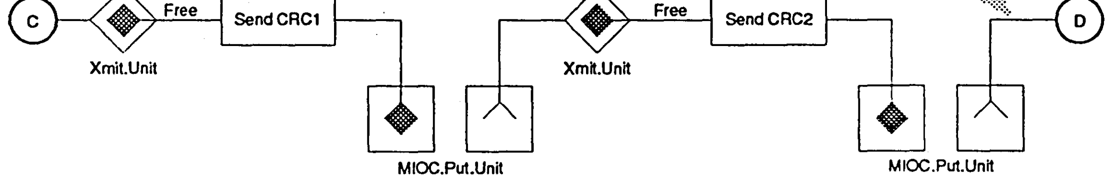


Figure 4-16. Send Packet (cont.)

Finally, after the CRC has been transferred, the MIOC sends out the closing flag and abort character (Figure4-17). In each case, a call to the MIOC framing layer formats the byte before it is sent out over the link with the MIOC.Xmit.Unit. After the abort character has be transmitted, the MIOC returns control to the caller with a successful indication.

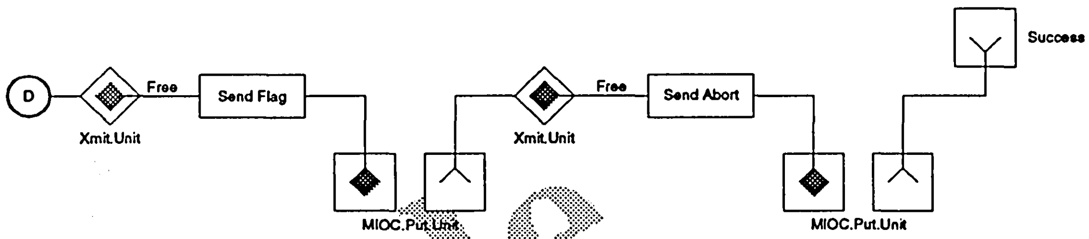


Figure 4-17. Send Packet (cont.)

Store ID

To prevent burdening higher levels in the AppleTalk protocol architecture, the MIOC filters the packets that are out on the link based upon destination. This causes some difficulty because LocalTalk acquires it ID dynamically and this process is handled by the MAC layer. In order to get the ID number into the MIOC, the MAC issues a MAC.Store.ID command. The flow diagram for the action of this command in the MIOC is shown in Figure 4-18.

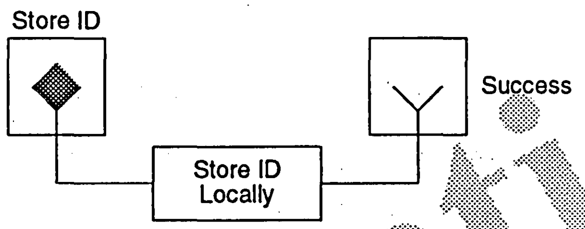


Figure 4-18. Store ID

MIOC as a Receiver

Unlike the send packet process described earlier, the MIOC can become active when data is available from the Physical layer. It is then the job of the MIOC to transfer the data from the Physical layer and check the incoming packet for errors.

While receiving data from the physical layer the MIOC is entered when the Unit.Available indication is asserted. The first portion of the process of receiving a packet is shown in Figure 4-19.

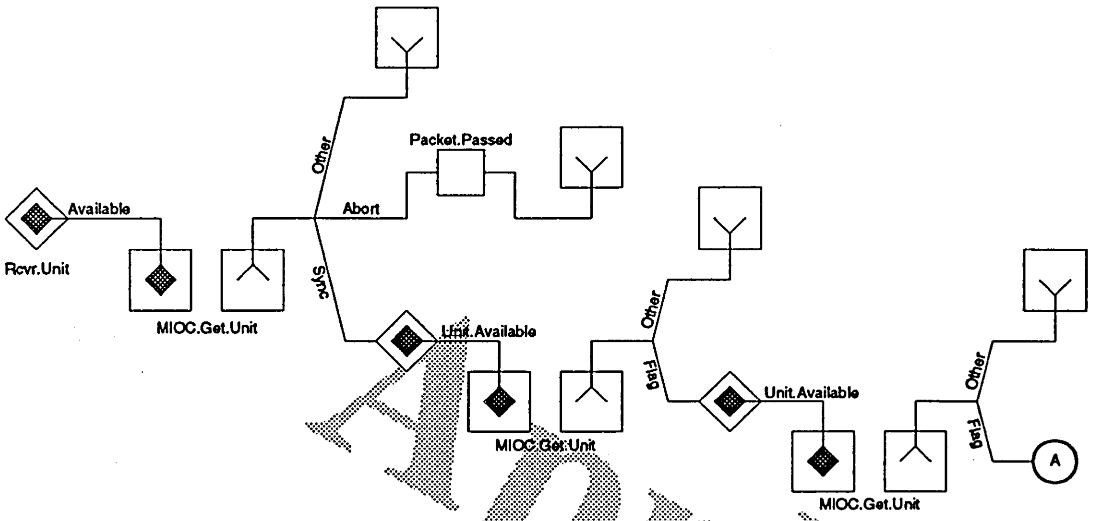


Figure 4-19. Receive a Packet

The first task of the MIOC is to classify the data available from the Physical layer. To do this, the MIOC reads the data from the Physical layer with the MIOC.Get.Unit command and the data is decoded by the protocol framing layer within the MIOC. There are two valid data types to be handled at this point: Abort, and Sync. If the data is neither an abort nor a sync, the MIOC does nothing with the data and returns control to another task.

If the data read from the physical layer is an abort, the MIOC will assert the Packet.Passed indication to mark the end of a transmission. This indication is used by the MAC to start a transmission that may have been deferring (See the MAC Layer in Detail section for more information).

If the data read from the Physical layer is a sync character, the MIOC recognizes this as the start of a LocalTalk packet, and begins the process of receiving the packet. The MIOC then reads the next two bytes, expecting flag characters. If either of the next two characters is not a flag the MIOC terminates and control can be passed to another task. If each of the characters is a flag the MIOC then goes on to the next stage of reception (Figure 4-20).

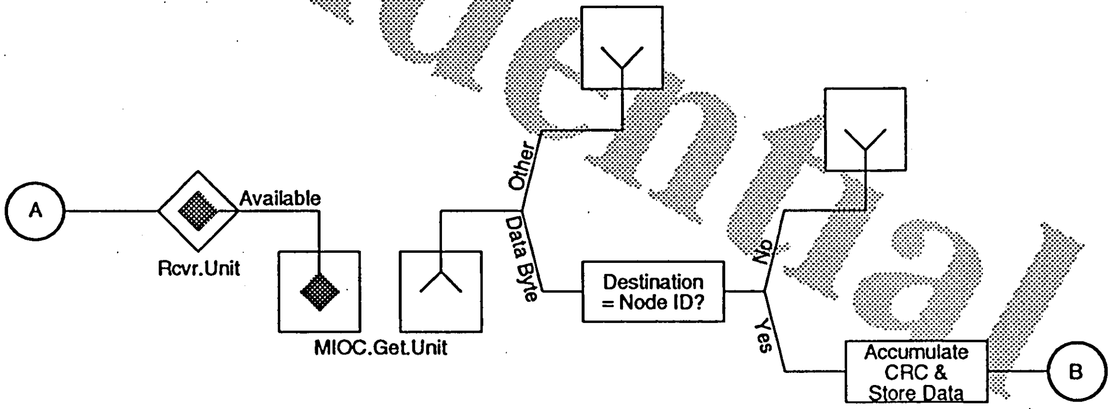


Figure 4-20. Receive a Packet (cont.)

At this point in the packet reception, the next character expected is a data character containing the destination node of the packet. When the Rcvr.Unit.Available indication is asserted, the MIOC reads the data from the Physical layer. If the character is not a data character the MIOC is terminated. If the character is data then the value is compared against the node ID and if the character does not match the node ID, the MIOC is terminated and MAC is not notified. If the character matches, it is stored and the CRC accumulation is started. At this point the MIOC enters a loop that continues until the closing flag is detected (Figure 4-21).

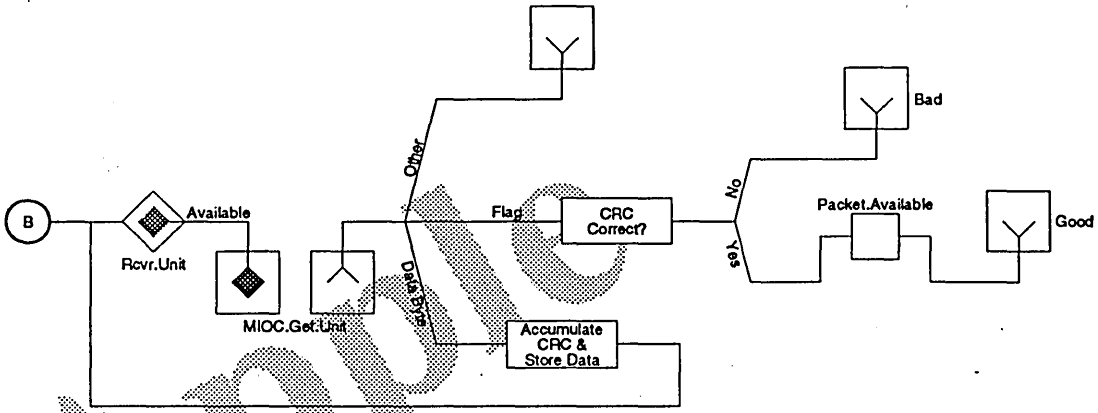


Figure 4-21. Receive a Packet (cont.)

For each data character, the MIOC waits on the Rcvr.Unit.Available indication and then reads the data with the MIOC.Get.Unit command. After the data is read it is classified through the Protocol framing layer in the MIOC. If its a data character, it is accumulated into the CRC and then stored.

If the character is a closing flag, the MIOC assumes that the packet is complete. The CRC is then checked for a good value. If the CRC was good, the Packet.Available indication is asserted and the MIOC terminates with a good result. If the CRC is bad, the MIOC terminates with a bad result.

While the MIOC is receiving the data portion of the packet any character other than a sync or data will cause the MIOC to terminate.

MIOC Layer Calls

`MIOC.Send.Abort()` **Request**

Command	Send an abort unit.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MIOC.Send.CRC()` **Request**

Command	Send CRC unit.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MIOC.Send.Flag()` **Request**

Command	Send a flag unit.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MIOC.Send.Sync()` **Request**

Command	Send a sync unit.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MIOC.Xmit.Data()` **Request**

Command	Call to the Data Transfer Services layer to initiate the conversion of segments to bytes that will fill the byte FIFOs that are read by the MIOC.
Return	Return to Caller
Result	{Success,Failed} Result reflects the success or completion of the command. There are a multitude of reasons that the process could have failed.

`MIOC.Rcv.Data()` **Request**

Command	Call to the Data Transfer Services layer to initiate the conversion of bytes to segments that will fill the segment buffer read by the MAC.
Return	Return to Caller
Result	{Success,Failed} Result reflects the success or completion of the command. There are a multitude of reasons that the process could have failed.

`MIOC.Get.Unit()` **Request**

Command	Get a received unit from physical layer receive buffer.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MIOC.Get.Rcvr.Unit.Status()` Request

Command	Get status of receiver unit buffer.
Return	Return to Caller
Result	{Empty, Full} Result reflects the status of the receiver unit buffer in the Physical layer.

`MIOC.Put.Unit()` Request

Command	Put out the transmit unit to the physical layer transmit buffer
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MIOC.Get.Xmit.Unit.Status()` Request

Command	Get status of transmit unit buffer.
Return	Return to Caller
Result	{Empty, Full} Result reflects the status of the transmitter unit buffer in the Physical layer.

`MIOC.Put.Rcv.FIFO()` Request

Command	Provides an alternate way to pass data during a receive operation. The command is used to transfer data from the physical layer buffers and place them in the byte FIFOs read by the Data Transfer Services layer. This is available to simulate a FIFO paradigm for transferring data.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MIOC.Get.Rcv.FIFO.Status()` Request

Command	Get the status of the receiver byte FIFO.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MIOC.Get.Xmit.FIFO()` Request

Command	Provides an alternate way to pass data during a receive operation. The command is used to transfer data from the byte FIFOs filled by the Data Transfer Services layer and place them in the physical layer transmit buffers. This is available to simulate a FIFO paradigm for transferring data.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MIOC.Get.Xmit.FIFO.Status()` Request

Command	Get status from the transmit byte FIFO.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MIOC.Fill.Byte.Buffer()` Request

Command	Command to the Data Transfer Services layer to fill the byte buffer.
Return	Return to Caller
Result	{Success,Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`MIOC.Get.Link.Status()` Request

Command	Get status from the Link.
Return	Return to Caller
Result	{Free, Busy} Result reflects the status of the Link. Free indicates that no other sender is currently transmitting on the line.

`Rcvr.Unit.Available()` Indication

Command	A physical unit is available from the Physical layer.
Return	Return to Caller
Result	{Available} Indicates that a unit is available to be read from the Physical layer.

`Xmit.Unit.Free()` Indication

Command	Indication of a state change in the receiver unit buffer.
Return	Return to Caller
Result	{Open} Indicates that space is available to transmit data from the Physical layer.

`Xmit.FIFO.Indication()` Indication

Command	Indication of a state change in the transmit byte FIFO. This indication can be used to trigger operations that utilize the FIFO paradigm to transfer data from the Data Transfer Services to the physical layers.
Return	Return to Caller
Result	{Command Code} Command code to indicate the state change in the transmit FIFO.

`Rcv.FIFO.Indication()` Indication

Command	Indication of a state change in the receiver byte FIFO. This indication can be used to trigger operation that utilize the FIFO paradigm to transfer data from the physical layers to the Data Transfer Services layers.
Return	Return to Caller
Result	{Command Code} Command code to indicate the state change in the receive FIFO.

MIOC Timing Requirements

The MIOC presents the most severe timing requirements of LocalTalk. One of the complicating factors in specifying timing constraints is the depth of FIFOs and memory structures used to buffer data. In the specifications below, the timing requirements will be specified in units of µs/byte where appropriate.

Table 2. Timing for MAC Dynamic Characteristics

Symbol	Parameter	Min	Max	Units	Test Conditions
t_{XMIT}	XMIT.Unit.Free to MIOC.XMIT.Unit	--	34.72	µs/byte	
t_{RCV}	Unit.Available to MIOC.Get.Unit	--	34.72	µs/byte	

Physical Layer in Detail

The Physical layer is composed of the minimal registers and circuits that could be address by a software implementation. That doesn't necessarily mean that some of the functions documented earlier can't be implemented in hardware, rather, the Physical layer contains "hardware" that is intuitively hardware. Therefore the best means to describe the Physical layer is with the conceptualized block diagram shown in Figure 4-21.

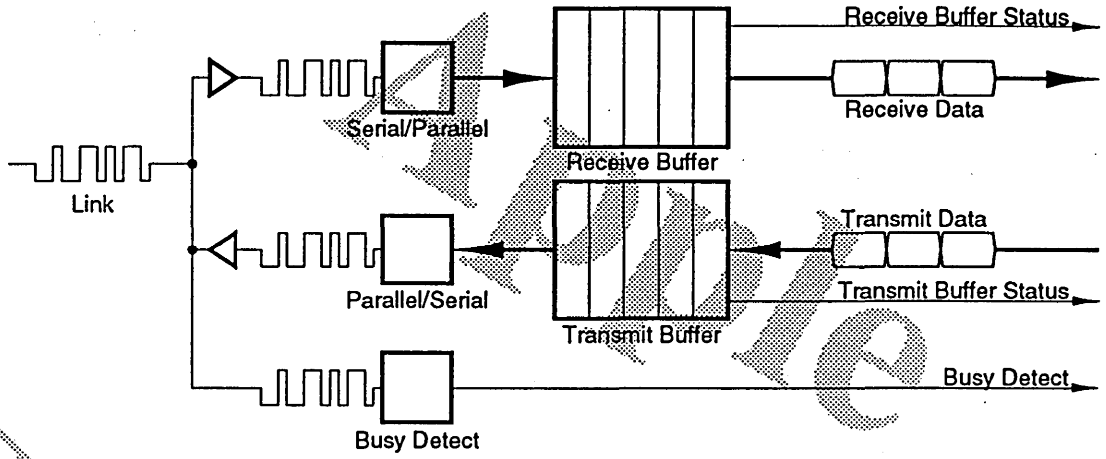


Figure 4-21. Conceptualized Block Diagram for the Physical Layer

There are three main components of the Physical layer: Serial/Parallel Converter, Parallel/Serial Converter, and Busy Detect.

The Serial/Parallel converter is used to receive data from the link and convert it to parallel form. The data is then written to a receive buffer that can be any size depending on implementation. The receive buffer in turn provides the parallelized data to the MIOC layer as well as an indication of the status of the buffer.

The Parallel/Serial converter is used to transmit data over the link. Bytes are written in parallel form to the transmit buffer and then are serialized. The transmit buffer can be any size depending on implementation. The transmit buffer provides an indication to the MIOC to be used while sending a packet.

The final component is the busy detect. The purpose of the busy detect is to monitor the status of the link and provide status upon demand. This line is used by the MIOC and the MAC to determine whether a collision has occurred during a transmit.

References

PAR Technical Report #2, "Effects of Reduced Processor Response on LocalTalk", Scott Sarnikowski, April5,1990

P.A.R Technical Note #3, "LocalTalk, I/O, and Cathedral DSP Performance", John Atwood, April 3, 1990

Inside AppleTalk, Gursharan S. Sidhu, Addison Wesley, 1989

Z8030/Z8530 SCC Serial Communications Controller Technical Manual, Zilog Inc., November 1989

IIci ROM Source Code, Version 1.0, Apple Computers

Chapter 5

Modems

version 1.0

Henry Kannapell

About This Chapter

This chapter describes the current model of modems as they integrate into the Macintosh. Engineering has created an architecture for modems called the Datapump Toolbox; as a result, this document will not fully describe the modem interfaces and protocols that have been defined. Instead, the document will make references to the appropriate modem specifications.

This will show the appearance of the modem system in Arioso. As usual, there are several layers to the model.

The modem system described here is a model; the actual calls and interface functions are not necessarily the actual ones that are used in the current or future modem code. This document is a logical breakdown of the modem protocol for Apple implementations. The goal is to define elements of the protocol that can become CPU independent. The Datapump ToolBox defines a set of hardware independent primitives to the modem system. Only these should have to change for each new modem implementation.

Glossary

Modem terminology is defined in this section.

Bell 103 — 300 bps modem, 300 baud
Bell 212A — 1200 bps modem, 300 baud

DATR	— Data transfer. Datapump Toolbox name for modem control and data movement calls
T.30	— CCITT fax modem specification
V.21	— CCITT 300 bps full duplex modem standard, 300 baud

V.22	— CCITT 1200 bps full duplex modem standard, 600 baud
V.22bis	— CCITT 2400 bps full duplex modem standard, 600 baud
V.23	— CCITT 1200 bps half duplex modem standard, 1200 baud
V.24	— CCITT version of RS-232
V.25	— CCITT phone calling/ answering protocol
V.27ter	— CCITT 4800 bps half duplex modem with adaptive equalizers, 1600 baud
V.29	— CCITT specification for 9600 baud full duplex service using two line pairs, each half duplex
V.32	— CCITT specification for 9600 baud full duplex service over standard telephone lines. This requires a complex adaptive echo cancellation filter. 2400 bauds
Hayes Compatible	— a modem that responds to a set of ASCII based commands defined by Digital Communication Associates (DCA).
PSTN	— Public Telephone Network

About Modems

Modems are devices that use analog signaling to send digital data. They are usually a composed of a modulator (digital data to analog signal transmitter) and a demodulator (analog signal to digital data receiver). They represent the oldest and most common method of computer to computer communications, since they can use the world wide telephone plant to form connections.

Unfortunately, many elements of the control of modems and many serial data communication devices are inconsistent or unique. This leads to great difficulty in both standardizing control sequences for the devices and in system layering of modem control software.

The CCITT, a worldwide organization has developed standards for signaling, telephone call processing, and protocols to make this situation more consistent. The V series of recommendations, which are published by the CCITT, are the foundation of modern modem behavior.

Modems create data connections between computers. There are two basic methods of transmitting data over a connection — synchronously or asynchronously. This only refers to the digital interface however; all modems from V.22 onward employ synchronous transmission for the analog symbols.

If asynchronous data is to be transmitted, it is first converted into a synchronous packet and then sent over the line. These synchronous packets are the usual 10 bit packets, but they are aligned on the synchronous edges of the analog modem link.

In the synchronous case, the data bits are transmitted using the clock of the underlying modem link also. In this case, no asynchronous to synchronous conversion is required.

In the asynchronous case, the clocks of the two computers operate at nearly the same rate. Using oversampling of the signal, the modem recovers the digital data from the analog symbols.

Modems require a clear channel to communicate. Normally this is either the telephone plant or direct wire connections. There is an assumption that the channel between them does not support DC signaling. Further, the channel has limited available bandwidth. Often this is the telephone plant that only has about 4 Khz bandwidth.

For dial up lines, the modem must handle ring processing and the generation and recognition of calling tones. This requirement is in addition to any tone generation and detection that is part of the modem operation.

Dialing processing has timing constraints that are beyond the ability of current Macintosh central CPUs to process in normal operation, as it requires below 10 millisecond guaranteed timing. Normally an external modem performs the dial processing; it has its own processor, and thus does not have the timing limitations of the Macintosh. Apple is considering new systems that would include the modem control in the main CPU, and thus has to deal with the timing issues.

Modems consist of three parts. There is the Phone line processing (called the PSTN functions in the Datapump toolbox), the Analog signaling functions (called the DATR functions) and finally the buffer interface to the rest of the software system. Consider data transfers that do not use all three levels. If two modems connect, with no intermediate telephone switching, the PSTN section does not need to exist. An example is long distance data transmission over dedicated lines. This is also a descriptor for leased line configurations. Alternately, for short distances, the modems themselves are unnecessary; instead, RS-232 drives might directly connect two devices through a null modem. In this case there is only digital signaling. Carrier detection, data modulation, etc., is unnecessary.

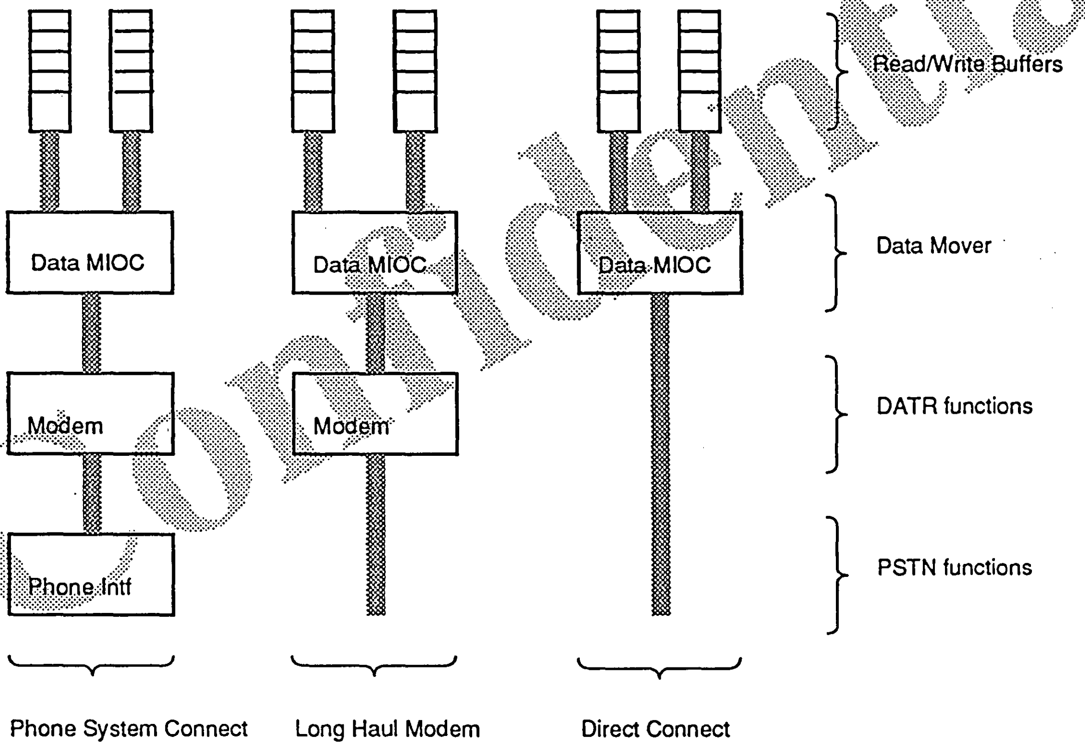


Figure 5-1. Different Modem connections illustrating partitioning choice.

At the bottom are the means of creating the direct channel. This is either the telephone network and associated dialing systems, or a direct connection. We will consider the PSTN functions in this Arioso description of modems.

The next level is the modem signaling itself, the operation of the modulator and demodulator. This uses the connected channel to transmit and receive analog signals. It does not matter whether it is a direct connect or a connection over a telephone line to the modem.

These analog signals are the physical layer of the next level of the system. The next higher connection of the modem level uses the modem signaling to send or receive bits and special delimiters. The Data MIOC layer groups, packages, and checks the CRC of the data. It connects to the rest of the system by either a set of frames or a set of bytes. Asynchronous data requires bytes; synchronous data requires frames. This document will assume that the grouping of bits into bytes and the clock recovery and use are all part of the Data MIOC layer.

The synchronous and asynchronous data types travel different paths from this point. Normally the synchronous frames go to a protocol handler that will process the frames and respond to errors. The async data usually goes to a buffer pool and flow control algorithms. As seen from figure 5-2, there are several possible clients of the synchronous data. For the async data, there is usually only one client.

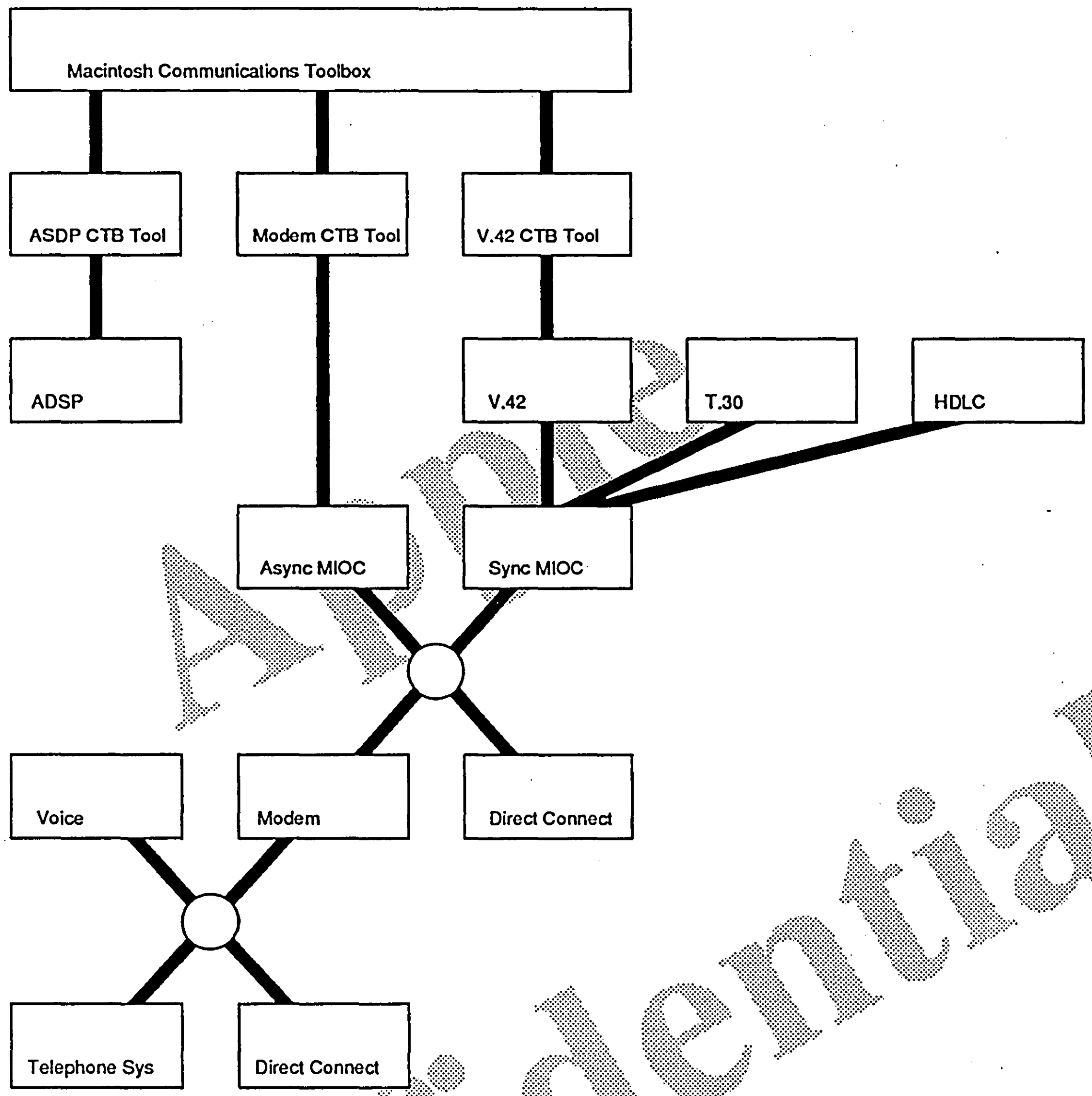


Figure 5-2. Clients of the Modems and related modules

Implementation Tree

Now let's consider the current Macintosh Modem software interface. There are several interfaces to Modem code modules. These correspond to the different implementations of the Modem. The Macintosh 3615 user's guide describes the integration of modems and the Macintosh.

There are two basic types of modems — external and internal modems. External modems are normally Hayes compatible connected to the serial ports of the Macintosh. Global Village has a modem that connects to the ADB interface.

Internal modems are either bus based or motherboard based. Nubus modems such as Mozart fit into the first category. In the second category, there are 3 different types. The portable design center is using the Rockwell Datapump series of modems that include some telephon signaling, modem signaling, clock extraction and CRC checking. The Cyclone is using the DS to do the same basic functions. The interface here is a set of buffers between the Mac CPU and the DSP, and messaging using the standard DSP control definition. The RISC based products are planning to do the above functions in the main RISC processor.

The Datapump Toolbox ERS describes the proposed common interface to these disparate modem implementations.

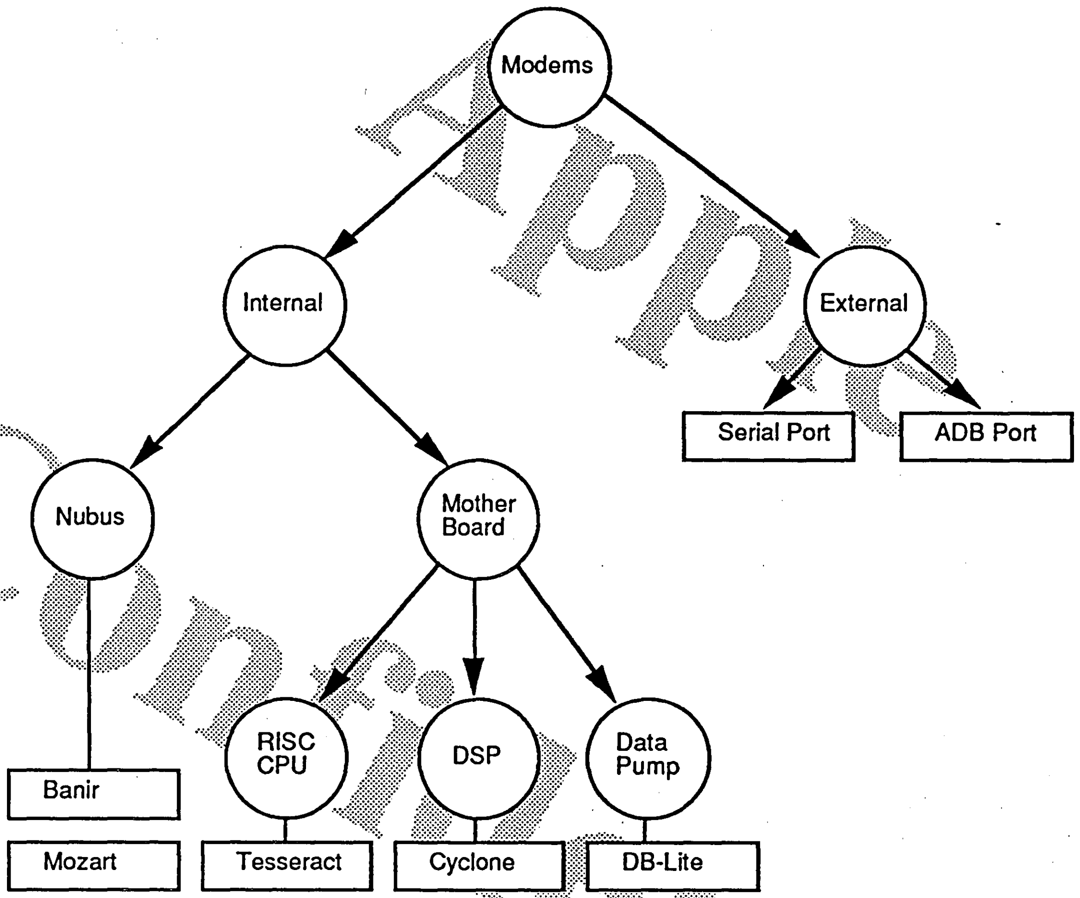


Figure 5-3: Modem Implementation tree

Arioso Model of Modem

The remainder of this chapter defines the Arioso layers of the Modem system. Each layer has an interface to the layer above it and an interface to the layer below it. The realization of a layer is an *implementation.*

The layers begin with the I/O devices at the bottom, and continue up to the device driver interface at the top.

As mentioned, modems can be viewed as a hierarchical stacking of entities. Unfortunately, this did not fit directly into the Arioso model. Arioso is a description of the layered breakdown of only a single protocol entity. Modems may involve several systems, each using the data of another system.

The description of modems thus looks a little different than that of other systems. For example, if a telephone system connection exists, the first phase of establishing a connection executes the phone system protocol. After the phone system connection is complete, the modem can't tell if it is directly connected or connected over a phone system.

Because of this, modems are described as a grouping of entire I/O systems (phone, modem, data connection) rather than trying to roll the entire system into a single Arioso model. Each can be thought of as separate entities; voice circuits may be a client of the phone connection, just like modem signals can.

Each I/O system involved in the modem is relatively simple; we divide them only into MIOC and MAC layers. The MIOC layer is the protocol for sending data out to the medium; the MAC layer uses the MIOC to interact with the connected device (usually another modem).

The serial data stream may be direct connected, or may go through a modem; the modem may be direct connected or may go through a telephone plant. The following three figures show the possible configurations. The first is serial data stream directly connected to a wire. This is the direct wired RS-232 case. The second is a modem directly connected to another modem. The last case is a modem connected to the phone plant.

This can also be thought of as a time compression of connection establishment. First the phone system connection is made, then the modem connection is made, and then the data connection is made.

The description of the modem will assume the most complex model, that is one that has the phone plant, the modem, and the data connection.

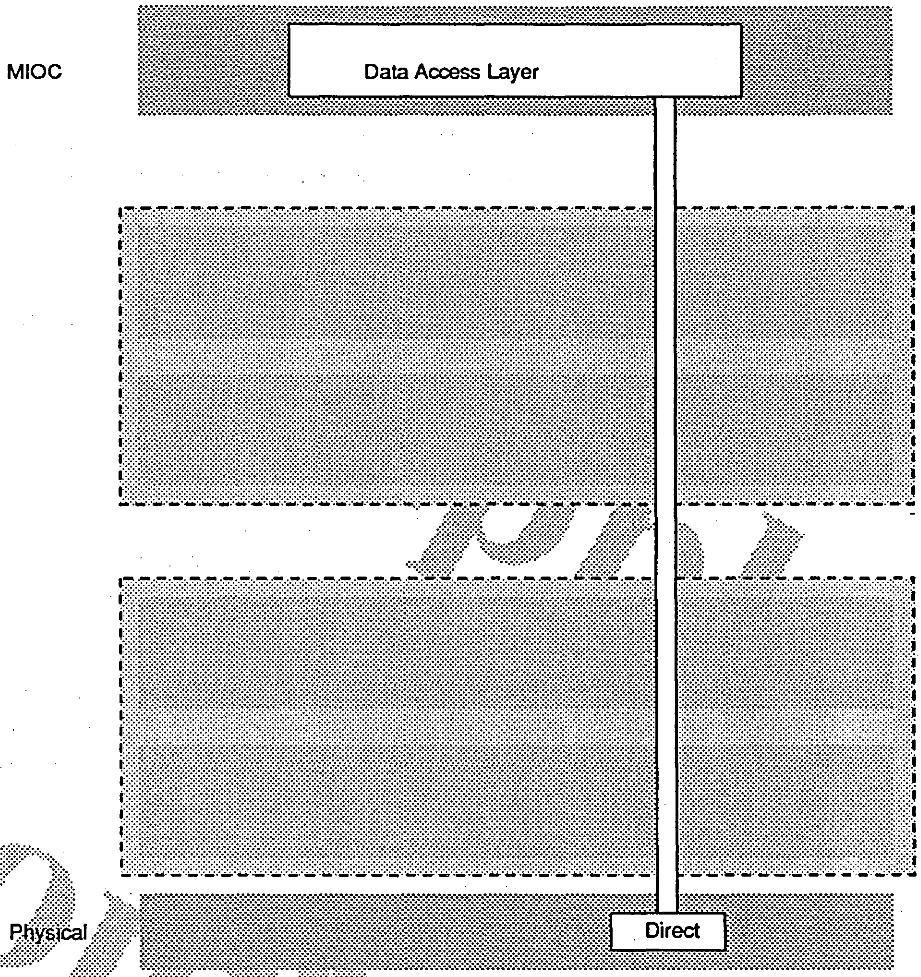


Figure 5-4. Modem layering model

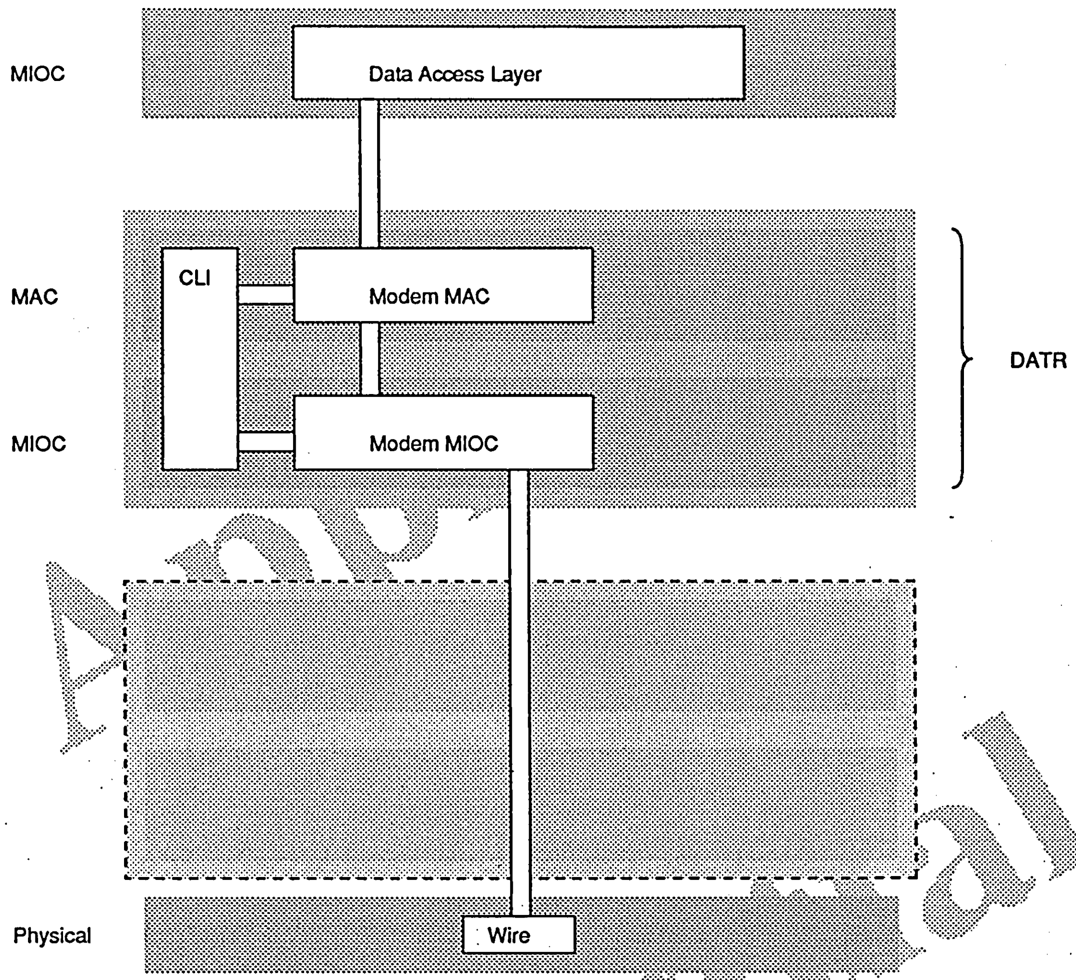


Figure 5-5. Modem layering model

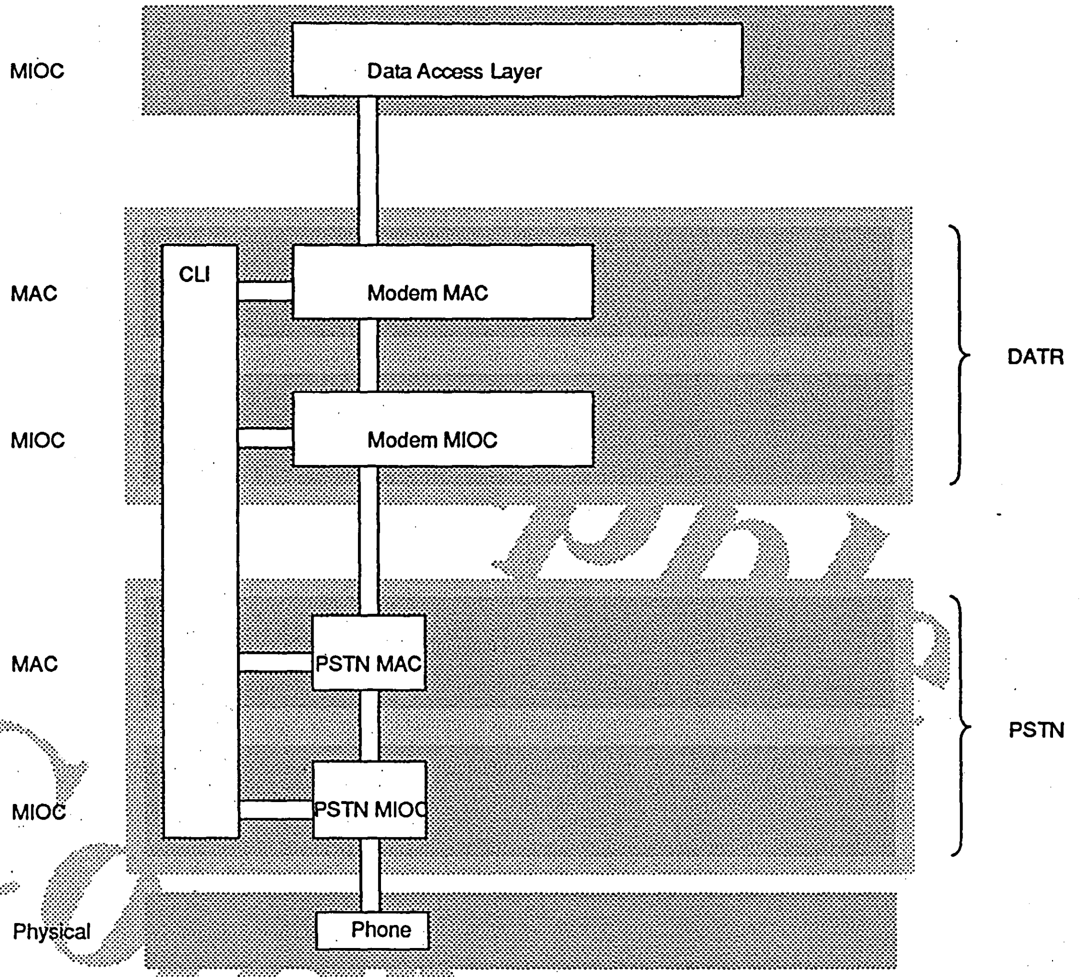


Figure 5-6. Modem layering model

Physical layer Description

The Physical layer is either the Phone system or a simple set of wires. For the Wire connect, it is a one or two wire pair. For the direct connect, it is the number of wires required to do the RS-232 connection.

PSTN MIOC layer Description

The PSTN MIOC Layer performs the basic functions required to communicate with a telephone connection. As pointed out in the Introduction to Arioso, The MIOC layer treats the connection as though it was a one way connection, the device talking to the line. The MAC layer builds upon this service to manage communication with the destination phone device. Thus the MIOC will generate tones, recognize tones, take the handset off hook, and put it back on hook. The generation of tones and recognition of tones is where most of the tight timing constraints in modem implementations come from.

PSTN MAC Layer Description

The PSTN MAC layer uses the tone generation and recognition ability of the PSTN MIOC layer to send a sequence of tones. It assigns meaning to these tone sequences. They may be a dialing sequence, redialing sequence, or terminating if it detects busy tones. It will send and receive tones to establish a connection between the Macintosh and the destination phone device.

DATR MIOC layer Description

After establishment of the physical channel, the modem can use it to send another set of analog signals over the connection. The modem uses these signals for initial negotiation of the data transfer rate, actual transfer of information symbols, clock information, and state detection of the peer modem. As this is a MIOC layer, the signaling does not need a connected modem at the other end of the line.

This connection can either be through the telephone system or simply a direct connection.

DATR MAC Layer Description

The DATR MAC layer manages the connection between two modems. This layer will perform data transfer rate negotiation, establishment and tear down of connections, retraining, and actual bit transfer. This connection appears like an end to end bit stream. If the connection is a synchronous one, then clocking information is also sent across the boundary along with the data.

Data MIOC Layer

The final layer considered in the modem specification is the data MIOC layer. This layer converts the data bytes in a set of buffers into a bit stream suitable for transmission through a modem. The buffers are a set of frames for synchronous transmission, with flags for errors and retries. For asynchronous transmission, the buffers are simply individual data bytes. There is a send and receive buffer for every channel, whether full or half duplex.

Command Line Interpreter (CLI)

The Arioso model is a hierarchical model in which the control flow goes from one layer to the next. Traditionally, modems have a command line interpreter that accepts Hayes Smartmodem commands (the Hayes AT command set). The CLI accepts commands that take the phone on or off hook, for example.

In order to make the document more clear, we show the control section of the CLI as a separate module that connects to all lower layers of the modem.

Physical Layer in Detail

The physical layer is that of the telephone system. It will not be described in this document

PSTN MIOC Layer in Detail

The Datapump Toolbox specification describes access to this and the subsequent layers. Each section will simply show a list of the calls and which category they belong to.

RING_PARAMS()
LINE_PARAMS()
DIAL_PARAMS()
HOOK()
DIAL_MODE()
START_TONE_FILTERS()
GENERATE_TONE()

PSTN MAC Layer in Detail

DIAL_STRING()
V.25()
GET_COUNTRY()

DATR MIOC Layer in Detail

onCarrier()
CONFIG()
IDLE()
FD_SDLC()
HD_SDLC_TX()
HD_SDLC_RX()
FD_SYNC_TRANSP()
HD_TRANSP_TX()
HD_TRANSP_RX()
FD_ASYNC()
FD_ASYNC_NO_RES()
FD_MNP_ASYNC()
LOOPBACK()
OUTPUT_LEVEL()
SPEAKER()

DATR MAC Layer in Detail

DO_RETRAIN()
GET_EQM()
REM_WAKEUP()
GET_VERSION()
MODEM_STATE()
MODEM_CAPS()
HANDSHAKE()

DATA MIOC Layer in Detail

SET_FIFO()
FIFO_BUFFER()
PB_READ()
PB_WRITE()
DO_TCF()
SEND_ATTENTION()

Acknowledgments

Special thanks to Jim Nichols for his help and information.

References

More information on the Modem System of the Macintosh can be found in the following books and articles.

[1] Guide to the Macintosh Family Hardware. Addison-Wesley 1990

[2] Inside Macintosh vol 1-5. Addison-Wesley 1986

[3] Inside Macintosh vol 6. Apple Computer, Inc.

[4] Data-Pump ToolBox 1.0a4

[5] 3615 ERS

[6] The Theory and Practice of Modem Design , John Bingham, Wiley, 1988

Chapter 6

Small Computer Systems Interface

version 1.0

Henry Kannapell

About This Chapter

This chapter describes a proposed SCSI (Small Computer Systems Interface) layering and interface for the SCSI interface of the Macintosh. The chapter begins with an overview and then covers details of the SCSI system. This specification divides SCSI based device drivers into interface layers.

The SCSI described here is a model; the actual calls and interface functions are not necessarily the actual ones that are used in the current SCSI code. This document is trying to serve two purposes - first it is a logical breakdown of the SCSI protocol for Apple implementations, and second it is an example of a format that could describe a layered I/O system on the Macintosh. Those who have read any of the IEEE 802 specifications will see some similarity in the terminology and the format.

Glossary

SCSI uses a number of terms which will be covered in this document. They are defined in this section. The terminology is the same as that used for the Common Access Method subcommittee of the ANSI X3T9 committee.

LUN • Logical Unit Number • This refers to a device within the SCSI device attached to the SCSI cable. In SCSI there may be eight of these for every attached device

I/O Process • This refers to an addressable entity within a LUN. This was introduced as part of SCSI-II. There may be up to 256 I/O processes within any LUN associated with an Initiator

Initiator • The SCSI device that issues a command to another SCSI device

Target • The SCSI device the receives a command

Disconnect • The SCSI target may decide to release the Bus, and still process the command

Reconnect • The SCSI target will rearbitrate for the bus, get the bus and select the Initiator to complete the command. There is only one reconnect outstanding per SCSI target allowed.

CAM • The common access method •A programming interface to a SCSI system designed to be common across multiple platforms.

SIM •SCSI Implementation Module • A "bus manager" for a single bus of the SCSI system defined in the CAM. This is the same as the LLC layer of the Arioso model.

Tagged Queuing•Each device may have a queue of pending commands and requests that need to be operated on. A disk drive can have several pending reads active at once. These requests are placed in a queue, and the queue elements are referenced by integers called Tags.

About SCSI

SCSI is a bidirectional parallel protocol used to communicate between computer peripheral devices. It currently uses an 8 bit bus with a set of control signals for state identification and handshaking. The bus supports multiple masters or "initiators", as well as multiple slaves. The protocol also defines addressable units within each slave.

SCSI transmissions occur over a bus topology with a maximum of 8 devices connected to the media, numbered 0 through 7. Within each device there may be up to 8 Logical Units. These are independent addressable entities. For example, there could be five disk drives, each a logical unit, within a single SCSI device. Each logical unit may have a Queue of 256 elements for each host. These Queues can hold pending SCSI commands that will be executed in order.

Traditionally, the Macintosh has used device ID 7 to refer to the Macintosh itself, and device ID 0 to refer to the internal hard drive. The current implementation of the SCSI Manager supports the Macintosh as an Initiator, and allows multiple Initiators. It does not support the Macintosh as a target.

The SCSI model of interaction has a host (initiator) acquiring the bus, followed by a selection sequence in which the target is selected. Communication is always between two devices. A command is then issued to the target . Messages, which are small (1-3 byte) units, may be sent between the two connected devices to support the connection. Information is transferred between the devices as one of six types: Command, Status, Message In, Message Out, Data In, and Data Out. The direction of transfer is always understood to be from the the point of view of the initiator. The target decides which type of transfer will take place at any given point in the protocol.

The target may *disconnect*; that is, it may remove itself (and the host) from the bus to allow the host to address another target or to allow another target to get on the bus. It may reconnect at a later time by acting as a host and acquiring the bus and then selecting the host who issued the original command. The command will then be completed and a Status byte will be sent, finishing the command sequence. The transaction completes with a final message transfer.

The SCSI protocol is an industry standard now controlled by the ANSI X3T9 Committee. Their current specification is for SCSI-II, which is a successor to SCSI-I, often referred to as just

"SCSI". Most peripherals at this point in time are still SCSI-I. Apple's current implementation is neither SCSI-I or SCSI-II compatible.

One of the goals of the Arioso project is to identify layers that can be implemented by the design centers instead of by system software. For the SCSI system, the most probable split is the MIOC Framing layer. The MIOC and MIOC Framing layers are the primary parts to the system that change with every new CPU, and they are the most hardware dependent. The MiIOC Control and MAC layer and above remain reasonable candidates for system software to maintain.

Implementation Tree

Now let's consider the current Macintosh SCSI software interface. There are two interfaces to SCSI code modules. The first is the SCSI manager. This is basically modelled closely after the 53C80. In fact, the SCSIStat call simply returns two status registers of the 53C80.

A second SCSI interface, known as the "New SCSI Manger" was written to provide an interface at the transport layer. This layer is also basically the same as the SCSI committee CAM (Common Access Method). This is a hardware independent layer that includes many software only processes. This module, written for system 7, never became part of a product.

Two new SCSI managers, the Atomic SCSI Manager for Pink and the SCSI Manager v2.0 for the New Kernel are planned. These are similar to the New SCSI Manager, but have different features and implementations.

The SCSI chips that have been used are the 53C80 and the 53C96. The latter chip will be used in some of the newer CPU designs. In addition, the Curio chip "SuperCombo" from AMD is based on a 53C94, which is substantially the sames as a 53C96.

An implementation tree of the current and planned SCSI designs is shown below. This shows the type of SCSI controller versus the types of CPUs.

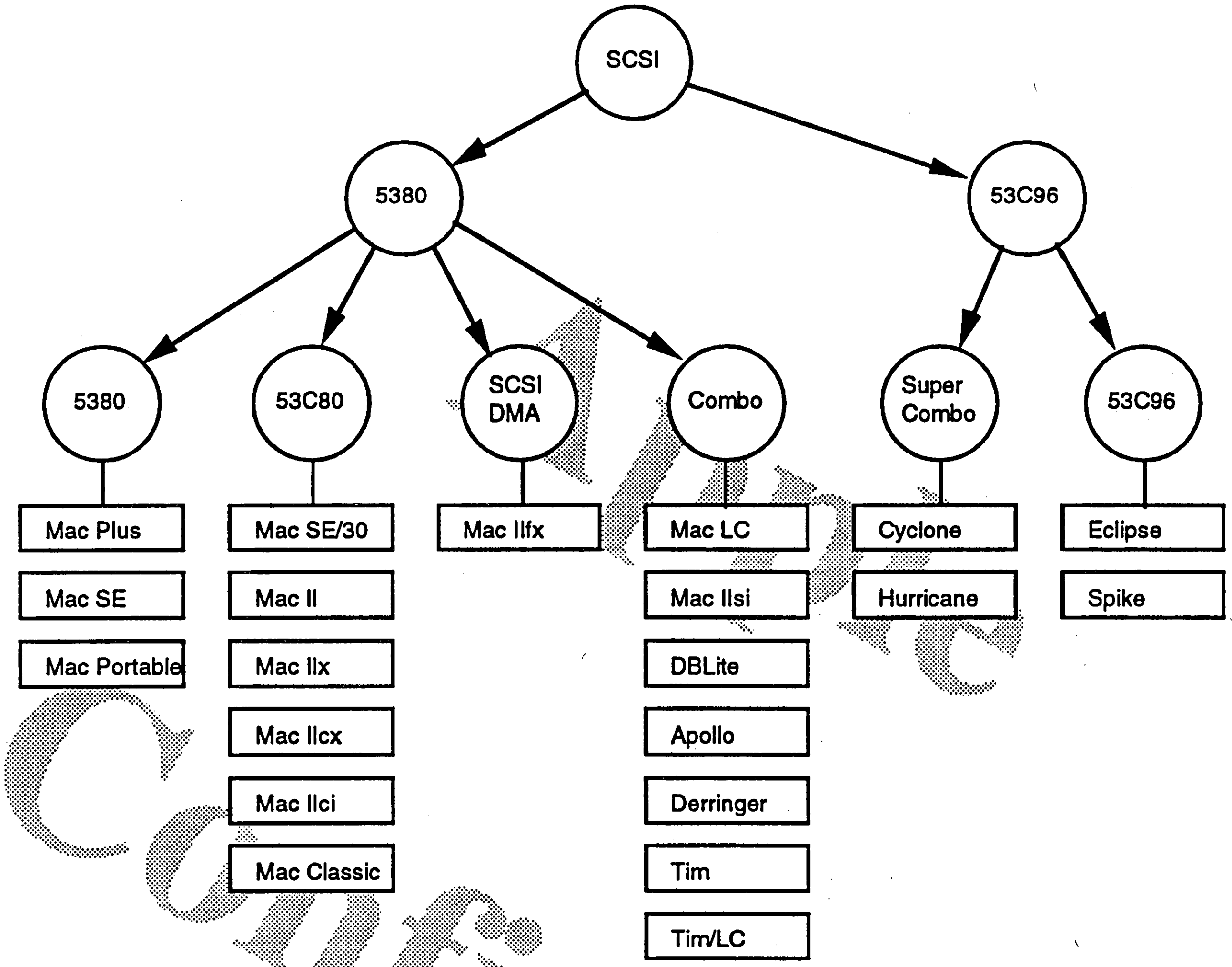


Figure 6-1: SCSI Implementation tree

Arioso Model of SCSI

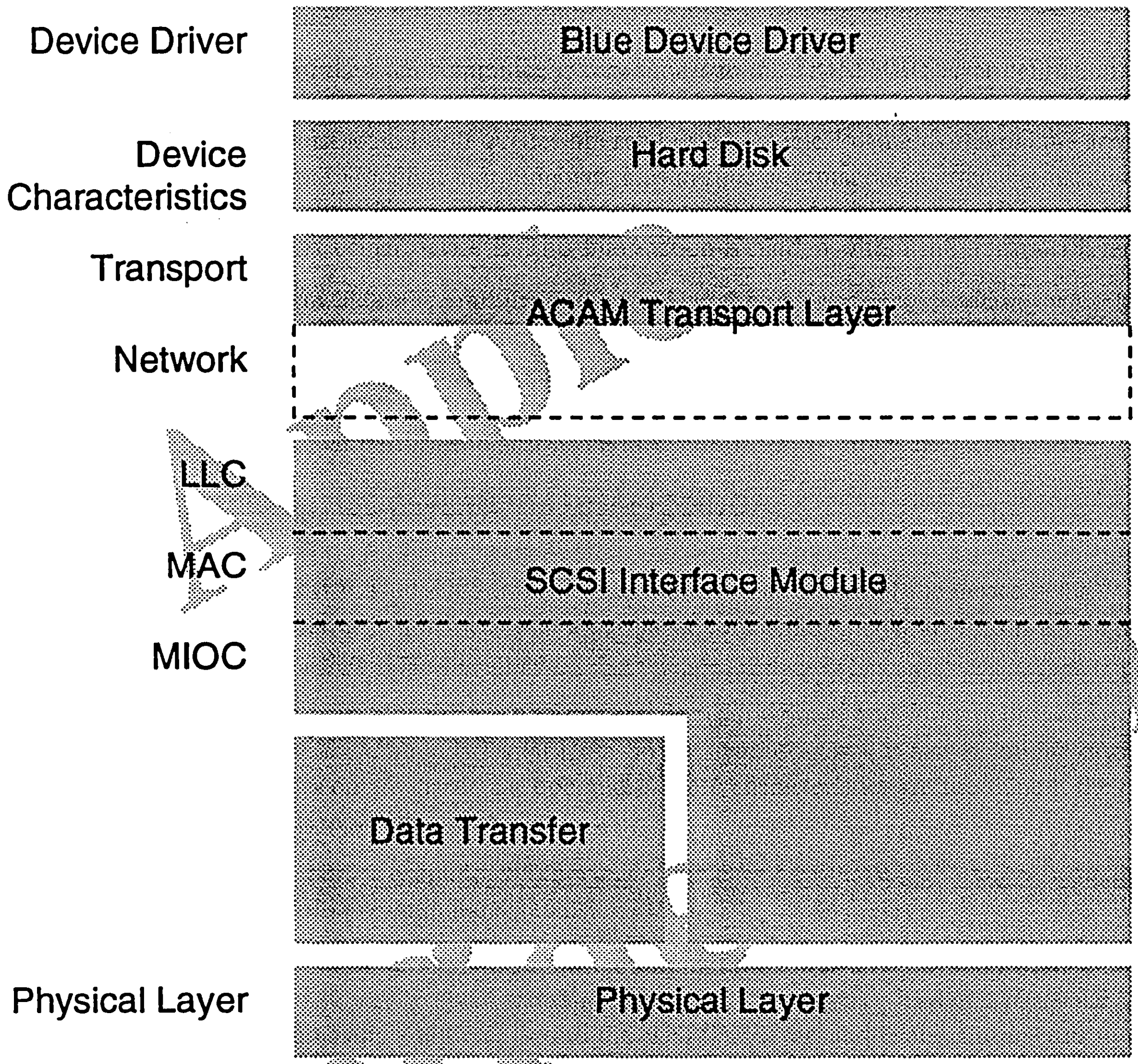


Figure xx. Breakdown of a SCSI Hard Disk device driver

The Arioso layers of the SCSI device drivers are defined in the remainder of this chapter. Each layer has an interface to the layer above it and an interface to the layer below it. The way that a layer is realized is called an *implementation*.

The layers are described from the I/O devices at the bottom, up to the device driver interface at the top. As shown the system has a device driver interface at the top, which then the device characteristics, which represents the actual device. The transport layer below that actually routes the SCSI commands to the final destination. This is implemented as the SCSI CAM layer.

SCSI does not use all of the layers in the model. There is no network layer in SCSI, which would only be useful if one was bridging SCSI commands across network segments.

The lower levels control the communication between the initiator and the target. The Memory transfer section supports segments of physically contiguous data and the byte FIFOs that are part of the final transmission circuit.

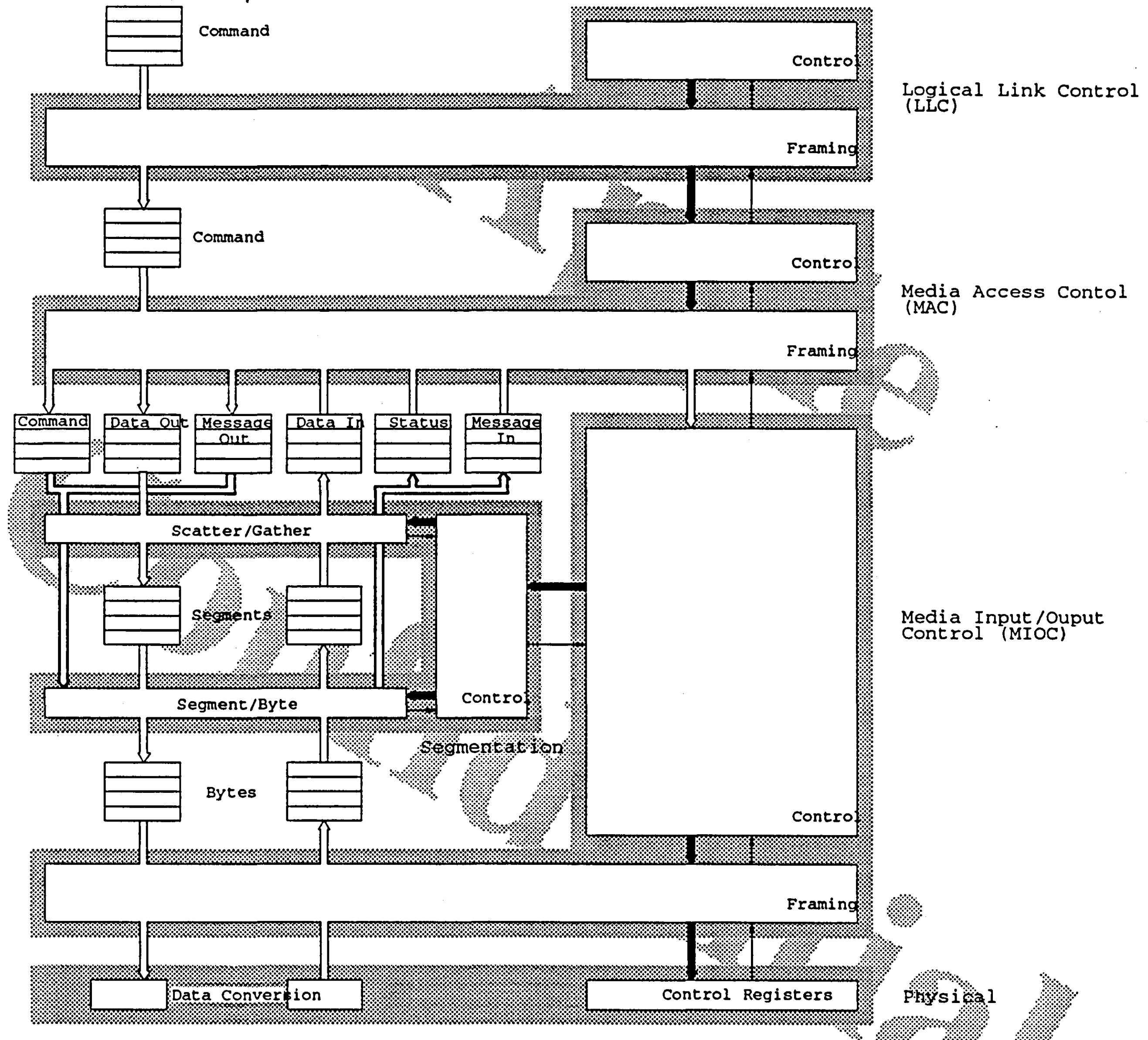


Figure 6-2. Detailed look at the lower SCSI layers

Physical layer Description

At the bottom of the device driver, there is the physical signalling that arbitrates for the bus, selects the target, and transfers information. This information is documented in the SCSI-II draft specification.

MIOC layer Description

The set of commands, messages, and data are given to the MIOC layer for transmission. Since SCSI can transmit the different information units in any order, they are shown as 6 parallel structures. There are Command, Status, Data in, Data out, Message in, and Message out buffers. The MIOC layer thus transmits one of these blocks to the media, depending on which information type the target selects. While there are some complicated rules for what to expect following a certain information type, in general it is simpler to assume that they may be transmitted in any order.

The MIOC also does the equivalent of the SCSIGet() and SCSISelect() calls of the old SCSI manager. These calls are implemented in the MIOC framing level. That is, it will arbitrate for the bus using the SCSI specification's method for getting the bus. After successfully attaining the bus, it will then select a peripheral device that will be the target of the commands.

If the ATN line is asserted when the selection takes place, then the target is instructed to go to Message In phase and acquire the next message from the message queue. This should be an indentify message in SCSI-II, and selects a logical unit in addition to specifying certain characteristics of the connection that will be observed.

The Macintosh may also addressed and selected either to support disconnect/reconnect or when the Macintosh is acting as a target instead of a host. In this case, the command must be routed to the correct higher level entity that will process it before and data transfers begin. The MIOC will then wait for instructions on what information to transfer from the higher layer entity.

The MIOC layer pulls and pushes data from the 6 parallel queues, and either pushes the data into the transmission FIFOs or pulls it from the receive FIFOs of the hardware interface device.

In the Macintosh implementations of SCSI, the equivalent of the MIOC framing level is included in ROM. The rest of the device driver is loaded from the driver partition on the hard drive itself.

In SCSI Mgr 2.0, the MIOC framing layer is called the Low Level Software Interface.

The MIOC layer is responsible for transmitting and receiving information to and from the bus. The information content of the buffers should not be known by the MIOC layer; that is the responsibility of the Mac layer.

Mac Layer Description

The MAC layer sends commands to attached devices, selects which device to process, manages the data transfer, and returns status. It may also initiate messages to be transferred and process messages that the target sends to it. For a SCSI target, the MAC layer receives commands, manages the data transfer, and sends status. It may also initiate message transfer and process incoming messages.

The MAC layer should not know which logical unit number within a device is being addressed. Unfortunately, the addressed logical unit is selected with an identify message. Normally messages are passed between two devices on the SCSI bus; the identify message is a case of the LLC layer sending a message also.

The MAC layer should not know what the content of the command buffer is, or what the status is. His job is to issue the command, generate and respond to any messages that need to be processed, and then return the status. The generating of the command and processing of the status are done by a higher layer.

If a disconnect occurs during a command, the MAC layer will save the state of the connection and switch to the next connection to execute. The saved connection will be saved in anticipation of a later reconnect that will occur that will reinstate the connection and continue from there.

If the Macintosh is selected as a target by another device, the the MAC layer must route the command to the correct higher level entity that will process it. If the Macintosh is reselected, then it will restore the connection state for the target and continue.

LLC (LUN) Layer • SIM Description

The SCSI Logical Unit Numbers (LUNs) are instances of Logical Link Control (LLC) entities. The terms LUN and LLC will be used interchangeably in this document. The SCSI LUN layer is simple, and only affects addressing issues. There may be 8 Logical units and 8 I/O processes per attached device.The LUN layer supports these entities and orders them for transmission in the MAC layer.

Each logical unit can support a queue of I/O processes to execute. They are identified with a 8 bit number with a Queue Tag Message. Since each of the 256 possible queue elements are associated with an initiator, and there are 8 possible logical unit numbers, there may be 256*8*7 = 14336 addressable queue elements on a target. For the purposes of this document, the queued I/O processes are considered part of the LLC layer.

This layer is the one identified in the SCSI CAM committee documents as the SIM (SCSI Implementation Module).

The LLC layer itself only does a routing function and doesn't actually process anything. Thus a real implementation would presumably merge the LLC layer with the MAC layer.

Transport layer • CAM Description

There is currently no network layer in SCSI. There may however be multiple busses within a system. Furthermore there are redundant names between these different busses (i.e., more than one target ID 0). The transport layer performs a routing function between the device drivers and the actual busses. Thus a single programming interface can be provided to the device driver, and the TRANSPORT interface will direct the request to the correct bus and logical unit queue.

The CAM layer of the CAM committee is very similar to the TRANSPORT layer defined in the Arioso I/O model. However, there is one significant difference. The Arioso model addresses each of the distinct logical unit queues of each bus by a logical name that is unique across the entire set of SCSI logical unit numbers. This has the advantage that after the machine is at operation level, there is not need for the device driver to know what target ID or what logical unit number a particular device is. This information is hidden within the transport layer.

The motivation to hide this information is based on the object oriented notion that unless a software module has a need to know a piece of information, is should be hidden from it. Because of the probable industry acceptance of the CAM interface, the weaker CAM method will be used for the TRANSPORT layer interface for SCSI. The Arioso SCSI layer is then a thin layer above this that performs logical name to {Path ID, LUN, Target ID} translation.

Device Characteristics Layer Description

The DEVICE CHARACTERISTICS layer defines the useful attributes of the device being addressed. This layer is not specific to a particular connection media; instead it is a logical representation of the attached device.

An example is a hard disk drive. The drive can be read, written, and formatted; it can be addressed on 512 byte boundaries (at least the ones we use now); it may be offline or online; it can have errors on any particular operation. These characteristics are the same if the device is connected via Appletalk or SCSI.

A tape drive is another example. While many of the characteristics are the same as the hard drive, it can also be rewound and retensioned.

Each of these types of devices are supported through the lower layers of the I/O system. Thus a "read" to a hard disk drive will translate into a SCSI command to read at a particular address. This command will be passed to the SCSI CAM layer, and eventually a status to the command will be returned.

If the same hard drive was attached via a different media, the characteristics of the drive do not change. Instead, the commands and data passed through the media change.

While this layer is close to the operating system device driver interface, it is not the same. For one thing, the several operating systems that Apple supports all use different calling interfaces and conventions, while the device may be the same. For another, some parts of the system device driver interface do functions unrelated to the device, such as switching address spaces, copying parameters from one place to another, and getting the icon of the device.

Device Driver Layer Description

The DEVICE DRIVER Layer will take the device driver interface from the operating system and convert it into requests that map to the common device characteristics layer. No matter which operating system is used, once the file system mapping is done, the requests are simply to read or write different sections of the disk, or to get information about the disk. The task of the DEVICE DRIVER layer is to convert the operating system specific requests to requests to the logical device represented by the device characteristics layer. For simple devices, such as a single hard disk, the logical device will be the same as the physical device, and the DEVICE CHARACTERISTICS layer will be merged with the DEVICE DRIVER layer.

Addressing Structure

The addressing structure of the SCSI I/O system can be viewed as a tree either from the root to the leaves or from the leaves to the root. The difference is in whether one considers physical devices or the logical representations of these devices. Arioso views each layer as representing the devices at the lower layers. Thus it views the I/O system from the leaves to the root, with the actual media connection being the root. For those who wish to view the system as physical devices, the media connection is the root, and each device, LUN and tagged queue element are leaves. The correspondence between these two views is shown in figure xx.

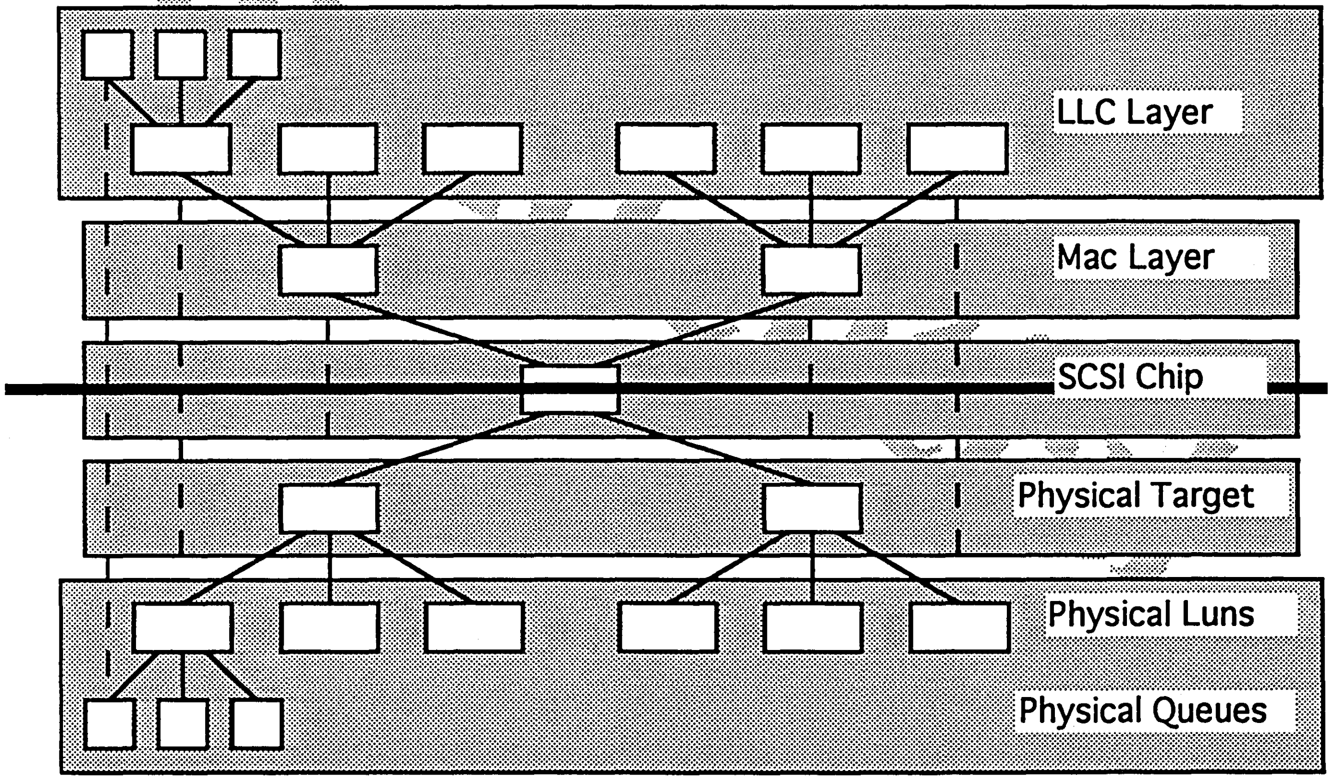


Figure 6-3. Addressing structure of the SCSI I/O System

Addressing in the system is done with a mix of data lines, messages, and commands. The primary addressing is done with the Select bus phase. After this, the identify message can select either one of 8 LUNs or one of 8 "target routines". Finally, if a Queue Tag message is added, the command may be addressed to one of 256 different Queues on a particular LUN.

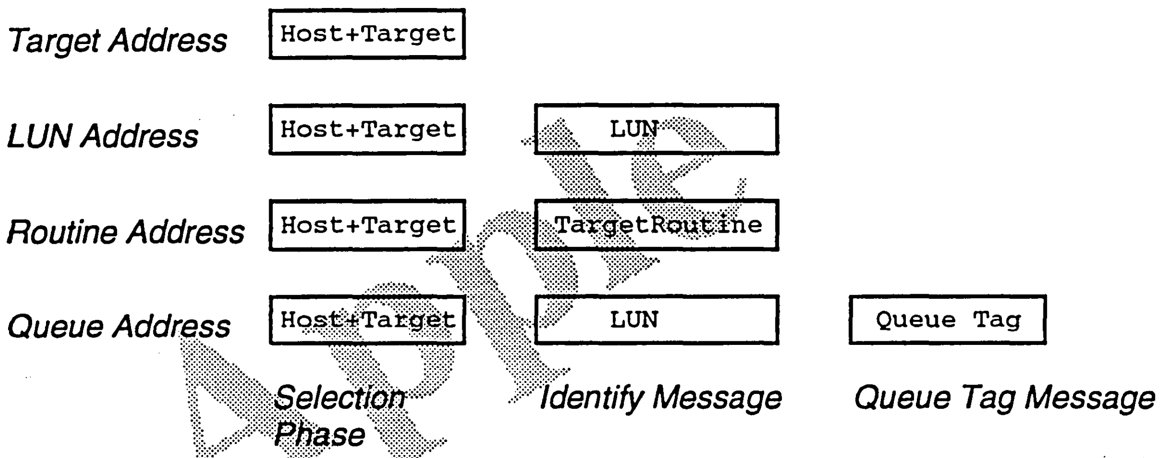


Figure 6-4. Addressing terms of SCSI

The specification describes the SCSI system in three ways. First, the semantics of the main interface calls is the description of what the call does, or what "functionality" it has. This is described at the Operations section of each call and in the written description at the beginning of each section. Since there is no formal method for describing this currently accepted by the industry, this section will be described by several properties of the call. State diagrams may be used to further illustrate the operation of a layer.

Second, the *syntax* of the interface functions specifies what services can be requested and what parameters are used to implement the call. This is described in the call header of each subroutine. The parameters will be shown in set notation to indicate that the order of the parameters and their exact type are not important to this specification. They *are* important to the implementation specification.

Third, the *timing* of each call is the description of how long the call takes minimum and maximum, and what underlying timing is required for the subsystem below it. This is shown in the timing requirements at the end of each layer definition. There are two important aspects to timing. First, what is the time that must be met or guaranteed for the system to work correctly? Second, what is the time that it take a particular implementation to perform a task. The first is a set of constraints that the implementation must meet. The second is a measure of the performance of the system.

Device Access Layer in Detail

The device access layer describes the low level access to the SCSI chips themselves. While many calls might be defined, only the access to the DREQ line is included here.

Device Access Calls

DA.GetDreq() **Request**

Command	get the state of the Dreq line and return it
Return	Return to Caller
Result	{Asserted, Not-Asserted}

MIOC Layer in Detail

1. MIOC Control Layer

The Media Input/Output Control Layer addresses the media and attempts to transfer data bytes out over the media interface, and receive them from the media interface. It is cognizant of the different bus states that take place, but is unaware of the meaning of any of the command or message data bytes. To the MIOC layer, data is just some sequence of bytes of unknown information.

For SCSI, there is only one kind of data sent out to or received from the media. These are sequences of bytes to or from one of six buffers. The current bus phase will determine what the interpretation will be of this kind of data it will be interpreted as. Unfortunately, while the MIOC does not know about the contents of various messages, it must be aware of the bus phase changes. When a bus phase change occurs, data from a new location will begin transmitting. For example, when the SCSI phase changes from Message Out to Command phase, the MIOC will begin transmitting from the Command area.

The SCSI transfers from a host can be broken into three parts.

First, there is the initiator sequence. The initiator on Apple supported drives will arbitrate, select, and then send a command. This is the initiator sequence.

Second, the target controlled sequence will start. In this phase, the target will set the bus phase and request transfers of data. Normally this is data in or out, message in, and status transfers. Also included in this section is the targets change to bus free phase at the completion of the SCSI sequence.

Third, the initiator can send asynchronous messages to the target, or reset the target. It requests a message to be sent by setting the attention line. The Target will then change to Message Out phase as soon as possible, and then transfer the data out.

A SCSI transfer from a Macintosh acting as a target is the reverse of the above sequence.

The following detailed look at the MIOC commands assumes an implementation on a 53C9x class of SCSI chip. If it is implemented on a primative interface, such as the 53C80, each of the calls that affect the 53C9x will be replaced by the equivalent function on a 53C80.

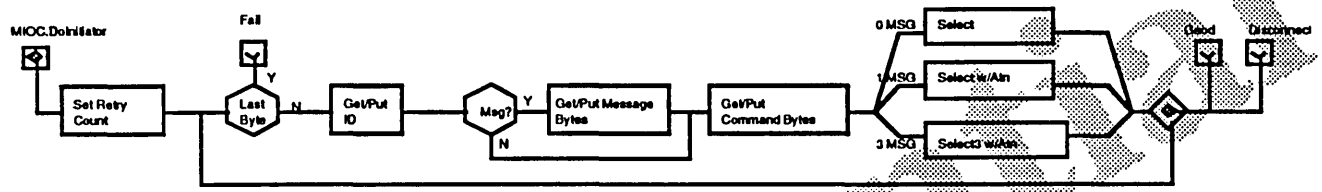


Figure 6-5. DoInitiator graph -53C96

The DoInitiator command will do the first part of the processing of a SCSI transaction when the Macintosh is acting as the initiator. First a retry count is set, to set a limit on the number of times the transaction will be attempted. A timing limit could also be set at this point. It then gets the ID of the target and puts it into the FIFO. It then preloads any message bytes that will be sent by copying them from the message buffer and writing them to the FIFO. Finally it will copy a command from the command buffer into the FIFO.

If the message had 1 byte only, the Select with ATN command is used. If it was three bytes, the Select3 with ATN command is used. This will instruct the 53C96 to begin executing the arbitration, select, and message and command transfers. The Macintosh must wait for some status to return from the chip, which is shown by the block of execution thread symbol. Different implementations may either busy wait, or block, or check other devices during this time. Eventually the 53C96 will respond with either a command successful, in which case the thread moves forward to either the disconnect or the the good state, or an error takes place. Some errors will result in the command being retried again; others will result in a failure being returned to the caller; others may result in the abandonment of this request and a reset of all SCSI devices.

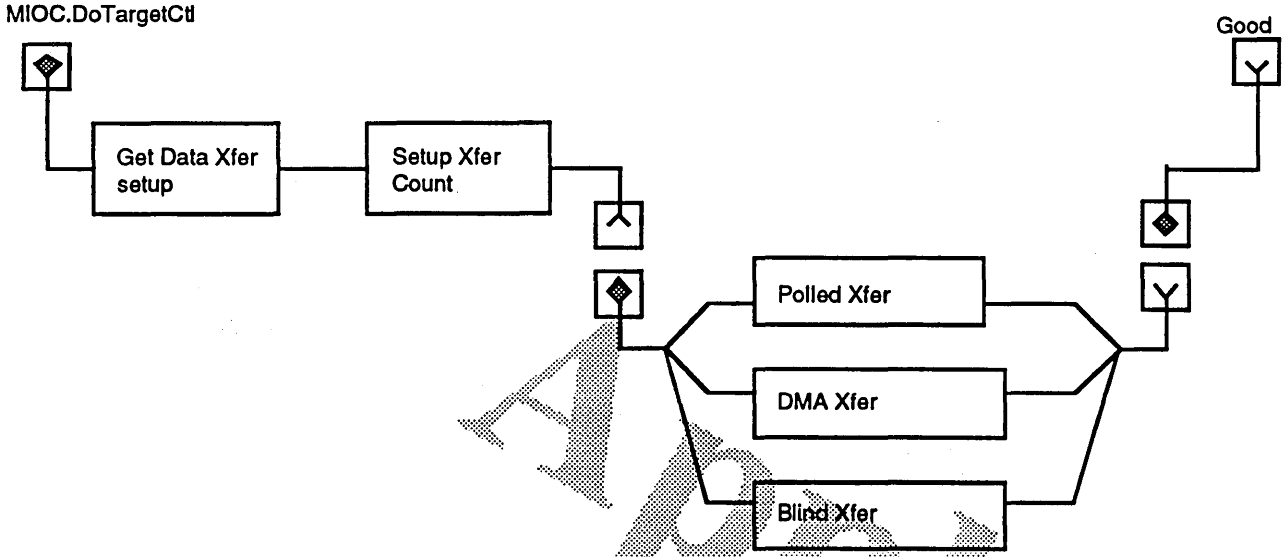


figure 6-6. DoTarget Control graph

DoTargetCtl is the section of the SCSI protocol that is primarily controlled by the target. While the target technically controlled the identify message and the command transmission, since they are always done at the start they are considered part of the initiator controlled section. This call primarily supports data transfer, since any messages that are sent must be forwarded to the Mac layer for processing.

This command begins by setting up the data transfer, since this is the most likely thing to take place next. It must set both the pointers to where the data goes and the count of how many bytes to actually transfer. It then calls the data transfer framing function appropriate for this type of device.

A variety of things can happen at this point. The target can change the phase to the data transfer phase as expected, and the data transfer can take place and either successfully or not successfully complete. The target can change to another phase, such as Message In to send a disconnect or some other type of message. In this case, the data transfer is suspended and the control transfers to the Message In code, which will asynchronously notify the MAC layer that a message has been received and needs to be processed.

Once the data transfer is complete, the function returns a good to the caller.

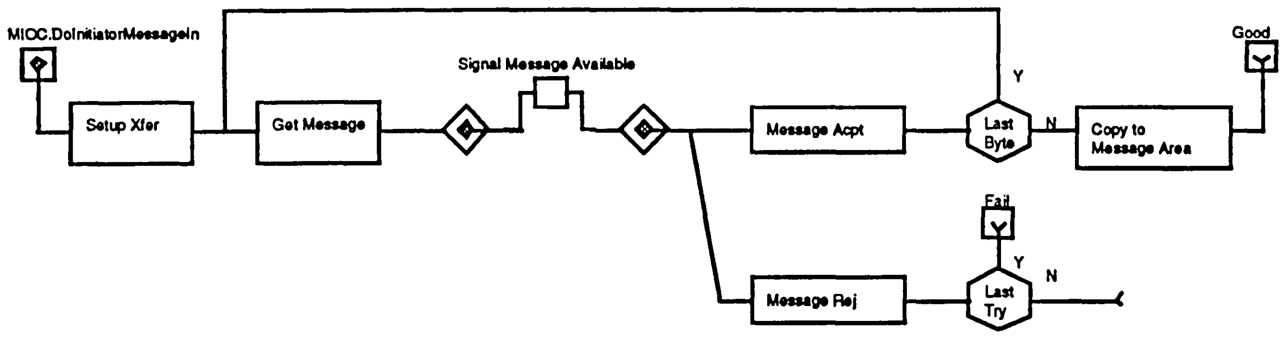


Figure 6-7. DoInitiator-MessageIn graph

This command will process a Message In when the Mac is operating as an initiator. First the setup of the transfer takes place, which involves programming the 53C96 to accept the data. Next, the message in command issued to the 53C96 to actually acquire the message. The thread of execution is blocked while this takes place. Once the message has been received, if it is good, the MAC layer is signaled that a message is available, and the thread of execution blocks again. The MAC layer considers the message and labels it either good or bad. If it is bad, the message is rejected, and the message is sent again. If that was not the last byte, the remaining bytes are acquired. If the retry count on bad messages expires, then the command returns a failure. Otherwise it tries again.

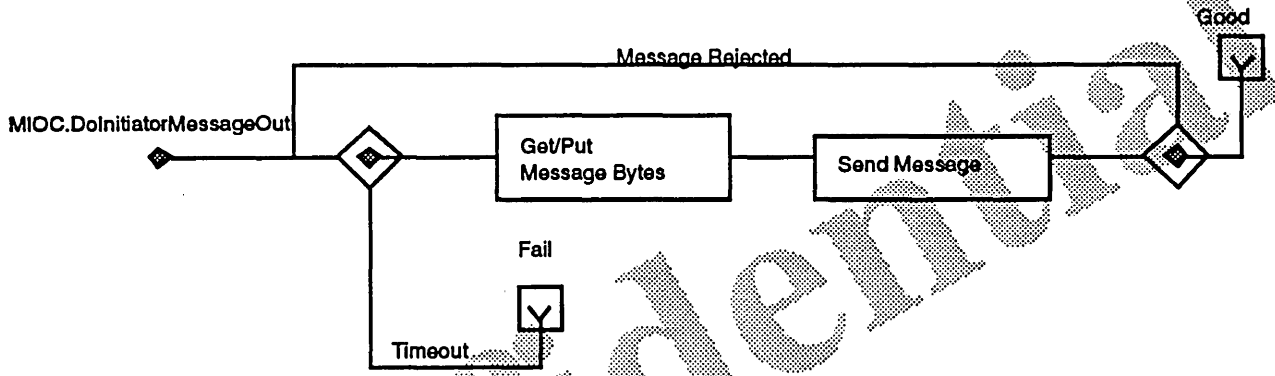


Figure 6-8. DoInitiator-MessageOut graph

The Message out is controlled by a target phase change expecting a message transmission. Thus, this command is not called by the MAC layer; control is transferred internally to this section. This command will process a Message Out when the Mac is operating as an initiator. First the message buffer is checked to see if one is available, since the Message must be created and placed by the MAC layer. The code blocks at this point. It if times out before a message is available to send, it returns a failure to the calling program.

After the MIOC observes that a message is ready to send, it retrieves it from the Message Queue and writes it to the FIFO. It then issues a send message command to the 53C96 to actually send the message. The code blocks again at this point. If the message is rejected, then it is resent. If it is accepted, then control returns to the caller.

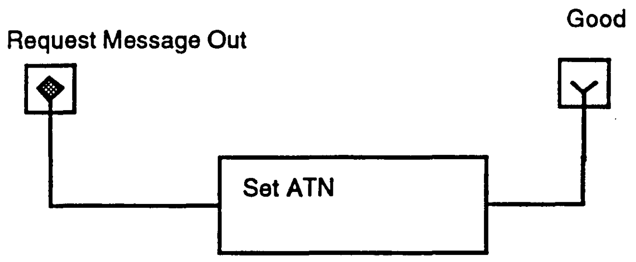


figure 6-9. Request Message Out graph

In the SCSI protocol, only the target may control information transfers. Thus if a host wants to send a message to a target, she must request that the target enter Message Out phase. This is done by asserting the ATN line on the bus, and that is all that this function does.

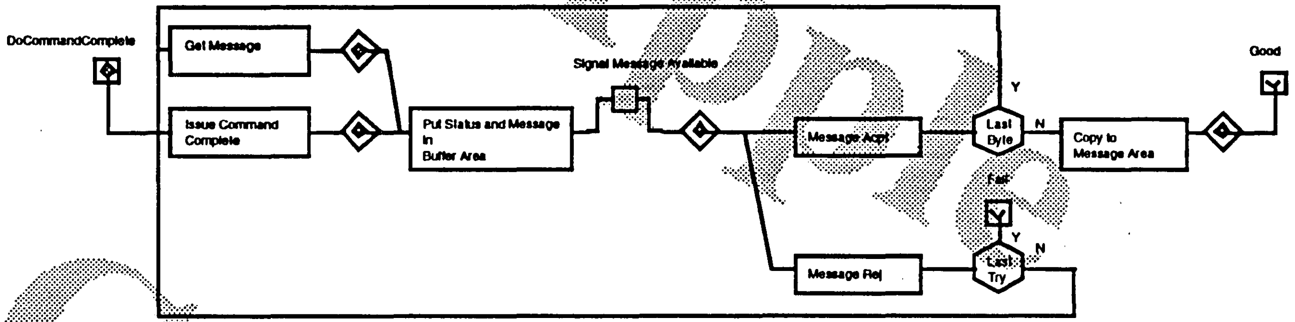


figure 6-10. DoCommandComplete graph

This function is called when the Initiator is ready to conclude a SCSI transaction. This function will prepare to receive Status and Message bytes. After the Mac layer analyzes the message, the MIOC will acknowledge the message and then allow the bus to go to the bus free state.

The command begins by issuing the command complete command to the 53C96. It then blocks the thread of execution until the 53C96 has received the Status and Message bytes. If successful, the Status and Message bytes are put into their respective buffers, and the MAC layer is signalled that a new message has arrived. The execution is blocked again until the MAC layer responds.

If the message is rejected, the ATN line is set and the message accepted command is issued, causing the message to be rejected. If the message is accepted, a check is made as to whether it is the last message or not. If it is the last message, the entire message is copied into the message buffer. The execution then blocks again waiting for a bus free state to be achieved. When that occurs, a response of good is sent to the caller.

If the message is rejected, it is retried a finite number of times, and if still not successful, a failure is sent to the caller.

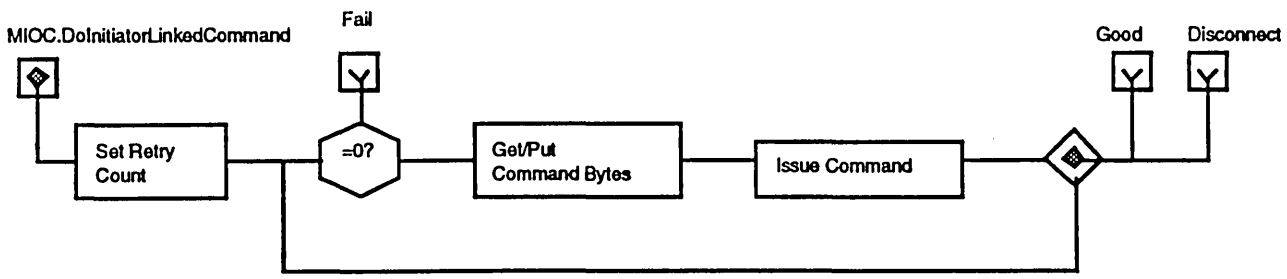


figure 6-11. DoInitiatorLinkedCommand graph

If the previous command was a linked command, the target does not go to bus free at the end of the command. Rather, it will request the next command. It is required that the target issue a Linked command complete message as the final message of the previous command, signalling the Initiator to load the 6 buffers with the new command, data, status and message values.

After the target goes to command phase, this function will be called. First a retry count is set, and possibly a timer. Then the command bytes are copied to the 53C96 FIFO. The transfer command command is then issued to the 53C96, and the execution is blocked. If the 63C96 reports that the command was successful, the good or disconnect response is sent back to the caller. If the transmission fails, then it is retried.

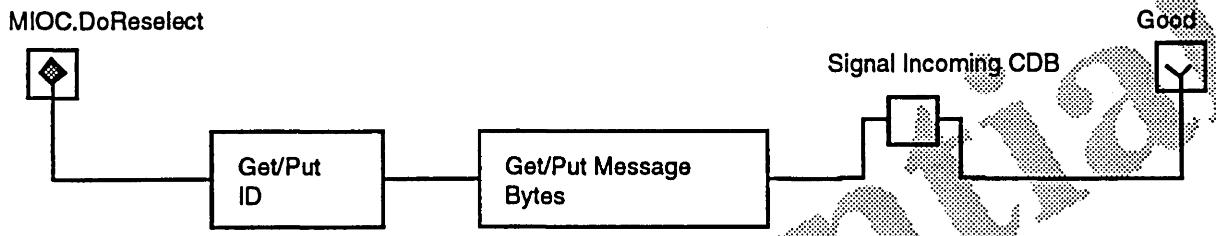


figure 6-12. Do reselect graph

If the Macintosh is reselected by a target wishing to complete a command, it will be notified via an interrupt. This call will then be made to retrieve the ID and the message from the 53C96 FIFO and place them in the buffers.

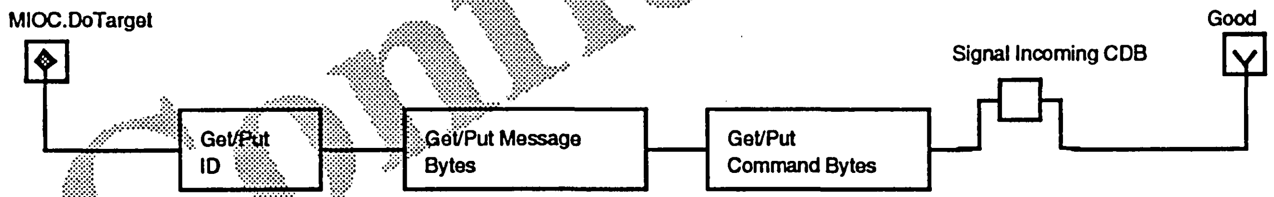


figure 6-13. Do Target graph

If the Mac is acting as a target this call will be made to retrieve the information from the 53C96 FIFO. An interrupt signalled the fact that a selection was made and the Macintosh was the target.

The command will get the ID, the message bytes, and the command bytes, and copy them to the associated buffers. The MAC layer is signalled with an incoming CDB, and then a return of good is made.

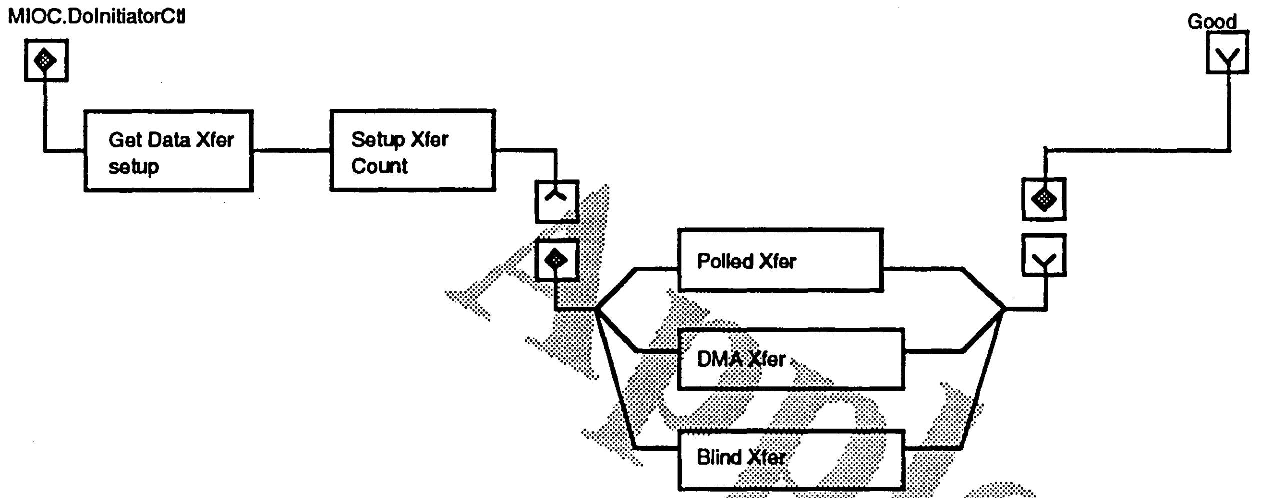


figure 6-14. Do Initiator Control graph

This function will control the data transfer portion of the target sequence. It is essentially the same as the corresponding operation as an Initiator. The difference is that in this command, the Mac is acting as a target and controls all transfers.

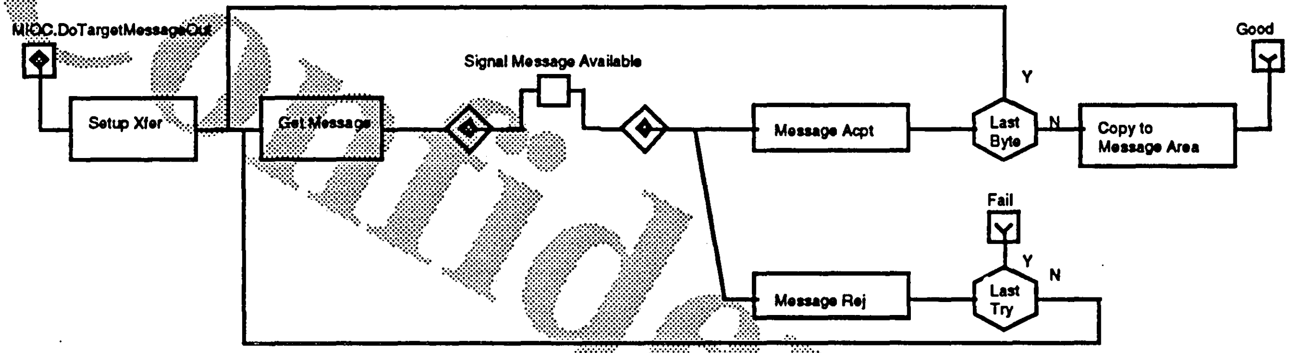


figure 6-15. Do Target Message Out graph

This function will receive a message from the bus. It is essentially the same as the DoInitiator MessageIn command, except that it is the Mac that controls the data transfers.

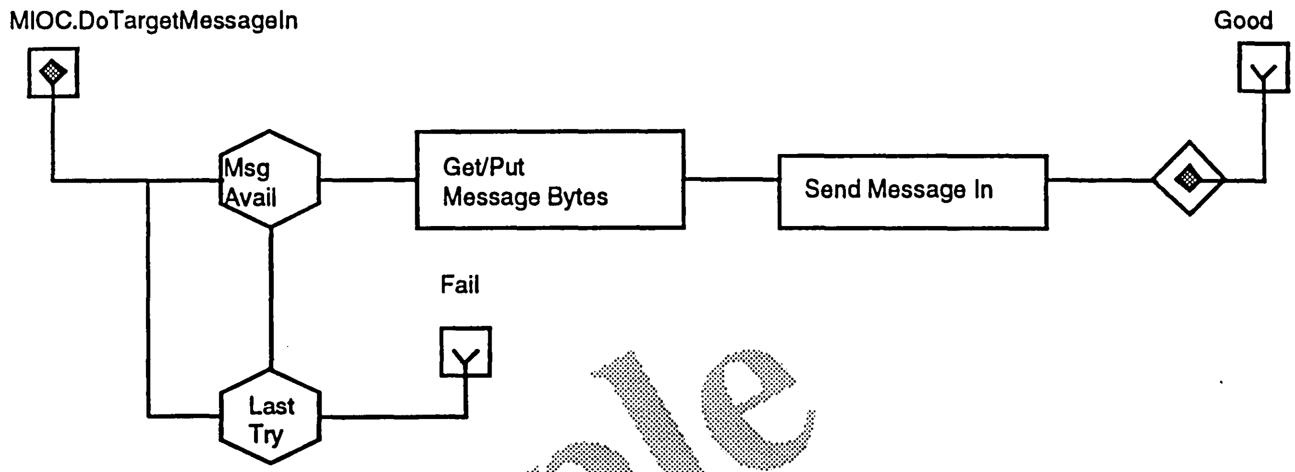


figure 6-16. Do Target Message In graph

This function will send a message to the host. It is essentially the same as the DoInitiator MessageOut command, except that it is the Mac that controls the data transfers.

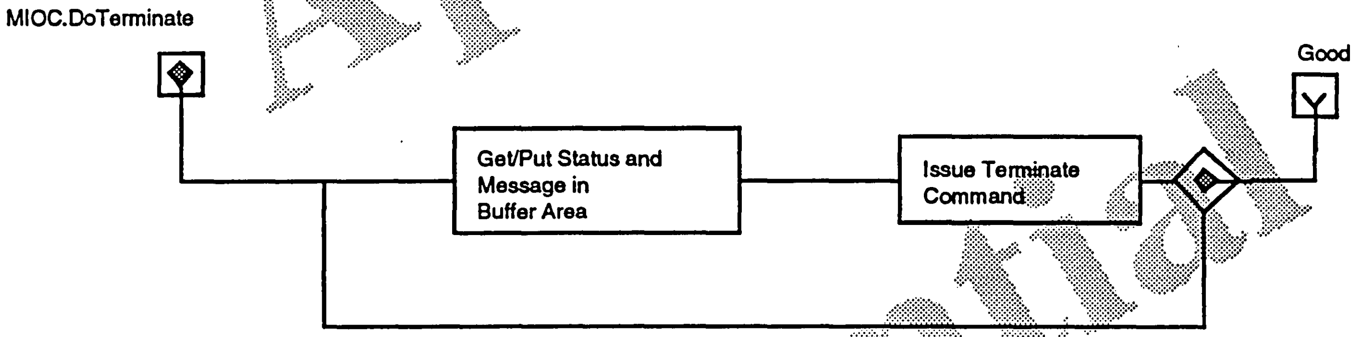


figure 6-17. Do Terminate graph

This function is called when the Mac acting as a target is ready to complete the transaction. It first will put the Status and command complete or linked command completed message in the FIFO of the 53C96 and will then issue the terminate command. After the command is issued, the execution blocks. If the message was accepted, the command returns with a good response. If it was bad for some reason, the status and message are sent again.

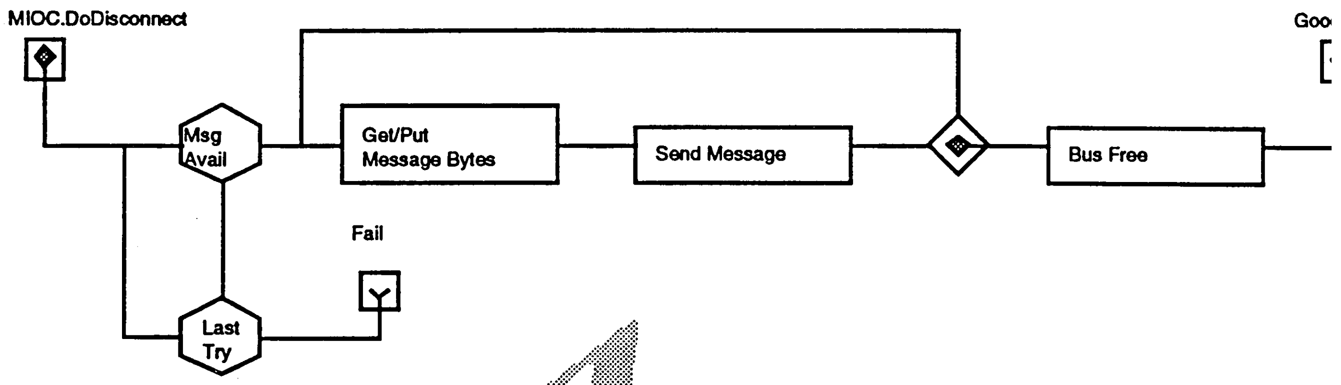


figure 6-18. Do Disconnect graph

When the Mac acting as a target is ready to disconnect to free the bus while it processes a command, this function is called. A disconnect message should be in the MIOC message buffer.

If the message is not available, the command will wait a while, and if it is still not available, a fail will be be reported back to the caller. If one is available, it will be retrieved from the message buffer and put in the 53C96 FIFO. The send message command is then issued, and the execution blocks.

If the message is accepted, the Mac then goes to the bus free state. If it is rejected, then the message is tried again.

MIOC Control Layer Calls

MIOC.DoInitiator([CMDPtr]) — Request

Command	Execute the command with associated messages, status and data.
Return	Return to Caller
Result	{Success, Fail, Disconnect}. There is either success or an unexpected status.

MIOC.DoTarget([CMDPtr]) — Request

Command	Get the command, ID, and messages that were sent
Return	Return to Caller
Result	{Success, Fail}. There is either success or an unexpected status.

MIOC.AbortCommand() — Request

Command	Stop the execution of the current command
Return	Return to Caller
Result	{Success}. The current command is stopped.

MIOC.UnexpectedPhaseChange() — Indication

Indication	An unexpected phase change has occurred that requires attention (normally Message In or Message Out after Command phase).

MIOC.GetSCSIPhase([ID]) — Request

Command	get the current bus phase.
Return	Return to Caller

Result	{BUS-FREE, ARBITRATION, SELECTION, RESELECTION, COMMAND, DATA, STATUS, MESSAGE, UNKNOWN, TRANSITIONAL}.
	The bus can be either in one of the SCSI committee approved states, in an unknown state (since estimating the current state relies on a history of past events), or in an area between states. The SCSI specification does not precisely specify when one state ends and another begins for some of the states, so it is possible to be in between two states.

MIOC.AddMessage([Ptr]): Request

Command	This will add the message to the message queue. As long as there are messages in the queue for the target the ATN line will be asserted.
Return	Return to Caller
Result	{Complete,Error,Timeout,Full} The message may be sent, an error may take place, a Timeout may occur.

MIOC.RequestMessageOut(): Request

Command	This will request that a message be sent out to the connected device. Attention will be set. When the target changes the phase to Message out, the message will be transmitted.
Return	Return to Caller
Result	{Complete}

MIOC.GetMessage([Ptr]): Request

Command	After a message has been received, the bytes can be extracted with the GetMessage command.
Return	Return to Caller
Result	The message bytes.

MIOC.Reselect() Indication

Indication	This Macintosh has been reselected in order to complete a previously started transaction.

MIOC.GetTargetAddress() Request

Command	Get the address of the target. Normally used when reselection is taking place.
Return	Return to Caller
Result	[int 0..7] The address of the current target

2. MIOC Framing Layer

The MIOC Framing layer will take the calls and sequences from the MIOC layer and convert them into actual bus sequences. The MIOC has decided which elements will be done, and then calls the services of the MIOC Framing to actually accomplish them.

Device dependencies affect the interfaces at this layer. In particular, the 53C80 has separate arbitrate and select commands; the 53C9x family does this as one atomic operation.

MIOC Framing Layer Calls

MIOC.F.BusFree([ID]) Request

Command	Release all control signals that are currently being driven on the bus. This may result in a protocol violation occurring.

Return	Return to Caller
Result	Void
Operation	This command will set the following SCSI signals to false: BSY, ACK, ATN, RST, SEL. It then returns. The completion routine happens at the same time as the return.

MIOC.F.ResetBus([ID]) — Request

Command	Initiate the SCSI Reset sequence
Return	Return to Caller
Result	Void
Operation	This command will drive the SCSI reset signal for a minimum of 25 usec. All signals on the bus will go to a non-driven state at the end of this.

MIOC.F.GetSCSIPhase([ID]) — Request

Command	get the current bus phase.
Return	Return to Caller
Result	{BUS-FREE, ARBITRATION, SELECTION, RESELECTION, COMMAND, DATA, STATUS, MESSAGE, UNKNOWN, TRANSITIONAL}.
Operation	The bus can be either in one of the SCSI committee approved states, in an unknown state (since estimating the current state relies on a history of past events), or in an area between states. The SCSI specification does not precisely specify when one state ends and another begins for some of the states, so it is possible to be in between two states. This call will make the best guess of the state given the current information. It is up to the caller to try again if the phase is transitional or unknown.

MIOC.F.StateChange() — Indication

Indication	This indication shows that a state change has occurred. The Mac layer will presumably call MIOC.GetSCSIPhase() to find out what the new state is.

MIOC.F.DoInitiatorMessageOut([Ptr])//SCSIMsgOut() — Request

Command	send a message to the target
Return	Return to Caller
Result	Result interrupt {COMPLETE, TARGET PHASE CHANGE}.
Operation	The message bytes are written into the SCSI chip for transmission. This call is different for a 53C9x and a 53C80. For the 53C9x, the FIFO is filled with the message, and then a NonDMA information transfer command is given.The end of the transfer is indicated by an interrupt, caused either by a phase change or the operation completing or failing.

MIOC.F.DoInitiatorLinkedCommand([Ptr])//SCSICmd() — Request

Command	send a command to the target
Return	Return to Caller
Result	Result interrupt {COMPLETE, DISCONNECT, FAIL}
Operation	The command bytes are written into the SCSI chip for transmission. This call is different for a 53C9x and a 53C80. For the 53C9x, the FIFO is filled with the command, and then a NonDMA information transfer command is given.The end of the transfer is indicated by an interrupt, caused either by a phase change or the operation completing or failing.

MIOC.F.DoInitiatorMessageIn([Ptr])//SCSIMsgIn() — Request

Command	get a message to the target using the FIFO
Return	Return to Caller
Result	Result interrupt {COMPLETE, TARGET PHASE CHANGE}
Operation	Message bytes are received one at a time until the message in phase changes. The command completes when the phase changes to something else. The Information phase is checked between each message byte.If it changes, the next byte is not read. After all of the message bytes have been received, the Message accepted command is issued, and the 53C9x acknowledges the transfer.

MIOC.F.DoCommandComplete([Ptr]) — Request

Command	send a command to the target using the FIFO
Return	Return to Caller
Result	Result interrupt
Operation	get the status byte and message bytes.

MIOC.F.InitiatorWriteData([Ptr]) //SCSIWrite()//SCSIWBlind() — Request

Command	send a data to the target using the FIFO
Return	Return to Caller
Result	Result interrupt
Operation	Send the data currently in the data buffer to the target. There are several implementations of this call; for the highest speed transfers, the SCSIWBlind procedure is used. In this, after the first REQ is detected, the processor will write each data byte with the transfers synchronized by the DREQ line of the SCSI chip. Interrupts are enabled during this time, so it is possible to break out of the transfer loop, and return to it after the interrupt. If the device is too slow, that is if the time between successive REQs is too long, a bus error will be generated that will stop the transfer. When control returns to the transfer loop, the transfer will continue.

MIOC.F.InitiatorReadData([Ptr]) //SCSIRead()//SCSIRBlind() — Request

Command	receive data from the target using the FIFO
Return	Return to Caller
Result	Result interrupt
Operation	Send the data currently in the data buffer to the target. There are several implementations of this call; for the highest speed transfers, the SCSIRBlind procedure is used. In this, after the first REQ is detected, the processor will read each data byte with the transfers synchronized by the DREQ line of the SCSI chip. Interrupts are enabled during this time, so it is possible to break out of the transfer loop, and return to it after the interrupt. If the device is too slow, a bus error will be generated that will stop the transfer. When control returns to the transfer loop, the transfer will continue.

MIOC.F.TargetReadData([Ptr]) — Request

Command	read data from the host using the FIFO
Return	Return to Caller
Result	Result interrupt
Operation	Receive the data into the data buffer from the host. All bytes will be transferred.

MIOC.F.TargetWriteData([Ptr]) — Request

Command	send data to the host using the FIFO
Return	Return to Caller
Result	Result interrupt
Operation	Send the data currently in the data buffer to the target. Since the Target controls the transfer, all of the bytes will be transmitted

MIOC.F.Addressed() — Indication

Indication	Another SCSI device has selected this Macintosh, presumably for reselection

MIOC.F.SetATN() — Request

Command	Set the ATN line to indicate the desire to send a message
Return	Return to Caller
Result	Void

MIOC.F.ResetATN() — Request

Command	Reset the ATN line
Return	Return to Caller
Result	Void

MIOC.F.ArbAndSelect([ID]) **Request**

Command	This command will go through the arbitration and then selection procedure to attempt to address another device. If the message buffer is non empty, the ATN line will be set during selection to force the target to go to message in phase after the selection.
Return	Return to Caller
Result	{Successful, ArbSuccessful-Selection fail, ArbFail, Timeout}. The arbitration and selection either passes, fails, or the attempt times out.

MIOC.F.GetMsgInFIFOStatus() **Request**

Command	This command will check the state of the Message in FIFO
Return	Return to Caller
Result	{Full, Empty, NotFullNotEmpty}

MIOC.F.GetMsgOutFIFOStatus() **Request**

Command	This command will check the state of the Message Out FIFO
Return	Return to Caller
Result	{Full, Empty, NotFullNotEmpty}

MIOC.F.GetDataInFIFOStatus() **Request**

Command	This command will check the state of the Data in FIFO
Return	Return to Caller
Result	{Full, Empty, NotFullNotEmpty}

MIOC.F.GetDataOutFIFOStatus() **Request**

Command	This command will check the state of the Data out FIFO
Return	Return to Caller
Result	{Full, Empty, NotFullNotEmpty}

MIOC.F.GetCommandFIFOStatus() **Request**

Command	This command will check the state of the Command FIFO
Return	Return to Caller
Result	{Full, Empty, NotFullNotEmpty}

MIOC.F.GetStatusFIFOStatus() **Request**

Command	This command will check the state of the Status FIFO
Return	Return to Caller
Result	{Full, Empty, NotFullNotEmpty}

MIOC.F.GetChipFIFOStatus() **Request**

Command	This command will check the state of the chip FIFO
Return	Return to Caller
Result	{Full, Empty, NotFullNotEmpty}

MIOC.F.PutMsgInFIFO() **Request**

Command	This command will put the Message Bytes into the FIFO
Return	Return to Caller
Result	

MIOC.F.GetMsgOutFIFO() **Request**

Command	This command will get message bytes from the FIFO
Return	Return to Caller
Result	Void

MIOC.F.PutDataInFIFO() **Request**

Command	This command will put data into the FIFO
Return	Return to Caller
Result	Void

MIOC.F.GetDataOutFIFO() **Request**

Command	This command will get data from the FIFO
Return	Return to Caller
Result	Void

MIOC.F.GetCommandFIFO() **Request**

Command	This command will get a command from the FIFO
Return	Return to Caller
Result	Void

MIOC.F.PutStatusFIFO() **Request**

Command	This command will put status into the FIFO
Return	Return to Caller
Result	Void

MIOC.F.GetChipFIFO() **Request**

Command	This command will check the state of the chip FIFO
Return	Return to Caller
Result	Void

MIOC.F.PutChipFIFO() **Request**

Command	This command will check the state of the chip FIFO
Return	Return to Caller
Result	Void

MIOC Timing Requirements

Dynamic Characteristics

Symbol	Parameter	Min	Max	Units	Test Conditions
tRLBID	Release Bus max interrupts disabled	--	0	us	
tAASAID	Arb and Select /w ATN max interrupts disabled	--	--	us	
tABF	Last Ack to Bus Free		280	us	† Mac SE only

† The Mac SE start manager requires a disk drive to return to the bus free state in at most 280 us. If not, the boot code will reject the device and not allow the machine to recognize the device [Harlan Andrews].

MAC Layer in Detail

1. MAC Control Layer

This layer is half of the Data Link Layer, and is used in the same sense as the OSI protocols. The MAC Layer is the first layer at which a conversation takes place between the CPU and another peripheral device. Multiple logical connections may exist, but all devices that are part of these connections are directly connected to the media.

The MAC layer can initiate and process message transmissions.

Another way of stating this is to say that there is exactly one address per attached peripheral maintained by the MAC layer. Thus the SCSI driver has a data structure for each of the possible 8 attached devices (including the Macintosh itself).

Each of these connections looks like a distinct entity to the upper layers of the model. In the SCSI example, there appear to be a maximum of 8 different devices to the upper layers of the model. In a sense, the attached peripherals are represented to the upper layers by the operations of the MAC layer. The only characteristics that are represented are those required to maintain a connection and transfer data. The difference between a scanner and a printer, for example, are not visible at this layer.

The commands and statuses are not interpreted at this level; rather, they are formatted and returned to a higher layer. Messages are interpreted and initiated at this level.

There is only one entry point for the calls; thus the LLC layer must decide what order requests get placed into the MAC layer for each logical unit. For each LUN that is associated with a target address, there is a structure in the LLC layer. Requests for each LUN associated with a particular target are ordered by the LLC and passed to the MAC layer.

MAC.DoCommandPacket([TargetID], [CMDPtr]) **Request**

Command	Do the command packet. The command packet here is composed of the command, data, status, but does not have any messages. The messages are entirely handled by the MAC layer. The request is to direct the message to a particular target specified by address.
Return	Return to Caller
Result	{Complete,Error,NoResponse,InvalidID}. The command was either complete or in error. NoResponse means There was no response from the Device. Protocol Timeouts and retries are handled by the MAC layer. Invalid address means that the requested address is outside of the viable range of 0-7.

MAC.GetStatus([TargetID]) **Request**

Command	Get the current status of the target address
Return	Return to Caller
Result	{Waiting,Disconnected,Running,Stopped} The command can be in any one of the states shown.

MAC.Indication([TargetID]) **Indication**

Indication	New Status is available for the given address.

2. MAC Framing Layer

The protocol framing and verification layer is responsible for taking the tasks that the MAC layer wants to do, and actually implementing the functions. Since messages may be exchanged between the initiator and target, there is a command to send and receive a message, independent of the usual identify and final message in of the normal command processing.

MAC.F.SendMessage([Mtype]): **Request**

Command	This will request that a message be sent out to the connected device. The message request is encoded into an Mtype field, which is then expanded into the full number of bytes required to do the operations. The message will be actually sent when the target goes to the Message In state. Mtype includes parameters for each variation of the message that Apple supports. The currently supported messages are: {Identify, Command-Complete, Save-Data-Pointer, Restore-Pointers, Disconnect, Msg-Reject}
Return	Return to Caller
Result	{Complete,Error,Timeout,StateChange} The message may be sent, an error may take place, a Timeout may occur, or the Target may go to another state.

MAC.F.SendCommandPacket([CMDPtr]): **Request**

Command	This will request that a command packet be sent out. The command packet is composed of an identify message, a command, data, status, and message in. The entire set is treated as a unit to be transmitted and received on the SCSI bus.
Return	Return to Caller
Result	{Complete,Error,Timeout,UnexpectedPhaseChange,Received Message, WantMessage} The command may complete, or the Device may select message out state after already sending the identify message, or requesting more bytes to be transmitted than exist, or there may be an unexpected Phase change. An asynchronous message may have been received, or the target may be requesting a message out when none are available. It may also timeout.

MAC.F.GetMessage([MType]): **Request**

Command	Encode and return a message that has been received.
Return	Return to Caller
Result	The current message types that are supported for incoming messages are {Identify, Msg-Reject, Abort}.

MAC.F.Disconnect(): **Indication**

Indication	The Attached Device has disconnected

MAC.F.ProtocolError(): **Indication**

Indication	The Attached Device has stepped through an invalid sequence of states

MAC.F.Reselection() **Indication**

Indication	A device on the bus has reselected the Macintosh to complete a transaction.

MAC.F.Selected() **Indication**

Indication	A device on the bus has Selected the Macintosh as a target

Logical Link Control layer in Detail

1. LLC Control Layer

Each SCSI address can have associated with it up to 8 Logical unit numbers (LUNs). These address a peripheral within the target space. The current SCSI manager does could use this addressing, but it is not normally used. The boot code for the Macintosh always uses LUN 0 when trying to find bootable volumes.

The LLC layer maintains support for each of the LUNs that one might try to address. This layer only supports an addressing function in SCSI. It is a duplication of the calls to the MAC layer, with the LUN addressing added.

The LUNs that are associated with a particular target address operate independently from the LUNs that are associated with any other target address. The LLC manages ordering of requests for any particular target address. The different target addresses can run independently of each other.

This layer is essentially the same as the SCSI Interface Module (SIM) that is specified by the Common Access Method.

LLC.DoCommandPacket([Lun],[Address], [CMDPtr]) **Request**

Command	Do the command packet. The command packet here is composed of the command, data, status, but does not have any messages. There messages are entirely handled by the MAC layer. The request is to direct the message to a particular target and logical unit number specified by address.
Return	Return to Caller
Result	{Complete,Error,NoResponse,InvalidAddress}. The command was either complete or in error. NoResponse means There was no response from the Device. Protocol Timeouts and retries are handled by the MAC layer. Invalid address means that the requested address and LUN is outside of the viable range

LLC.GetStatus([Lun],[Address]) **Request**

Command	Get the current status of the target address
Return	Return to Caller
Result	{Waiting,Disconnected,Running,Stopped} The command can be in any one of the states shown.

LLC.Indication([Lun],[Address])

Indication	New Status is available for the given address.

Transport Layer // SCSI Manager v2.0 // in detail

1. Transport Control Layer

A subsystem may have multiple physical links that implement a class of I/O; there may be two or more SCSI busses. They each possibly have logical unit numbers and device addresses that may be redundant. The transport layer provides an addressing umbrella that gives some type of logical naming to each of the active devices to maintain uniqueness to the Computer. This layer also routes requests between the upper layer software and the destination bus by selecting the correct bus for the destination of the request.

This layer is often responsible for guaranteed end to end delivery of information. All error recovery ends at this layer, and the attempt at communication is deemed either Bad or Good.

Note that this layer is essentially the same as the New SCSI Manager interface. See the SCSI Manager V2.0 from Jon Abilay for more details.

Transport Control Layer Calls

TRANSPORT.SCSIRequestIO([SCSI_PB_PTR]) Request

Command	Do a CAM specified SCSI parameter block. This will route the request to the correct Bus, and execute the function. See the SCSI Manager v2.0 for details.
Return	Return to Caller
Result	{Complete}. Details of the call result can be found in the SCSI Mgr status field.

TRANSPORT.LostDevice([Lun]) Indication

Indication	The Lun is unable to be reached.

TRANSPORT.Open(DevNum,BusNumber, Device Number, LunNumber) Request

Command	Try to open a connection to the device at the BusNumber, Device Number, and LunNumber using DevNum. This will not actually send any commands; it will only create the connections through the intervening layers
Return	Return to Caller
Result	{Complete, NumAlreadyUsed}. Either this was successful or this number is already used.

TRANSPORT.GetStatus() Request

Command	Get the current state of the transport layer
Return	Return to Caller
Result	

TRANSPORT.GetAllAttachedDevices() Request

Command	Get a table of all of the currently attached devices. This can be run at initialization or during operation to poll the attached devices and try to read data from them. This information can then be used for a set of Open calls to connect to the devices.
Return	Return to Caller
Result	{DevTable}. This is a table that has SCSI ID number and Logical Unit Number of all attached devices.

TRANSPORT.AbortIO([DevNum]) Request

Command	Abort the I/O process taking place with the current DevNum.
Return	Return to Caller

TRANSPORT.GetSenseData([DevNum}) **Request**

Command	Get the sense data in the buffer for the attached DevNum. Note that this information is automatically retrieved by the Transport layer whenever a check condition returns.
Return	Return to Caller

2. Transport Framing Layer

This layer only supports one command. This is the Request Sense command. Rather than have the peripheral driver each have to request sense data whenever a check condition occurs, it shall be the responsibility of the Transport layer to request this information whenever a check condition is observed for a command.

The transport layer must have a Sense buffer for each attached device.

Transport Framing Layer Calls

TRANSPORT.RequestSenseData([Lun],[Address]) **Request**

Command	Get the sense data in the buffer for the attached DevNum. Note that this information is automatically retrieved by the Transport layer whenever a check condition returns. This call fills out the command and sends it to the device and awaits the response.
Return	Return to Caller
Result	{[SenseData]} The device specific sense data.

Device Characteristics Layer in Detail

1. Device Characteristics Control Layer (Hard Drives)

All of the previous layers have focused on delivery of data and information between one node and another. Within a device driver, there is usually a logical representation of the actual entity at the other end. Disk drives are assumed to have different characteristics than scanners; modems are different than MIDI synthesizers. These characteristics depend on the device, not on the operating system.

This layer captures the unique elements of any particular device. There is also assumed to be one module that will determine the types of connected devices through the use of the inquiry command to classify each device and return the information to the operating system. It will use the services of the Device Characteristics Framing layer to determine what sort of devices currently exist.

This may be independent of the boot code, in which the operating system uses the services of one of the lower layers to issue a read command to the first block of a device at LUN 0 to see if it has the correct contents for booting.

This section will focus hard disk drives. The Device characteristics level will take device specific commands and will compute geometries and partitions. It will then issue a SCSI command to the logical address of the device, which the transport layer will map to a physical bus, a logical unit number and a device number, and then issue the command.

Partitions are a fiction created by this layer in which multiple interfaces are presented to the next layer, but they all translate into the same device number.

If a disk array was being created that used SCSI as the connection method, this is the section that would have to change to support it. Note that disk array support at this position would allow disks to exist on multiple busses and yet be part of the same disk array.

If a disk used an alternate connection method, such as ethernet or ESDI, there are two possibilities. The first is that a new implementation of this level is done in which the alternate connection commands are emitted from the module.

Another approach is to replace a logical unit at the lower part of the layers with an interpreter and redirector. This will take each SCSI command, determine an equivalent set of commands for the alternate connection, and issue them to the connection. This is more of a translation, where the previous method is a native emulation.

Device Characteristics Control Layer Calls

DC.Format(DevNum) — Request

Command	Format a drive
Return	Return to Caller
Result	{Complete, Error}. Either this was successful or not

DC.Read(DevNum,Address,Size) — Request

Command	Read a portion of the drive
Return	Return to Caller
Result	{Complete, Error}. Either this was successful or not

DC.Write(DevNum,Address,Size) — Request

Command	Read a portion of the drive
Return	Return to Caller
Result	{Complete, Error}. Either this was successful or not

DC.GetStatus(DevNum) — Request

Command	Get the current status of the drive
Return	Return to Caller
Result	{Complete, Error}. Either this was successful or not

DC.GetCapacity(DevNum) — Request

Command	Get the capacity of the attached device
Return	Return to Caller
Result	[DoubleLong] Size of the device in bytes using a 64 bit integer

DC.MediaRemoved([Lun]) — Indication

Indication	The media has been removed from the SCSI device.

DC.MediaInserted([Lun]) — Indication

Indication	The media has been removed from the SCSI device.

DC.UnexpectedDeviceLoss([Lun]) — Indication

Indication	The device has unexpectedly become unable to reach

DC.UnexpectedDeviceAvailable([Lun]) **Indication**

Indication	The device has unexpectedly become available

2. *Device Characteristics Framing*

The device characteristics level will translate the calls to it to a medium dependent set of commands. The Device characteristics framing level will convert these commands to the actual bytes required to be transmitted, and then will call the transport layer to send them. So, for instance, if a new format of a command was created, this is the only area that would be affected.

All currently supported SCSI commands are indentified in this portion.

Device Characteristics Framing Layer Calls

DC.F.Inquiry([DevNum]) **Request**

Command	Issue an inquiry
Return	Return to Caller
Result	Void

DC.F.ModeSense([DevNum]) **Request**

Command	
Return	Return to Caller
Result	Void

DC.F.FormatUnit([DevNum]) **Request**

Command	
Return	Return to Caller
Result	Void

DC.F.Read(6)([DevNum]) **Request**

Command	
Return	Return to Caller
Result	Void

DC.F.Read(10)([DevNum]) **Request**

Command	
Return	Return to Caller
Result	Void

DC.F.ReadCapacity([DevNum]) **Request**

Command	
Return	Return to Caller
Result	Void

DC.F.Release([DevNum]) **Request**

Command	
Return	Return to Caller
Result	Void

DC.F.RequestSense([DevNum]) **Request**

Command	
Return	Return to Caller
Result	Void

DC.F.Reserve([DevNum]) **Request**

Command	
Return	Return to Caller
Result	Void

DC.F.SendDiagnostic([DevNum]) **Request**

Command	
Return	Return to Caller
Result	Void

DC.F.TestUnitReady([DevNum]) **Request**

Command	
Return	Return to Caller
Result	Void

DC.F.Write(6)([DevNum]) **Request**

Command	
Return	Return to Caller
Result	Void

DC.F.Write(10)([DevNum]) **Request**

Command	
Return	Return to Caller
Result	Void

Device Driver in Detail

Apple's current implementation is not SCSI-II compatible. It is mostly SCSI-I compatible, but will not boot off of a device other than device 0 that supports the *Unit Attention Condition* . If the device is in unit attention condition, it will only correctly respond to a INQUIRY or REQUEST SENSE command. Other commands are rejected and returned with a check-condition status.

After a bus reset, devices that support Unit Attention Condition will go into this mode. The ROMs on all of Apple's current machines will issue a READ command for the first access. If this fails (which it will for any device in unit attention), the ROMs will skip the device and go to the next one.

The Startup manager will retest a drive if it is SCSI device 0 (the internal drive).

References

More information on the SCSI System of the Macintosh can be found in the following books and articles.

[1] Guide to the Macintosh Family Hardware. Addison-Wesley 1990

[2] Inside Macintosh vol 1-5. Addison-Wesley 1986

[3] Inside Macintosh vol 6. Apple Computer, Inc.

[4] L. Brett Glass. The SCSI Bus Part 1. *Byte*, pages 267-274, Feb 1990

[5] L. Brett Glass. The SCSI Bus Part 2. *Byte*, pages 267-274, Mar 1990

[6] SCSI Development Package. Apple Computer, 1986

[7] Cheng Lin. SCSI Bus Performance Study. Apple Computer, 1990

[8] Henry Kannapell. Peripheral System Interconnection, PAR Technical Report #8. Apple Computer, 1990

[9] X3T9 committee, Small Computer System Interface - 2, Revision 10-C (draft standard), May 1990

[10] X3T9.2 committee, SCSI - 2 Common Access Method Rev 2.3, Jan 1991

[11] Jon Abilay. SCSI Manager v2.0 ERS Apple Computer, 1991

Chapter 7 Sound Out

Scott Sarnikowski

About This Chapter

This chapter describes the new architecture of the output portion of the Sound manager. Unlike the previous chapters of this document, this chapter describes a high-level organizing structure for several devices - requiring their own Ariso models and chapters. As will become clear in this chapter, Sound Out, rather than being a single device, is best described as a collection of interacting devices.

Glossary

ASIC — Application Specific Integrated Circuit. Apple designed IC usually implement in Gate Array technology.

Co-processor — Auxiliary processor in the Mac architecture that perform various high-level functions with little or no-intervention from the main processor.

Component Manager — Portion of QuickTime that is used to manage and simplify the application interface with sound and video devices.

DSP — Digital Signal Processor. Special type of co-processor that is optimized for performing the mathematical operations need in signal processing.

LocalTalk — Low cost networking protocol shipped in all Macs.

PDS — Processor Direct Slot. Connector that allows devices to be directly integrated into the system on the system bus.

QuickTime — System 7 extension that provides the primitives necessary to maintain the time correlation of video and audio tracks.

Sound Manager — Collections of routines used by applications to generate sound in the Mac.

About Sound Out

Unlike other subsystems discussed by Arioso, Sound Out represents a class of functions. As such any implementation of Sound Out may also represent a class of devices. Figure 7-1 shows the new architecture of Sound Out.

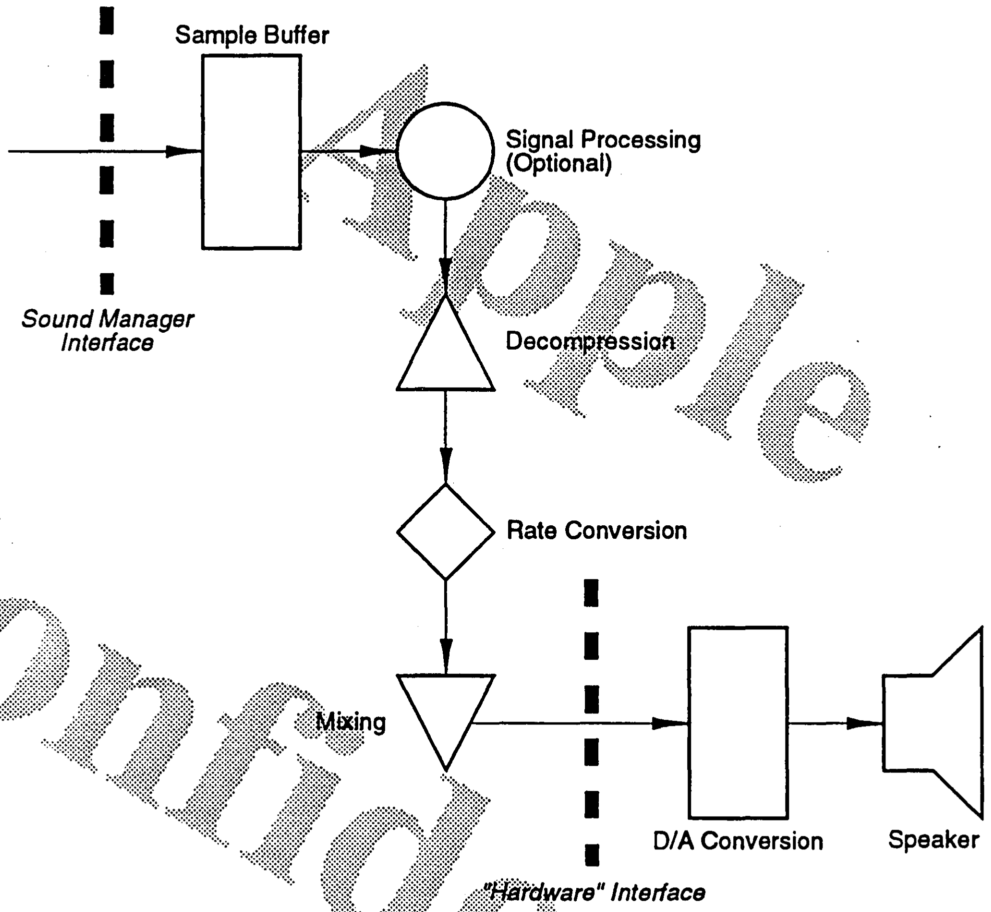


Figure 7-1. New Sound Out Architecture

The notable difference in the new Sound Out architecture is the partitioning of the "stuff" between the Sound Manager Interface and the Hardware Interface. Previous implementations were monolithic and very difficult to maintain. The new implementation has partitioned the manager into four functions: Optional Signal Processing, Decompression, Rate Conversion, and Mixing.

The Optional Signal Processing module is defined to provide general manipulation of the output data. Some examples could be reverberation and other special effects. The next stage is decompression. This stage is used to decompress the data and could use any of a number of algorithms. After decompression, the rate conversion function matches the data sample rate with the output hardware sample rate. The final stage, Mixing, combines multiple channels to create a single output channel. After mixing, the data passes to the hardware section where it is finally heard by the user.

The biggest advantage of this architecture is that any of the functions can be replaced with new software modules or hardware. The implementation provides a versatile and modular approach similar to the

Component Manager of QuickTime. Following the QuickTime model, each module, hardware or software must have a minimal set of capabilities to support administrative functions such as registering its functions with a central source. Each module then has the same interface to ease integration.

Arioso Model of Sound Out

The strengths of the new Architecture discussed in the previous section require that the Arioso model be significantly extended. The Arioso description of an I/O system makes three basic assumptions. The first assumption is that the I/O system contains only a single device type. Although, the network and transport layers provide for addressing many potential devices, the lower levels interface with only a single device. The second assumption is that Arioso is a hierarchical model, requiring that all communications occur between super- and sub-ordinates. Finally, the third assumption of Arioso is that the final or physical layer represents a "point of no return". Once data is written to the physical layer it is lost. In the case of LocalTalk, once the data goes out through the buffers, there is no means to get it back. Unfortunately, Sound Out, taken as a whole, does not fit this description.

Sound Out violates the first assumption of Arioso by providing for multiple devices. It is conceivable that in a single Sound Out implementation that there may be hardware devices for decompression, mixing, and rate conversion. All of these devices may be discrete in nature (i.e., different PDS cards, NuBus cards, or different ASICS), or they may be implemented as algorithms within a single co-processor such as the DSP.

Another way that Sound Out violates the single device model is the notion of processing hardware vs. output hardware. Processing hardware "processes" data and returns to be further processed by other devices. Output hardware is responsible for actually getting data to the speaker. The two are conceptually different, requiring that any model accommodate two types of devices, representing two qualitatively different functions.

Sound Out violates the second assumption of Arioso by the non-hierarchical organization of the four functions and the output functions. The data path shown in Figure 7-1 is misleading in that the four functions of optional signal processing, decompression, rate conversion, and mixing do not really represent a hierarchy in terms of addressing - a basic tenet of the Arioso model. The movement of data through the four functions represents a serial process but not a hierarchical process because each function can use the same references.

The final assumption of Arioso is violated by the nature of the processing hardware. In subsystems such as floppy and LocalTalk, once the data is written to the physical layer it is lost to the media. The definition of processing hardware exceeds this assumption because the hardware implements an intermediate step in the processing flow. There may then be other hardware or software modules that act on the output of the previous module. This requires that the Arioso module account for the processing by sending data out to a physical layer and then return it back to another module.

Given the nature of Sound Out, there are two possible approaches to documenting it with Arioso. the first alternative would be to extend the Arioso model to incorporate the dynamics of the architecture in layers below the Device Driver layer. The advantage of this approach is that the model would capture the entire architecture. But, because of the characteristics mentioned above, consistency would suffer, significantly complicating the description with only marginal payback. Its much better to approach Sound Out as shown in figure 7-2.

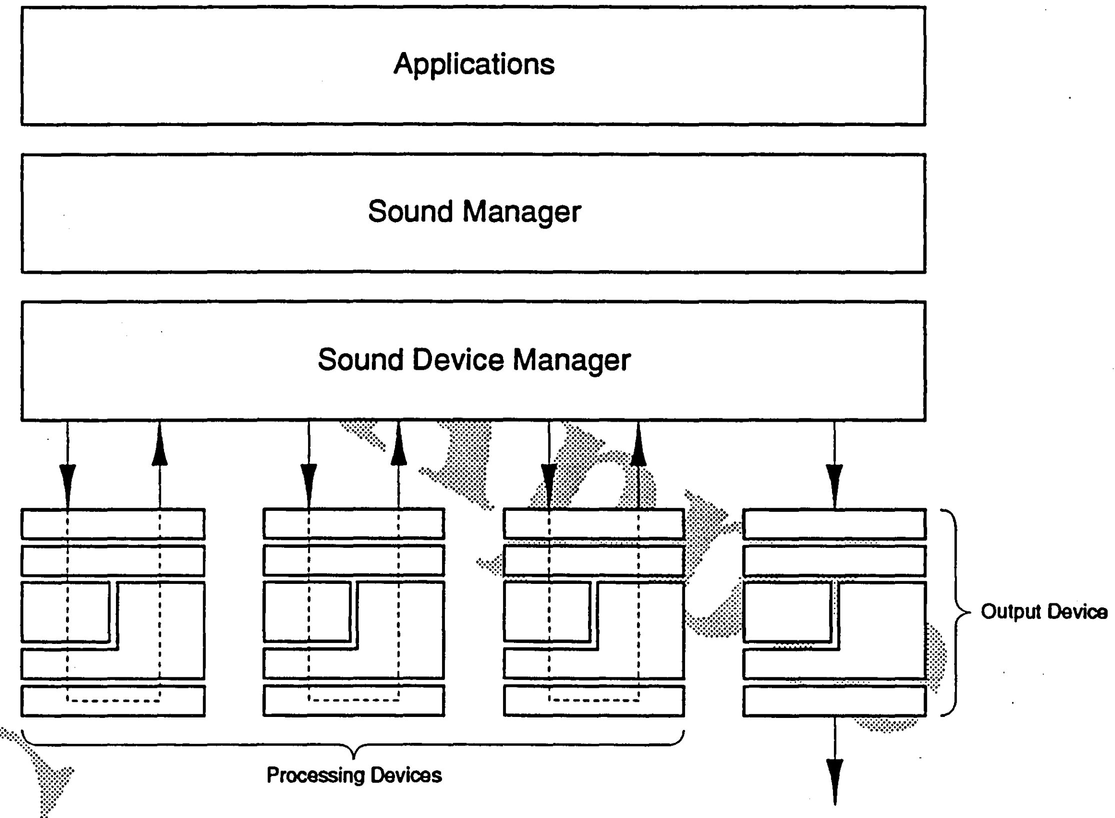


Figure 7-2. Arioso Model for Sound Out

In this model, each device has its own Arioso model, leaving the complexity of orchestrating the various devices to higher levels of description. In this context, devices are defined loosely to include multiple processing and output functions to accommodate co-processors such as DSP.

This approach maintains the spirit of Arioso: "Free up the CPU design groups to make hardware optimizations without incurring a large cost in software development." Thus the Arioso model for Sound concentrates on the individual devices and specifies that at the Device Driver level for each device, they must maintain the same interface. Therefore, Sound Out within the Arioso model is best addressed with separate chapters that cover devices such as decompression, signal processing, mixing, rate conversion, and output.

The rest of this chapter documents the common Device Driver layer of all Sound devices that interface with the Sound Device manager (analogous to the Component Manager of QuickTime). The details of the other layers within each device type should be document under separate, dedicated chapters for each type.

Device Driver Description

The description of the this layer in the Arioso model for a Sound device is modeled after the Component manager of QuickTime. The main calls implement and maintained at this level are the Register.Component, Get.Component.Info, Open.Sound.Device, Close.Sound.Device, and Call.Component.Function calls. The combination of these calls provide the functionality needed to control and move data through the devices.

Device Driver Layer in Detail

Device Driver Layer Calls

`Register.Component` Request

Command	Registers the functions of the sound device so that they can be called by applications.
Return	
Result	{Success, Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`Get.Component.Info` Request

Command	Returns all component information.
Return	
Result	{Success, Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`Open.Sound.Device` Request

Command	Opens the specified sound device.
Return	
Result	{Success, Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`Close.Sound.Device()` Request

Command	Closes the specified sound device.
Return	
Result	{Success, Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

`Call.Component.Function()` Request

Command	Invokes the specified component function. The functions were determined when the component was registered or found with the Get.Component.Info command.
Return	
Result	{Success, Failed} Result reflects either success or an unexpected status. It is expected that a failure of this command is extraordinary.

Device Driver Timing Requirements

Because of the high-level description, necessitated by the nature of Sound Out, there are no timing requirement specified for this layer. All timing requirements must be tailored to the individual devices.

References

QuickTime Developer's Guide, Apple Computer Inc. - Draft, 1991

Inside Macintosh, Volume VI, Apple Computer Inc., 1991

Guide to the Macintosh Family Hardware, Apple Computer Inc., 1990