1-5-1987

An image-capable interprocessor link communications protocol

Robert W. Brown

AN IMAGE-CAPABLE INTERPROCESSOR LINK COMMUNICATIONS PROTOCOL

BY

ROBERT W. BROWN

January 5, 1987

A thesis submitted to the faculty of the School of Computer Science and Technology, at Rochester Institute of Technology, in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

APPROVED BY:

James E. Heliotis _____ (Committee Chairman)

James E. Heliotis

John L. Ellis _____ (Committee Member)

John L. Ellis

Donald L. Kreher _____ (Committee Member)

Donald L. Kreher

Based on his original research and written by:

Robert W. Brown
Commercial and Information Systems Group
Eastman Kodak Company

in cooperation with:

School of Computer Science and Technology
Rochester Institute of Technology

ABSTRACT

Image-capable interprocessor links often require the use of specialized communications protocols due to the large amounts of data that can be transmitted and the high speeds at which the data can be transferred. Without a set of generalized image-capable communications rules, these protocols are commonly customized to each interprocessor link application. The image-capable interprocessor link communications protocol described in this paper (IPLIMP) has been designed to limit the need for such customization at a low level. In order to accomplish this, IPLIMP provides simple methods for synchronizing and controlling link operations, sending and receiving data, downloading additional protocol layers, and communicating with higher level services. By providing this basic set of functions with minimized overhead, IPLIMP can be used as a fundamental building block for numerous image-capable interprocessor link applications. In this manner, a common basis for image communications is developed, thereby ensuring at least a low level degree of compatibility.

ACKNOWLEDGEMENTS

Peter R. Puccini:

Project suggestions, development, and support.

David M. Schond:

Hardware modifications to DRV11-B boards for initial prototyping.

James R. Forger:

SBC-11/21 (FALCON) protocol implementation, development, and debugging.

Kodak Research Library Staff:

Research materials, literature, and investigation.

TABLE OF CONTENTS

1 INTRODUCTION TO IMAGE-CAPABLE COMMUNICATIONS

The concern for how image information is transferred from place to place has grown with the increasing importance of image processing. Disciplines such as astronomy, defense systems technology, graphics, geophysics, medical and X-ray technology, and photographic science, rely upon the ability to communicate digital image data effectively. Despite the increased focus on image-capable communications, there has hitherto been little progress toward the development of a basic set of image-capable communications rules. Without a set of generalized communications guidelines, image-capable interfaces have had to be customized [1]. This customization poses problems for systems development and integration. For instance, the effort needed to develop one image-capable interface may have to be duplicated during the subsequent development of similar interfaces. Also, systems that have been designed independently are often difficult to integrate together, although they may be functionally compatible, due to incompatibilities between their communications protocols. Hence, customized interface designs may be quite useless for developing additional interface applications or integrating with similar systems.

Before describing some of the preceding customized approaches to developing image-capable interfaces, a brief introduction to this area of communications is needed. In order to provide the appropriate background, the first sec-

tion of this chapter presents interfaces in terms of interprocessor links and protocols. Next, image-capable link requirements are covered in section 1.2. Section 1.3 then introduces image-capable protocols and the customization problems from a historical perspective. In chapter 2, a general purpose image-capable interprocessor link communications protocol is presented as a possible solution to the problems associated with protocol customization. Finally, chapter 3 describes actual implementations of the generalized protocol in support of image-capable interprocessor communications.

1.1 INTERPROCESSOR LINKS AND PROTOCOLS

Whenever two processors are interfaced together, an interprocessor link is formed that loosely couples the systems by providing a mechanism for the transfer of data and control information. Typically, interprocessor links are constructed by connecting two or more systems to some type of communications medium through system interface modules, which operate under the independent control of their host processors. Though the processors are interfaced together, their system independence requires the implementation of a governing set of rules and conventions to ensure reliable communications over the link. Such a set of procedures and formats, which are mutually agreed upon for the purpose of providing communications, are collectively referred to as a protocol [14]. Often, in order to increase flexibility and

simplify design, protocols are layered together. Regardless of how control is organized, protocols, which apply to both the hardware interface and the controlling software, must be designed to maintain communications in support of the speed, efficiency, and service reqirements of the link.

For example, protocols are sometimes required to include downloading services for transferring and controlling software modules across a link. Such functionality is necessary for those systems that are incapable of archiving additional protocol layers or applications. Without download protocols, these systems would not be able to function. Unfortunately, as is the case with most link services, downloading results in additional communications overhead. This overhead originates from the extra control messages and/or header bits needed to distinguish the data and control associated with a particular service from the other information being sent across the link. Therefore, as link services are added, overhead increases, which may adversely impact system performance and prevent the fulfillment of all link requirements.

1.2 IMAGE-CAPABLE LINK REQUIREMENTS

As shown in table 1-1, there are three main requirements for supporting image-capable interprocessor link communications [9,12]. First, the number of records transmitted per image must be accommodated. This number can be

quite large, as an image may consist of a consecutive stream of thousands of equally sized records. Second, the size of image records must be handled. Again, this can be a significant amount, since each image record may be on the order of thousands of bytes in size. Third, the speed at which image data is to be transferred must be supported. While transfer speeds are typically on the order of millions of bits per second, rates can be obtained that are thousands of times higher. Often these rates are sustained by some real time device, such as an image scanner that continually produces a stream of unbuffered data, or a drum printer that requires continuous input of digital data. Thus, an image-capable interprocessor link can potentially be required to support the transmission of millions of bits of image information at some real time data rate.

IMAGE-CAPABLE COMMUNICATIONS REQUIREMENTS

1. LARGE NUMBER RECORDS - Thousands of records per image
2. LARGE RECORD SIZE - Thousands of bytes per record
3. REAL TIME SPEED - Millions of bits per second

Table 1-1

Along with these three main requirements, an image-capable interprocessor link is also typically required to support any of a number of link services. Such services in-

clude remote image access, downloading, image security, compression, and a variety of other functions. Though various systems have successfully fulfilled the image-capable communications and service requirements, they have not yet implemented a generalized interprocessor link protocol for image communications. In fact, most imaging equipment has been customized in order to meet specific device requirements and functionality [1]. Such customization is not only wasteful, but it hinders the transportability of devices and the interconnectability of systems [1,9].

1.3 HISTORICAL PERSPECTIVES

Over the last ten years, engineers at Eastman Kodak Company have successfully implemented numerous image-capable interfaces [2,3,4,5,6,16]. Oftentimes, these interfaces have been constructed using customized data channels, electrical circuitry, and control software. The Dicomed D47 Image Recorder, the Dicomed D48 Image Recorder, the Optronics Colormation C-4500, and several custom-made devices, have been interfaced to computers at Kodak. Though image communications were realized, the inherent interface customization has resulted in a great deal of unnecessary effort to be expended in repeatedly performing similar work.

In 1984, Elms, Nelson, and Rothlauf (Picker International) developed an image transfer interface standard in an attempt to ease the effort required to implement image

communications systems [9]. The imaging equipment interface they described consisted of a serial RS-232-C control channel and a high speed 16-bit parallel direct memory access (DMA) data channel for connecting any imaging equipment device to a network interface. Details of their protocol standard included 4096-byte fixed length data records, a specified 32 Mbps data capacity, ASCII control commands and responses, control channel protocol, data channel protocol, electrical layout, and an image header specification. Indeed, this image transfer interface standard addressed many low level protocol requirements for image communications.

Also in 1984, Philips Medical Systems publicized a proposed standard product interface for digital medical imaging equipment [1]. This proposed interface consisted of three protocol layers designed to connect imaging equipment together or to networks. Physical and data link layers were organized to provide combined support for reliable network-independent communications over an interprocessor link. Above these two lower layers, a communications package layer was positioned to act as an interface to imaging devices or to client systems. Basic functions, including initialization, resource management, communication information presentation, and message transmission and reception, were specified to promote the exchange of image information. These communications package functions were designed to handle messages in a standard format consisting of (1) a fixed communications section for indicating message priority, size,

source, and destination; (2) a fixed command section for specifying a unique object and action, which give meaning to the data and instructions for its use; and (3) a variable data section for conveying user information. In this manner, the proposed standard product interface provided a general mechanism for connecting imaging equipment.

Similarly, in 1985, the ACR-NEMA Digital Imaging and Communication Standard was developed as a layered architecture for connecting imaging equipment together and to networks [11]. A physical layer was specified, including cable pinouts for 16-bit parallel data transfer and asynchronous control. Though the hardware was targeted to produce only 17 errant bits per billion transferred at 8 MBps over a cable length of 15 meters, other physical systems were permitted. Flow control and error checking were handled by a data link-media access layer, which also encapsulated each data block with frame number, sequence check, and frame descriptor words. Above this layer, a network/transport layer divided messages into data blocks consisting of a sequence number word, a block descriptor word, and a maximum of 2048 words of data. A fourth layer, the session layer, handled primitive requests, responses, and commands to support the higher level presentation and application layers. As in the previously discussed standards, the connection of imaging equipment was readily facilitated by this protocol.

Though any of the above mentioned protocol standards would certainly help reduce the time needed to interface new

equipment, they are all too restrictive to be widely used as a basis for image communication. By placing constraints on physical connections, the Picker International and ACR-NEMA standards exclude numerous image-capable interfaces. Also, both of these standards limit their flexibility by restricting the size of records transmitted across an interprocessor link. By fixing or maximizing record sizes, these standards cannot accommodate all systems requiring an integral number of image lines per record, or systems that cannot tolerate the reassembly time needed to reconstruct image sections. Although the Philips standard does not have these restrictions, it does require the addition of control information to each record. As in the ACR-NEMA standard, the Philips standard adds overhead to each record, which may cause the protocol to fail to meet speed and throughput requirements. Furthermore, the added control information may not even be necessary in many imaging systems. Finally, none of these standards specifically address the problem of initiating and controlling operations on various types of imaging systems. Hence, what might be more useful is a less restrictive, more flexible, image-capable protocol that can keep device interfaces regular, eliminate customization of control software, and support interprocessor communications at a low level.

2 A GENERAL PROTOCOL FOR IMAGE-CAPABLE INTERPROCESSOR LINKS

The following material presents a general purpose, flexible, image-capable interprocessor link communications protocol that may be used as a foundation for image-capable communications. This protocol, hence forth known as IPLIMP, an acronym for Interprocessor Link Image Mode Protocol, was designed to limit the customization needed to implement image communications systems. In attempting to do so, IPLIMP does not provide routing, flow control, process naming, or other networking functions. Instead, IPLIMP is limited in scope to the low level services required by interfaces desiring image-capable communications. As such, IPLIMP is concerned with the control and operation of a particular set of image-capable interprocessor link architectures.

2.1 CHARACTERISTICS

In order to serve as a foundation for image-capable communications, IPLIMP was designed to be adaptable to most interface architectures. Regardless of the application, the use of IPLIMP for low level link control will not significantly decrease the performance of any underlying protocol layers. That is, IPLIMP will not degrade the performance of the physical devices or software modules it directs. This adaptability is possible since IPLIMP was organized to minimize overhead while maintaining flexibility. The various

characteristics of IPLIMP that permit its application to numerous image-capable systems are listed below in table 2-1. Of course, the image-capable communications requirements listed in table 1-1 have also been addressed by IPLIMP.

IPLIMP CHARACTERISTICS

1. DOES NOT SEGMENT MESSAGES
2. DOES NOT INTERNALLY BUFFER MESSAGES
3. DOES NOT ENCAPSULATE MESSAGES
4. PROVIDES SIMPLE ERROR DETECTION AND REPORTING
5. REQUIRES UNDERLYING SEGREGATION OF CONTROL AND DATA
6. REQUIRES UNDERLYING INDICATION WHEN READY FOR DATA
7. FACILITATES USER SPECIFIED TIME OUTS
8. ACCOMMODATES VARIABLE MESSAGE LENGTHS
9. HANDLES AN ARBITRARY NUMBER OF RECORDS
10. OPERATES INDEPENDENTLY OF UNDERLYING SPEED

Table 2-1

IPLIMP limits overhead by omitting many potential low level communications features in favor of economized functionality and increased dependence upon the underlying layers. For instance, IPLIMP does not use message segmentation, which could be included to adjust record sizes between layers [13]. Segmentation, or framing, consists of breaking streams of data into packets or frames before transmission.

Following their reception, the packets are reassembled into the original data stream. This division and reconstruction of messages adds overhead to communications in a number of manners. First, segmentation requires the use of additional buffers to manipulate the data. However, moving data from one memory location to another takes time. Second, if there is a minimum delay incurred by each transmission, the total minimum delay for a series of packet transmissions will be greater than that of a single message transmission. Third, the processing time needed to divide and reassemble messages can be considerable. Finally, control information is often added to packets for transmission, which increases overhead.

Apart from segmentation, internal message buffering can also be used for throttling throughput. This buffering can be quite costly, both in terms of time and memory expense, especially when handling large image records. Similarly, the addition of control information by message encapsulation can be used independently of segmentation. While being useful for distinguishing message types and ensuring data integrity, message encapsulation increases overhead since the additional data must be transmitted, processed, and possibly buffered. As with segmentation, internal message buffering and message encapsulation were not included in IPLIMP in order to limit overhead.

Another characteristic of IPLIMP that helps reduce overhead is the manner in which it handles errors. Typical error handling schemes involve the detection, correction,

and reporting of errors. Such error handling functions require adding redundant data and coded information to each message. As stated before, overhead is increased by adding information to messages. Overhead also increases if messages are retransmitted when errors are detected. IPLIMP, however, simply monitors whatever operational information is provided by the underlying layers, and returns appropriate messages to the system and users. This simple approach to error handling ensures that the system and users are aware of any problems so that they may act accordingly. Furthermore, the system, users, and underlying layers can utilize checksums, error correction codes, or any additional error handling mechanism deemed necessary. In this manner, IPLIMP operates independently of whatever error handling services are needed by the particular link. Thus, IPLIMP avoids the overhead associated with error handling schemes while allowing a variety of systems to build upon it.

As well as monitoring the return of operational information, IPLIMP requires notification when the underlying layers are ready to receive data. This notification can occur when the peer IPLIMP layer is ready, or when the underlying layers themselves are ready (i.e., when underlying buffers are available). Also, IPLIMP expects the underlying protocols to segregate interface control information from user data. This segregation can be accomplished by multichannel interfaces, programmable hardware links, or underlying encapsulation protocols. Rather than performing these

services, IPLIMP provides streamlined functionality with minimized overhead. Thus, IPLIMP can either take advantage of physical systems that offer these services directly, or utilize additional protocol layers designed to provide these functions. In fact, these characteristics delimit the set of interprocessor link architectures that IPLIMP can support. Serial programmed I/O interfaces with separate data and control channels, DR11-W-compatible parallel DMA interfaces, and various interprocessor links with appropriately designed protocol layers are some of the architectures suitable for IPLIMP control.

Besides limiting overhead, IPLIMP characteristically supports image-capable communications in a flexible manner. First, IPLIMP allows users to specify their own timeout intervals since all imaging equipment interfaces are not operated within the same time constraints. Next, IPLIMP permits users to transmit variable size records, as imaging systems have varying formats and record sizes. Finally, IPLIMP does not limit the number of records that a user transmits or the speed at which data is transferred. By providing such a degree of flexibility for its users, IPLIMP can be adapted to numerous image-capable applications.

Both flexibility and minimized overhead characterize the manner in which IPLIMP is able to support image communications systems. This support is made possible by providing a set of fundamental low level image-capable communications functions and services. Since systems are not equally able

to tolerate the overhead associated with each communications feature, IPLIMP operations were streamlined to provide only the bare essentials for interprocessor link control. Full image-capable support can then be achieved by implementing IPLIMP functions and services as a basis for communications, and supplementing them with additional features, as dictated by the requirements of the interface to be supported.

2.2 OPERATIONS

The complete set of IPLIMP functions and services used to support image-capable interprocessor link communications are listed below in table 2-2. The following material describes the operation of each IPLIMP feature, while a more detailed specification is given in appendix B. In order to understand better the operational description, a conceptual model of IPLIMP is shown in figure 2-1, depicting how IPLIMP might be integrated into a system.

IPLIMP FUNCTIONS AND SERVICES

1. ASSOCIATE/DISSOCIATE FUNCTIONS
2. SEND/RECEIVE FUNCTIONS
3. SEND COMMAND FUNCTIONS AND COMMAND HANDLING SERVICES
4. INITIALIZATION AND DOWNLOADING/LOADING SERVICES

Table 2-2

IPLIMP MODEL

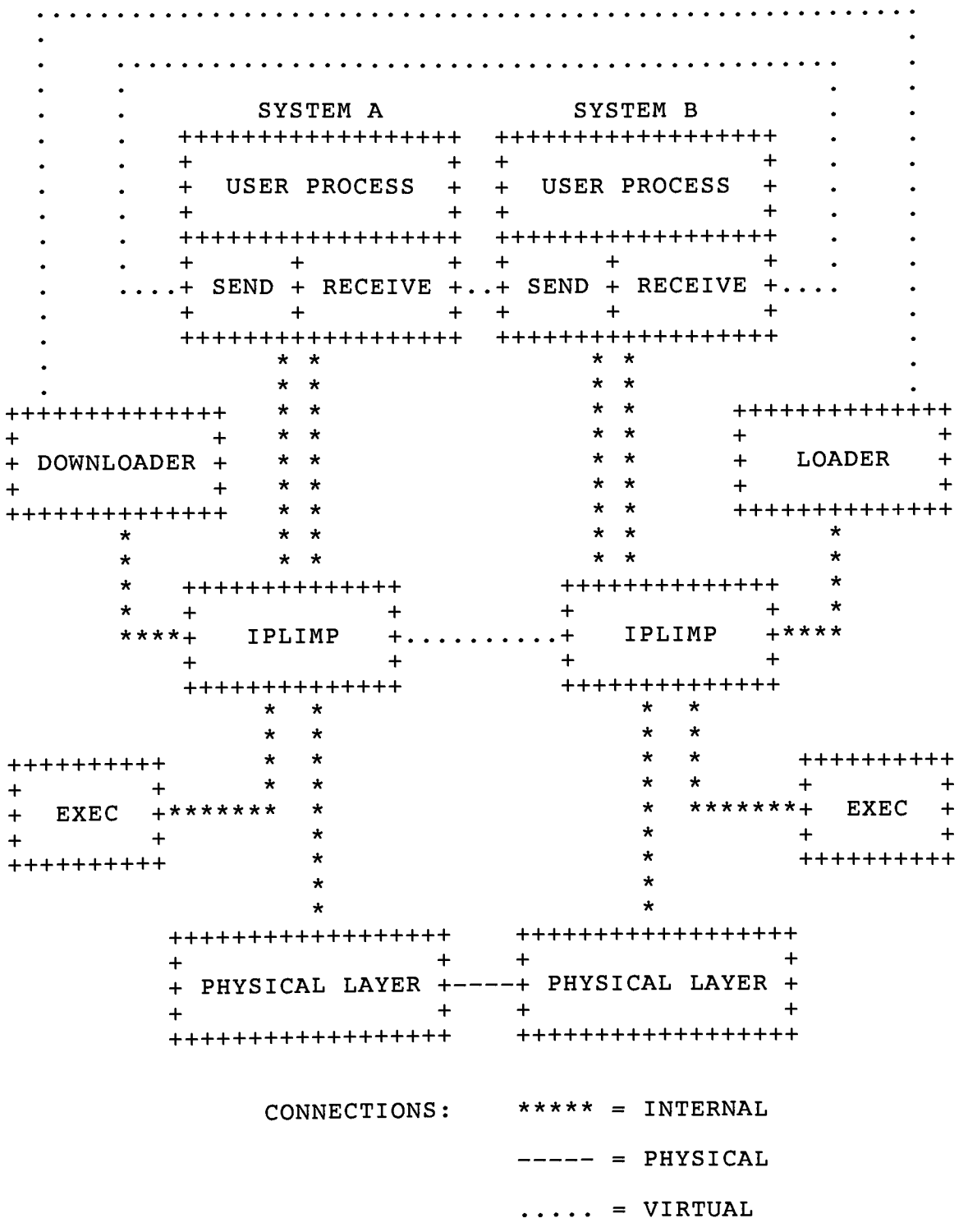

Figure 2-1

As detailed in section 1.1, interprocessor links are formed by interfacing processors together over some physical channel. This channel is depicted as a physical connection between physical layers in the IPLIMP model in figure 2-1. It is these physical layers and connections that provide the raw communications facilities needed to transmit information between systems. By making this raw functionality available for use within the constraints of the host operating system executive, higher level protocols can be designed to affect interprocessor communications. However, the raw communications functions provided by the specialized hardware and software within the physical layer are not necessarily the same on every system. One purpose of IPLIMP is to remedy this problem by presenting a uniform set of low level functions and services to higher level applications.

In order to do this, each IPLIMP implementation must access the particular physical layer and executive functions available on the given processor. In other words, each implementation of IPLIMP is entirely system dependent. Furthermore, as described in section 2.1, IPLIMP requires that the underlying modules be capable of segregating control information from data, as well as indicating when they are ready to receive data. In this manner, IPLIMP can be internally connected only to those systems possessing characteristics which make IPLIMP control feasible.

Once IPLIMP has been installed on a system, it forms a virtual connection with the IPLIMP module on its peer sys-

tem. That is, though IPLIMP modules are not physically connected across an interprocessor link, they are connected via the underlying physical layers. Hence, IPLIMP modules are able to exchange data by communicating with the underlying physical layers which are physically connected. In order to form such a virtual connection, a module on one system must have a peer, which is a corresponding module at the same level on the opposite side of the link. These modules then communicate by internally connecting to the underlying layers and executing appropriate lower level functions. For example, downloader and loader modules also form a virtual connection across an interprocessor link. In this case, the downloader is the module that transmits applications or additional protocol layers to its peer module (the loader). On the other hand, the loader receives transmitted load data from across a link and stores it in its processor memory. As shown in the IPLIMP model in figure 2-1, the downloader and loader modules can connect to their respective IPLIMP modules, then communicate by calling IPLIMP functions. Similarly, user processes can form a virtual connection across an interprocessor link by associating with IPLIMP and utilizing IPLIMP functions to transfer data. The protocol used to accomplish the exchange of data between peers is predetermined, and mutually agreed upon, by the corresponding peer modules on each system.

Before peer processes can communicate across a link by using IPLIMP functions, they must become associated with

the IPLIMP module on their host system. An association, or internal connection, is a defined pathway for communications between service modules on a system. When modules become associated, they open a bidirectional line of communication, so that the underlying protocol can send information to the higher level module, as well as complete functions requested by the higher level protocol. IPLIMP provides associate and dissociate functions so that a downloader, loader, user process, or some other higher level module can establish and close such a line of communication to IPLIMP. However, only one process or module can be associated with IPLIMP at any given time, so there is no sharing of IPLIMP capabilities.

When issuing the IPLIMP associate function to commence communication with IPLIMP, the higher level protocol identifies send and receive processes to IPLIMP so that they may be notified whenever the peer system needs them. In this manner, a logical connection is formed between IPLIMP and the higher level protocol, allowing IPLIMP to notify the associated sender and receiver as needed. Such notification may take place in a number of manners. Generating a software interrupt, setting a semaphore, and making an entry into an event queue are all possible ways to inform a process that it is being asked to communicate. The method used by IPLIMP is entirely system dependent.

Conversely, the dissociate function is used by the associated protocol layer to erase the inter-layer pathway, thereby terminating its logical connection to IPLIMP. Once

IPLIMP is dissociated from a higher level protocol, it is free to be associated with other processes. While it is not associated with a sender and receiver, IPLIMP can pass control information across the interprocessor link. However, IPLIMP is unable to honor requests by its peer system to exchange data without a line of communication to a higher protocol layer.

The associate and dissociate functions provide the ability to open and close lines of communication to IPLIMP, thereby enabling more flexible interaction with IPLIMP. After a line of communication has been established, associated processes access specific IPLIMP functions to exchange data across an interprocessor link. For example, when an associated sender wishes to send a block of data across the link, it invokes the send function, specifying a buffer address, a buffer size, and a timeout period for the operation to complete within. Following its initiation, the IPLIMP send function starts a timer for half the user specified timeout period and waits for the lower level protocol to signal that the peer receiver is ready. Once the ready indication is received, the specified buffer is sent directly across the link. If the timer fires before the ready indication is received or the operation is completed, IPLIMP starts a timer for the remaining timeout period and sends a "ready to send" command word to the peer system. If a "clear to send" command word is received, IPLIMP can proceed to wait for the receiver ready indication and restart the data transmission.

If the entire specified timeout period expires without obtaining the "clear to send" command word, receiving the ready indication, and completing the data transfer, then IPLIMP returns a timeout error to the sender process.

Similarly, when the associated receiver wishes to read a block of data from across the link, it issues the receive function, specifying a buffer address, a buffer size, and a timeout period for the operation to complete within. Once invoked, the IPLIMP receive function starts a timer for half the user specified timeout period, indicates its readiness to receive data, and attempts to fill the user buffer area with data from across the link. If the timer fires before the operation is completed, IPLIMP starts another timer for the remaining timeout period and sends a "ready to receive" command word to the peer system. After the peer responds with a "clear to receive" command word, IPLIMP proceeds to restart the data reception. If the entire specified timeout period expires without obtaining the "clear to receive" command word and completing the data transfer, then IPLIMP returns a timeout error to the receiver process.

As described above, both send and receive functions use command words to attempt to initiate data communications without letting the full timeout period elapse. The use of such command words, however, is not restricted to the internal operation of IPLIMP. In fact, IPLIMP permits the complete asynchronous control of an interprocessor link through a combination of internal and user specified command words.

These control commands are well-defined 16-bit values which have been divided into the following groups: Internal Command Protocol (ICP), Utility Command Protocol (UCP), Status Response Protocol (SRP), and Status Exchange Protocol (SEP). By grouping the command words in this manner, interprocessor link control is organized into functional categories, each of which addresses a particular area of IPLIMP operations.

For example, IPLIMP uses ICP commands to maintain internal control. The commands used by the send and receive functions to wake up the peer system and grant clearance to transfer data are ICP commands. Additional ICP commands are used to indicate when a sender or receiver is not present in response to peer ready commands. Finally, ICP command words are defined for aborting operations, initiating a status exchange using SRP and SEP commands, and flagging illegal commands. Thus, there is an ICP command word available to help regulate every facet of internal IPLIMP communications.

The next group of IPLIMP command words was designed to be used as input to the send command function available for higher level protocols to control IPLIMP operations externally. When an associated process wishes to transmit control information across the link, it issues the send command function, passing IPLIMP the UCP command word to be sent and a timeout period, if appropriate, for a response. For instance, UCP command words are defined to allow higher level protocols to wake up their peer send and receive processes. Though IPLIMP may attempt to wake up an adjacent

user process during the execution of a send or receive function, an IPLIMP user may try to notify its peer process even before it initiates a data transfer. In this manner, IPLIMP users can synchronize their interaction without having to wait for valuable timeout time to elapse. Accompanying the two commands to notify adjacent processes are UCP commands for requesting to download an adjacent system, requesting a load module from an adjacent system, and giving the go ahead to send a data block containing further control information. By using the send command function to transmit the various UCP commands, higher level protocols can exert external control over IPLIMP operations and solicit IPLIMP activity on the peer system.

The remaining two command groups are used internally by IPLIMP to complete the exchange of status initiated by an ICP status request command. A status exchange protocol is used by IPLIMP to obtain information about the operating characteristics of the adjacent system. When IPLIMP first connects to the underlying protocol layers, it attempts to send a status request to its peer. In response to the ICP status request, the peer IPLIMP module sends back an SRP command word containing the system status. Similarly, the initiating IPLIMP module returns an SEP command word, in order to exchange status, following the reception of an SRP command word. This transfer of status information is needed to convey specific system characteristics before enabling full interprocessor communications.

If IPLIMP did not provide such a mechanism for exchanging status, each system would have to make assumptions about the other, which could ultimately lead to a complete communication failure. For example, suppose a link has been operational for quite some time and one of the systems is powered down, then replaced by a processor with different characteristics (i.e., a loadable microcomputer is replaced by a multiuser multitasking minicomputer). Once the new system is powered up, the adjacent system must be informed of its capabilities before inappropriately attempting system downloads or remote control. Not only can functions fail, but others may never be tried, simply because one system may underestimate the capabilities of the other. Furthermore, imagine if one processor disables communication while the adjacent system is off-line, but is never informed of the availability of the link when the adjacent processor resumes operation. IPLIMP avoids these sticky situations by initiating the simple exchange of SRP and SEP command words while synchronizing operations, immediately upon its connection to the lower level protocols.

IPLIMP's implementation of SRP and SEP commands for communication synchronization is derived from the three-way handshake technique designed to handle delayed or lost messages and acknowledgements [19,20]. A three-way handshake usually consists of a series of three messages in which each successive message contains response data, a message number, and the number of the message being acknowledged. Since the

messages in the series must be acknowledged sequentially, any message containing an acknowledgement number that is out of order can be handled without confusion. In addition to acknowledging messages by number, the three-way handshake strictly defines message content, so that once the data exchange is initiated, it will either be completed or rejected within the next two transfers. Thus, a synchronized data exchange is achieved without deadlocks, infinite loops, loss of data, or similar problems associated with invalid message sequences. Although IPLIMP does not employ a message numbering scheme, the SRP and SEP commands are structured to provide a reliable three-step exchange of status, regardless of which system initiates the exchange or when it is requested.

Though the preceeding discussion of the four IPLIMP command groups has left their physical values to be revealed in the appendix, it is important to note that it is the definition of these command groups, the values chosen, and the ways in which they are handled that provide some of the key distinguishing points of IPLIMP. Even the manner in which IPLIMP relies upon lower level protocols to segregate these command words from normal user data contrasts with typical link control techniques. However, once identified, control commands are handled by IPLIMP as they would be by most link protocols. Unlike data that is communicated via the send and receive functions, reception of a command word is more of an asynchronous event. Since the lower levels take care

of identifying the command words, IPLIMP supplies a command handler to which the lower level protocols pass commands as they are received. Once activated, the command word handler checks command validity, performs any required IPLIMP operations, and returns information, if necessary, by sending an appropriate ICP command word. As with any control protocol, the command handler must execute as quickly and efficiently as possible in order to ensure that critical functions are immediately completed.

For instance, the "request to download" and "request for load" UCP command words serve to initiate IPLIMP downloading/loading services. These services may be required by a system that needs a load module to function properly. In order to facilitate the downloading of a system, the IPLIMP command handler quickly engages the downloading/loading services upon the valid reception of one of these commands. A download protocol is then used to handle the communication between a downloader process on the system sending the load module and a loader process on the system receiving the load module. Though somewhat higher level in functionality, such a download protocol is defined as an integral component of IPLIMP. As illustrated above in figure 2-1, this download protocol resides at an intermediate level between normal user processes and IPLIMP itself. Therefore, since the download protocol is so closely associated with IPLIMP, it seems to fit logically within the IPLIMP specification. In fact, Digital Equipment Corporation designed the Maintenance Oper-

ation Protocol (MOP) to provide similar lower level services for DECNET [7].

The IPLIMP download protocol was defined to support systems that cannot archive their own user applications. It is worth noticing, however, that the implementation of these downloading services is completely optional and dependent upon the functionality desired by the systems using the interprocessor link. For instance, if a system requires the ability to be loaded across the link, then it needs a loader process that adheres to the prescribed download protocol. On the other hand, if a system is relied upon to transmit loads across the link, then it needs the corresponding downloader module. In either case, the IPLIMP download protocol describes a methodology for loading systems across an interprocessor link.

A downloader process is either activated by the local system when it wants to send a load module to a peer or by IPLIMP upon reception of a UCP "request for load" command word. If a downloader has been activated by IPLIMP, it must issue the send command function to transmit a UCP "send control block" command word, which signals the peer loader process to continue. Next, the downloader issues the receive function to read a filename block (FNB) from the loader. An FNB is a 64-byte block containing an ASCII character string which is used to specify a load module. Such a specification may be represented differently from system to system due to the disparity between file systems. If the down-

loader was instead activated by the local system, the file specification for the load module must be passed internally to the downloader process. Conversely, a loader process is either activated by the local system when it wants to obtain a load module from a peer or by IPLIMP upon receiving a UCP "request to download" command word. If loader operation was initiated by the local system, it issues the send command function to transmit a UCP "request for load" command word across the link. When the downloader replies with the UCP "send control block" command, the loader issues the send function to send the FNB. Finally, the loader hibernates, waiting to be awakened by IPLIMP upon receiving a "request to download" command. The downloader sends this command only after it has located the file containing the load module (whether specified by the local system or an FNB), and it is ready to begin downloading data. At this point, the download protocol continues as if the downloading system had requested to download its peer.

Before the actual downloading can begin, however, the systems involved must exchange critical information describing the pending load. This handshake is initiated when the downloader issues the send function to transmit a load control block (LCB) to the peer loader. An LCB is a 16-byte block containing four 32-bit fields, each representing a 32-bit integer value stored low-order bit first. The first field (load address) specifies the memory location where the load module is to be stored. The next field (load size)

contains the length of the load module in bytes. The third LCB field (fragment size) is the size in bytes of the fragments that will be transferred across the link in succession until loading is completed. The size of the last fragment may be somewhat less than this size if there is not enough load module data remaining to form a full fragment. The last LCB field (transfer address) specifies the memory location where the execution of the load module begins. These four LCB fields are filled from information obtained by the downloader from the load module file. Once the loader has received the LCB, the data is examined and an LCB acknowledgement (LCB ACK) is created to approve or reject the download request. An LCB ACK is 2 bytes in length with bits set to represent reasons for download rejection, such as illegal load address, illegal load size, illegal fragment size, or illegal transfer address. Following the successful exchange of an LCB and an LCB ACK, with loader approval, downloading may proceed. The load module is then transmitted from the downloader to the peer loader, fragment by fragment, through the use of the send and receive functions.

Though IPLIMP details have been saved for specification in appendix B, the operation of IPLIMP functions and services has now been presented. According to the IPLIMP characteristics, such operation should be capable of providing a basis for image-capable communications over an interprocessor link. By using the IPLIMP specification as a guide for protocol development, systems may be constructed

to accommodate the transfer of images. By adhering to the IPLIMP specification more rigorously, systems will assure themselves of a degree of low level compatibility.

2.3 STANDARDS

As described above, by utilizing IPLIMP, systems can expect to maintain low level image-capable communications compatibility with other IPLIMP compliant systems. However, IPLIMP is just one of many available communications protocols, each of which has been optimized for a particular set of applications. Undoubtedly, this situation poses problems for connecting systems that do not adhere to the same communications guidelines. In fact, the growing number of proprietary protocols has prompted the push for the development of standard communications rules. The steps taken in this direction have been divided along two paths: acceptance of "de facto" standards and use of the International Standards Organization (ISO) proposals [10].

Since it will be difficult to arrive at a consensus as to what proprietary protocol is the "de facto" standard for communications, it might be more reasonable to adhere to a standard methodology for designing systems. Tanenbaum argues that using a layered design reduces complexity by associating groups of functions with particular levels of control, thereby creating a modular architecture which can be easily modified or adapted to at any level [20]. To provide

such a standard, ISO has developed the Open Systems Interconnect (OSI) reference model which describes a general layered communications architecture [13,21]. As established communications protocols such as SNA and DECNET have begun to adapt themselves to this model, support for ISO standards has increased [17,18]. Due to the growing importance of the ISO OSI reference model, it is imperative to recognize how IPLIMP either fits, or does not fit, into the standard layered structure of the model.

The ISO OSI reference model provides a framework for the development of communications architectures as a basis for simplifying the interconnection of systems [15]. Acting as a communications standard, the ISO OSI model offers a set of guidelines for classification of layers in terms of functionality at seven distinct levels [13]. As shown in figure 2-2, the three lowest layers (physical, data link, and network) specify an interface chain across a communications subnet, while the remaining four layers (transport, session, presentation, and application) define an end-to-end system interface. According to the OSI model, the subnet consists of a series of intermediate relaying systems which use only the lower layers of functionality in order to route information between the end nodes [13]. This exemplifies how ISO OSI based systems implement whatever layers, or functions within layers, they needed to meet specified communications requirements. The functions that ISO has grouped together for each OSI layer have been listed in table 2-3.

ISO OSI REFERENCE MODEL

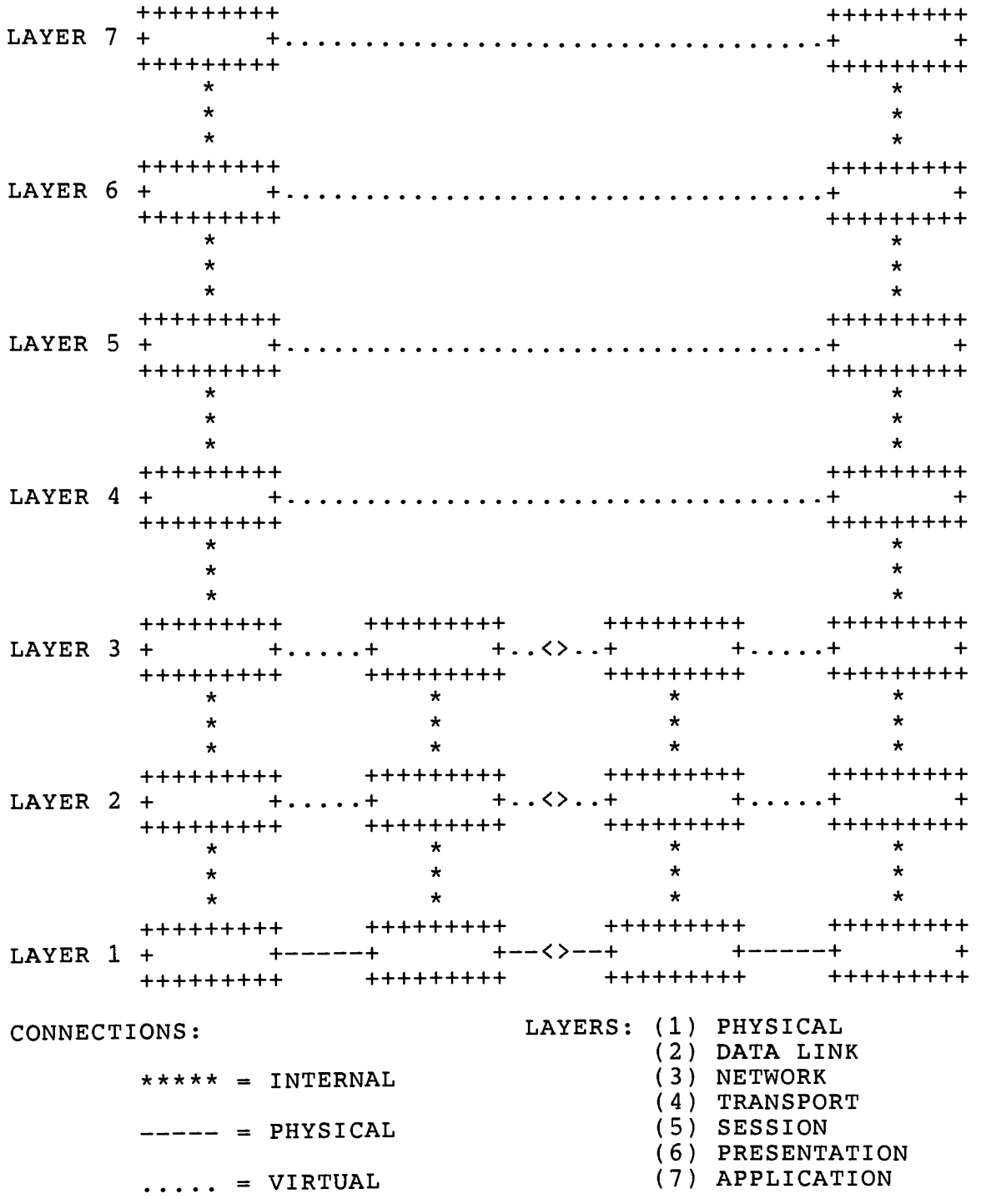

Figure 2-2

ISO OSI PROTOCOL LAYERS

LAYER	FUNCTIONS
1 - PHYSICAL	- physical characteristics (electrical & mechanical) - voltage & time specs - signalling requirements - connection type - number of connections
2 - DATA LINK	- link interconnection & control - synchronization - physical error handling - data encapsulation/framing - bit/character stuffing - flow control - message segmentation
3 - NETWORK	- subnet to host interface (datagram service vs. virtual circuits) - routing - subnet flow control - accounting services - congestion prevention - deadlock prevention
4 - TRANSPORT	- connection multiplexing - end-to-end flow control - connection establishment - connection termination - process naming
5 - SESSION	- system access verification - management services - crash recovery - synchronization services - transport error handling
6 - PRESENTATION	- security (data encryption) - data compression - terminal handling - file transfer
7 - APPLICATION	- user defined protocols

Table 2-3

Of the functions listed in table 2-3, those that are assigned to the data link layer are most closely associated with the services that are performed by IPLIMP. However, as indicated in table 2-1, IPLIMP does not perform such data link functions as segmentation or encapsulation. Furthermore, IPLIMP relies on some lower level protocol to provide services such as data and control segregation. The ISO OSI model, on the other hand, accommodates message segregation with data link framing services or specialized physical signalling. While the ISO OSI model does not directly address the problem of image capable communications in any of its layers, IPLIMP describes a basic set of data link functions that can support the transfer of images. Therefore, though it does not support every ISO OSI data link function, IPLIMP can be treated as a data link protocol. In that regard, IPLIMP can be combined with other data link functions, if necessary, to characterize the data link protocol for a given application. In a similar fashion, additional layers can be added to the customized data link layer in order to characterize an entire communications architecture. In this manner, IPLIMP is compatible with the ISO OSI layered architecture at the data link level. Similar comparisons have also been made by prior image capable interprocessor link protocols, indicating that a low level link protocol can be applied to full network communications under ISO OSI guidelines [1,9,11].

3 IMPLEMENTATION DESCRIPTIONS

By using IPLIMP as a basis for communications, several systems have been able to support image-capable operations successfully. In fact, three interprocessor link configurations were implemented in order to verify IPLIMP capabilities. In each of these three prototypes, IPLIMP was employed as a control facility for DR11-W compatible interface hardware. DR11-W-type interfaces were selected because they possess the underlying features required for IPLIMP support. Among the DR11-W characteristics that can help accommodate the adaptation of IPLIMP are DMA parallel data transmission, asynchronous control word transmission, error checking, and status bit notification when the peer receiver is ready for data. The actual interfaces, which were provided by Eastman Kodak Company's Digital Technology Research Lab, were manufactured by Digital Equipment Corporation as standard off-the-shelf communications components.

3.1 AN INTRAPROCESSOR LINK

The first IPLIMP compliant system, which is shown in figure 3-1, was created by connecting together two DR11-W 16-bit parallel DMA interface cards that plugged into the same PDP-11/34 minicomputer. The initial implementation was centered around a DR11-W device driver written for the RSX-11M operating system running on the PDP-11/34. While normal

RSX-11M driver mechanisms were used to provide a callable set of IPLIMP functions, a method for aborting IPLIMP operations, and an internal IPLIMP command word handler, special code was written to synchronize the timing of IPLIMP events and to set local event flags for notifying user processes [8]. The driver code was written in the PDP-11 assembly language (MACRO-11), since it is the language best suited for writing efficient RSX-11M system level software.

AN INTRAPROCESSOR LINK

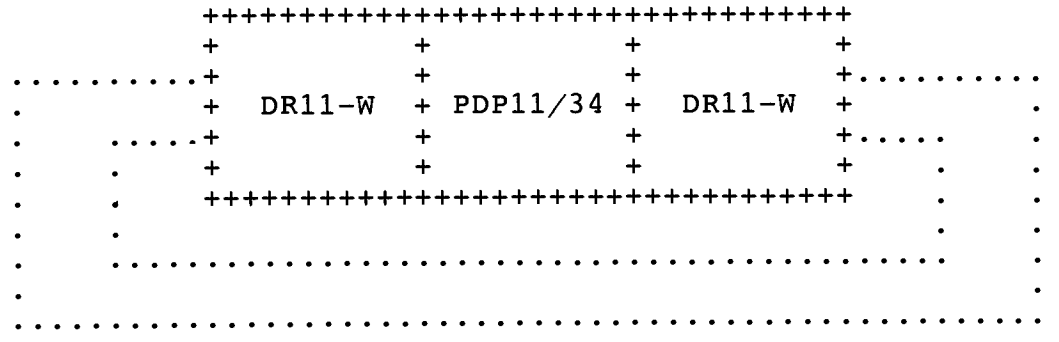

Figure 3-1

Though the driver code was written in MACRO-11, a test program was written in FORTRAN-77 to demonstrate that higher level programming languages can utilize the MACRO-11 coded driver functions. In order to exercise the intraprocessor link and verify IPLIMP functionality, the test program allowed users to select various combinations of send and receive wait states to perform a loopback test across the link. This loopback test consisted of creating records, sending data across the link, comparing the data received

with those sent, sending data back, recomparing, and compiling results. Use of this loopback test was important during the development of the IPLIMP specification as well as the DR11-W device driver.

Though the intraprocessor link configuration and its associated software were vital during the initial stages of development of the IPLIMP specification, they did not exercise every IPLIMP function and feature. However, it was in part due to the fact that it did not require complete IPLIMP support that the intraprocessor link was chosen as the first IPLIMP prototype. By utilizing a single processor for development and testing, the bulk of the IPLIMP protocol was implemented and verified with approximately half the effort of that needed for conventional multi-processor interprocessor link communications systems. Also, since there was no requirement for transferring additional protocol layers between processors, there was no need to employ IPLIMP downloading/loading services. Thus, this first step toward proving IPLIMP functionality was accomplished by keeping both hardware and software interfacing, debugging, and modification as simple as possible.

3.2 INTERPROCESSOR LINK #1

The second step toward proving IPLIMP functionality was to support an interprocessor link between two systems, using the full set of IPLIMP features. This was achieved by

expanding upon the initial device driver and test program written for the first prototype. In order to minimize the time needed to develop the second prototype, as much of the first prototype was used as possible. This was accomplished by connecting a third DR11-W interface card on the PDP-11/34 to a compatible DRV11-WA interface module on an SBC-11/21 (FALCON) microcomputer, as shown in figure 3-2 below. In this manner, the second prototype was able to utilize the initial RSX-11M implementation of IPLIMP, leaving the bulk of the development to the FALCON system. Originally, the FALCON used a DRV11-B interface module to communicate with the PDP-11/34, but since the DRV11-B needed hardware modifications in order to work properly, the more compact, fully functional DRV11-WA was installed.

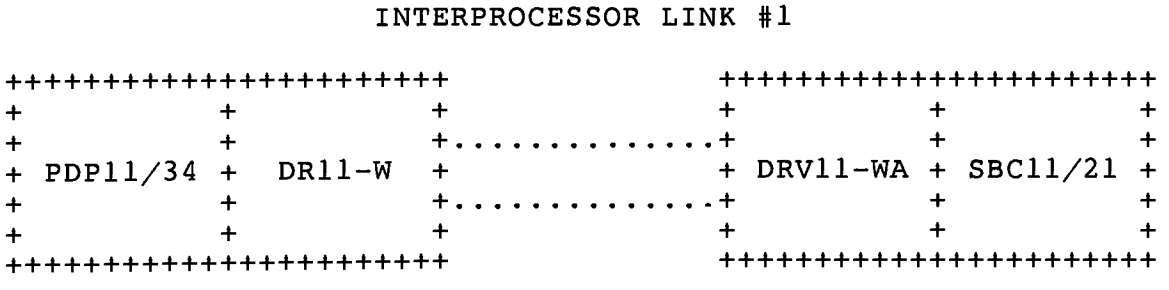

Figure 3-2

The development required for the FALCON system consisted mainly of writing a ROM-resident executive based on IPLIMP in MACRO-11. This executive included a loader module for obtaining applications software that could not be stored on the FALCON. Thus, a corresponding downloader utility was

required for the PDP-11/34 which could be activated either by the DR11-W driver upon receiving a load request from the FALCON, or by a preloader utility upon accepting a download command from a terminal. A test procedure, similar in functionality to that of the loopback link test, was written in FORTRAN-77 for the PDP-11/34, while a downloadable peer process was written in MACRO-11 for the FALCON.

3.3 INTERPROCESSOR LINK #2

Comprehensive verification of IPLIMP functionality was completed on a third prototype which featured a third type of CPU and another interprocessor link combination. The final IPLIMP compliant prototype, shown in figure 3-3, was formed by connecting a DRV11-WA interface board residing on a Q-BUS based PDP-11/23 microcomputer to another DRV11-WA interface on a second FALCON system. As with the previous interprocessor link implementation, the DRV11-B interface originally used on the FALCON was replaced with a DRV11-WA. Since the PDP-11/23 microcomputer operated under RSX-11M, IPLIMP support was obtainable by adding conditional-code to the DR11-W driver for DRV11-WA interface control. Aside from what resulted in minor additions to the DR11-W driver, identical software was used on both the second and third IPLIMP prototype interprocessor links. Thus, by using the UNIBUS based PDP-11/34 and the Q-BUS based PDP-11/23 and FALCON systems, UNIBUS to UNIBUS (intraprocessor link),

UNIBUS to Q-BUS (interprocessor link #1), and Q-BUS to Q-BUS (interprocessor link #2) image-capable communications systems were designed to operate under IPLIMP control.

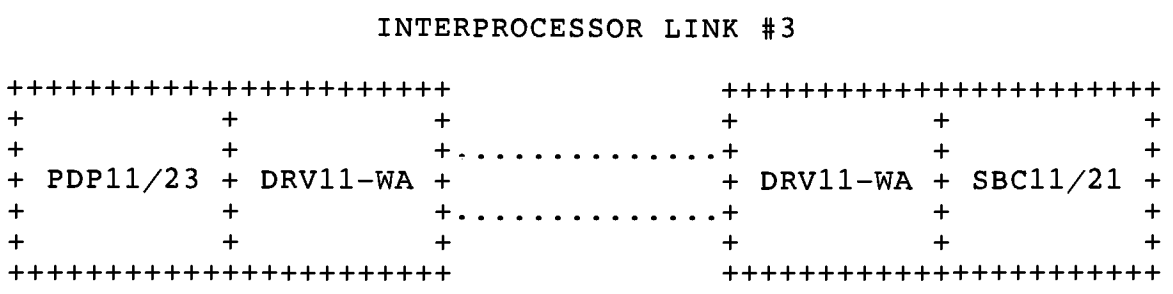

Figure 3-3

4 CONCLUSIONS

IPLIMP, an image-capable interprocessor link communications protocol, has been designed to provide flexibility with a minimal amount of overhead. At a level comparable to the ISO OSI data link layer, IPLIMP defines the basic set of functions and services needed to facilitate image communications over an interprocessor link. Use of such a protocol may help to improve the development of imaging systems. For example, following the development of the second prototype system described in chapter 3, little modification was necessary when changing processors to form the third prototype. Since this is a simple case, imagine if instead of replacing a FALCON host, an image digitizing unit was used to replace the FALCON. In this case, the image digitizer would have to be designed to follow IPLIMP, as was the FALCON, and test procedures would have to be designed to support the comparison of data sent by the digitizer when scanning a known image. Similar changes would be expected if changing from the image digitizer to an image printer or display unit. Notice that neither situation requires modification to the IPLIMP layer. Compare this, however, to the customization required to develop the Kodak interfaces mentioned in chapter 1. The same device changes made to a previously customized system would necessitate the development of new interface control protocols, user utilities, and test procedures, due to the tailoring of interfaces to specific device needs rather than

designing links for general image transfer functionality.

Thus, with IPLIMP, there is a potential for reduced design complexity and customization, as less attention needs to be paid to the host when interfacing new devices. At this point, several avenues may be explored to achieve the full benefit of IPLIMP services. First, DR11-W-compatible IPLIMP implementations could be developed for additional operating systems in order to build a base of common IPLIMP support for imaging devices. In fact, a VMS implementation is already nearing completion, while IPLIMP development has recently been initiated for a Motorola VMEbus system. Second, IPLIMP support could be provided for additional physical subsystems such as a simple dual-node base band channel or a fiber optic link (uni-fiber or multi-fiber). Building upon this base of IPLIMP support, routing and flow control protocols could be added to support complete image-capable network services. Of course, consideration must be given to image-capable communications requirements if such protocols are developed. Finally, work could be done to support image communications over topologies such as broadcast channels or rings. Here, specialized priority schemes might help maintain the flow of critical image information to/from highly volatile devices, while non-critical nodes are delayed.

Regardless of the avenue taken, IPLIMP can be employed as a basis for image-capable communications. For low level interprocessor link applications, IPLIMP defines a flexible protocol adaptable to various speed requirements on

several hardware architectures. With a common basis for image communications, imaging subsystems become more transportable. For higher level communications between multiple imaging systems, IPLIMP provides a callable set of functions to affect the transfer of images between adjacent points. If higher level image capable protocols can be designed, a complete architecture for image communications applications can be developed. At a low level, IPLIMP provides a potential starting point for this work.

5 BIBLIOGRAPHY

[1] Alzner, E., G. Arnik, F.W. Gutzwiller, K.C. Menke and M.F. Rossi. "A Standard Product Interface for Digital Medical Imaging Equipment." MEDICAL IMAGES AND ICONS, MEDPACS: PICTURE ARCHIVING AND COMMUNICATIONS SYSTEMS, MEDPICS: PICTURE INTERPRETATION COMPUTERS AND SYSTEMS, MEDGRAPH: COMPUTER GRAPHICS. Ed. Andre' J. Duerinckx, Judith Prewitt and Murray Lowe. Proc. SPIE 515 (1984), 129-135.

[2] Brown, Robert W. and Peter R. Puccini. "Dicomed D47 RSX-11M Interface Control Software." Unpublished. Eastman Kodak Company, 1979-1984.

[3] Brown, Robert W. and Peter R. Puccini. "Flatbed Scanner RSX-11M Interface Control Software." Unpublished. Eastman Kodak Company, 1980-1984.

[4] Brown, Robert W. and Peter R. Puccini. "Framestore RSX-11M Interface Control Software." Unpublished. Eastman Kodak Company, 1984-1986.

[5] Brown, Robert W. and Peter R. Puccini. "Scan/Play RSX-11M Interface Control Software." Unpublished. Eastman Kodak Company, 1984-1985.

[6] Brown, Robert W. and Ralph Schaetzing. "Optronics Colormation C-4500 RSX-11M Interface Control Software." Unpublished. Eastman Kodak Company, 1984-1986.

[7] Digital Equipment Corporation. DECNET DIGITAL NETWORK ARCHITECTURE (PHASE III): GENERAL DESCRIPTION. Digital Equipment Corporation, 1980.

[8] -------. RSX-11M GUIDE TO WRITING AN I/O DRIVER. Digital Equipment Corporation, 1981.

[9] Elms, Duane, Owen Nelson and Alan Rothlauf. "A Medical Imaging Equipment Digital Image Transfer Interface Standard." APPLICATION OF OPTICAL INSTRUMENTATION IN MEDICINE XII: MEDICAL IMAGE PRODUCTION, PROCESSING, DISPLAY AND ARCHIVING. Ed. Roger H. Schneider and Samuel J. Dwyer III. Proc. SPIE 454 (1984), 86-90.

[10] Glazer, S. "Committees, Vendors Chase Elusive Networking Standards." Mini-Micro Systems, 16, No. 8 (1983), 101-110.

[11] Horii, S.C., J.L. Lehr, G.S. Lodwick, Y. Wang and J.S. Zielonka. "Overview of ACR-NEMA Digital Imaging and Communication Standard." 3RD INTERNATIONAL CONFERENCE AND WORKSHOP ON PICTURE ARCHIVING AND COMMUNICATIONS SYSTEMS (PACS III) FOR MEDICAL APPLICATIONS. Ed. Roger H. Schneider and Samuel J. Dwyer III. Proc. SPIE 536 (1985), 132-138.

[12] Hottinger, C.F. and K. Koestner. "Ultrasound Picture Archiving and Communications Systems." 1ST INTERNATIONAL CONFERENCE AND WORKSHOP ON PICTURE ARCHIVING AND COMMUNICATIONS SYSTEMS (PACS) FOR MEDICAL APPLICATIONS. Ed. Andre' J. Duerinckx. Proc. SPIE 318 (1982), 369-374.

[13] ISO/TC 97/SC 21. INTERNATIONAL STANDARD ISO 7498: INFORMATION PROCESSING SYSTEMS - OPEN SYSTEMS INTERCONNECTION - BASIC REFERENCE MODEL. New York, N.Y.: International Organization for Standardization, 1984.

[14] Kuo, Franklin F., ed. PROTOCOLS AND TECHNIQUES FOR DATA COMMUNICATION NETWORKS. Englewood

Cliffs, N.J.: Prentice-Hall, Inc., 1981.

[15] Patton, Carole. "Emerging Standards and New Software Seek to Untangle Computer Networks." Electronic Design, 33, No. 30 (1985), 62-68.

[16] Puccini, Peter R. "Framestore VMS Interface Control Software." Unpublished. Eastman Kodak Company, 1986.

[17] Rauch-Hindin, Wendy. "OSI Standards Spur Product Profusion." Mini-Micro Systems, 19, No. 8 (1986), 67-82.

[18] --------. "Upper Level OSI Protocols Near Completion." Mini-Micro Systems, 19, No. 9 (1986), 53-66.

[19] Stallings, William. DATA AND COMPUTER COMMUNICATIONS. New York, N.Y.: MacMillan Publishing Company, 1985.

[20] Tanenbaum, Andrew S. COMPUTER NETWORKS. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1981.

[21] Zimmerman, Hubert. "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection." IEEE Transactions on Communications, COM-28, No. 4 (1980), 425-432.

APPENDIX A GLOSSARY

ASSOCIATE: To form a logical connection between communications modules on the same system.

DISSOCIATE: To terminate a logical connection between communications modules on the same system.

DOWNLOADER: A module that transmits operational data to its peer loader module over a communications system. The transmitted data is typically used to continue execution on the processor which is being loaded.

DOWNLOAD PROTOCOL: The rules and regulations governing the transfer of data between a downloader and a loader over a communications system.

ENCAPSULATION: The process of adding control information to units of data being transmitted over a communications system.

FRAMING: See SEGMENTATION.

HEADER: The added control information transmitted with the original data after encapsulation.

IMAGE: A visible entity, or the data used to create such an entity, i.e., digital information.

IMAGE CAPABLE: Able to communicate images or large amounts of data at speeds consistent with the image generator.

INTERFACE: A connection between communications modules. Usually, however, the physical connection between processors, or the hardware modules used to form such a connection.

INTERNAL CONNECTION: A logical link or association formed between modules on the same system.

INTERPROCESSOR LINK: A communications system that allows the transfer of information between two otherwise separate processors.

INTRAPROCESSOR LINK: A communications system that allows the transfer of information between different locations within the same processor.

LOADER: A module that receives operational data from its peer downloader module over a communications system. The transmitted data is typically used for further execution by the processor receiving the load.

PEERS: Modules residing at the same level on opposite ends of a communications link.

PHYSICAL CONNECTION: An actual link between otherwise separate processors that enables the raw exchange of communications data.

PROTOCOL: A set of rules, formats, and/or procedures that defines the interaction between modules in a communications system.

SEGMENTATION: The process of dividing units of information to be transferred into smaller units before actual transmission.

THREE-WAY HANDSHAKE: A technique for handling delayed or lost messages and their respective acknowledgements. Based on the use of numbered messages and defined responses in a three message sequence.

VIRTUAL CONNECTION: A link between modules on separate systems that relies on the physical link created at some lower level.

APPENDIX B IPLIMP ALGORITHMIC SPECIFICATION

The complete set of IPLIMP functions is presented in this appendix, modelled in the form of a general algorithmic notation. The pseudo-code, along with the associated data structure definitions and value assignments, is intended to serve as the primary specification for any implementation of IPLIMP. Although this model is presented in a form which is optimized for clarity of presentation, actual IPLIMP implementations will undoubtedly exhibit more complexity, since they must adapt to any architectural constraints imposed by the particular interprocessor interfaces. Within this context, it is important to realize that adhering to the structure of this model is not mandatory for achieving a correct IPLIMP implementation. Instead, the structure of this model is useful only as an aide to understanding the required behavioral aspects of the internally and externally initiated IPLIMP functions. Furthermore, the use of the algorithmic notation for this model does not imply that IPLIMP must be implemented in software. Therefore, an actual IPLIMP implementation may be a complex structure consisting of software, firmware, and/or hardware modules.

As an aide to following this IPLIMP model, a number of conventions are listed below in tables B-1, B-2, and B-3. The various symbols and keywords described below are not required IPLIMP mnemonics, though they do provide a consistent terminology for describing functions, data, and operations.

PARAMI.descriptor - A logical, system dependent value used as input to a function.

PARAMO.descriptor - A logical, system dependent value used as output from a function or as a status indicator following a function call.

ICP.descriptor - A physical, system independent value that is transferred across an interprocessor link for IPLIMP operations control.

UCP.descriptor - A physical, system independent value that the IPLIMP user sends to IPLIMP to transfer across an interprocessor link for IPLIMP operations control.

STATUS_RESPONSE.modifier - A physical, system dependent value that is transferred across an interprocessor link as an answer to a status request and as a request for a status exchange.

STATUS_EXCHANGE.modifier - A physical, system dependent value that is transferred across an interprocessor link as an answer to a status response.

descriptor = An identifier specifying a unique value.

modifier = An identifier specifying a modified or partially system dependent value.

Table B-1

FUNCTIONAL CONVENTIONS

Primitive.function - A primitive function or service provided by IPLIMP to an IPLIMP user or higher level protocol.
System.function - A system function or service accessed by IPLIMP.
Sublevel.function - A lower level function or service accessed by IPLIMP.
function = An identifier specifying a unique service.

Table B-2

OPERATIONAL CONVENTIONS

wait - Specifies a set of wait conditions and suspends execution until at least one of them is met. A suspended state can be superseded by the occurrence of an event which causes IPLIMP to terminate execution at the wait point, continuing at some other location.
execute - Specifies a function which must be executed before continuing with the next instruction.
"state" - A logical condition flag used to describe the state of internal IPLIMP operations.

Table B-3

B.1 CONTROL COMMANDS AND STATUS CODES

CONTROL COMMAND AND STATUS PROTOCOLS

16 BIT COMMANDS	PROTOCOL
000000 - 037777	INTERNAL COMMAND PROTOCOL (ICP)
040000 - 077777	UTILITY COMMAND PROTOCOL (UCP)
100000 - 137777	STATUS RESPONSE PROTOCOL (SRP)
140000 - 177777	STATUS EXCHANGE PROTOCOL (SEP)

SRP & SEP STATUS CODE BIT DEFINITIONS

BIT MASK	MEANING
000001	Downloader_Resident
000002	Loader_Resident
000004-020000	RESERVED
040000	Status_Exchange_Flag
100000	Valid_Status_Flag

UCP COMMAND CODE DEFINITIONS

COMMAND CODE	DEFINITION
040001	UCP.Notify_Adjacent_Receiver
040002	UCP.Notify_Adjacent_Sender
040004	UCP.Load_Request
040010	UCP.Download_Request
040020	UCP.Send_Control_Block
040040	RESERVED
040100	RESERVED
040200	RESERVED
040400	RESERVED
041000	RESERVED
042000	RESERVED
044000	RESERVED
050000	RESERVED
060000	RESERVED
040000-077777	ILLEGAL (UNLESS DEFINED ABOVE)

ICP COMMAND CODE DEFINITIONS

COMMAND CODE	DEFINITION
000001	ICP.Ready_To_Send
000002	ICP.Ready_To_Receive
000004	ICP.Clear_To_Send
000010	ICP.Clear_To_Receive
000020	ICP.Receiver_Not_Present
000040	ICP.Sender_Not_Present
000100	ICP.Abort_I/O
000200	ICP.Status_Request
000400	ICP.Invalid_Function
001000	RESERVED
002000	RESERVED
004000	RESERVED
010000	RESERVED
020000	RESERVED
000000-037777	ILLEGAL (UNLESS DEFINED ABOVE)

B.2 ASSOCIATE FUNCTION

```
Primitive.ASSOCIATE
        (PARAMI.Send_Process_Access_Information,
         PARAMI.Receive_Process_Access_Information,
         PARAMO.Status.modifier):

1)  if "IPLIMP IN USE"
        then return PARAMO.Status.Already_Associated

2)  if "INVALID INPUT"
        then return PARAMO.Status.Bad_Parameters

3)  set S_ACCESS = PARAMI.Send_Process_Access_Information

4)  set R_ACCESS = PARAMI.Receive_Process_Access_Information

5)  mark "IPLIMP IN USE"

6)  return PARAMO.Status.Success
```

B.3 DISSOCIATE FUNCTION

```
Primitive.DISSOCIATE
        (PARAMO.Status.modifier):

1)  if "IPLIMP NOT IN USE"
        then return PARAMO.Status.Not_Associated

2)  reset S_ACCESS

3)  reset R_ACCESS

4)  mark "IPLIMP NOT IN USE"

5)  return PARAMO.Status.Success
```

B.4 RECEIVE FUNCTION

```
Primitive.RECEIVE
        (PARAMI.Buffer_Address,
         PARAMI.Buffer_Size,
         PARAMI.Timeout_Count,
         PARAMO.Status.modifier):

1)  if "IPLIMP NOT IN USE"
        then return PARAMO.Status.Not_Associated

2)  if "LINK DOWN" or "INITIALIZING"
        then return PARAMO.Status.Link_Not_Ready

3)  if "INVALID INPUT"
        then return PARAMO.Status.Bad_Parameters

4)  set TIMER = (1/2) * (PARAMI.Timeout_Count)

5)  execute Sublevel.Receive_Block(PARAMI.Buffer_Address,
                                   PARAMI.Buffer_Size)

        NOTE:  The sublevel protocol must handle whatever
        ready indication is needed by the peer system.

6)  wait {"RECEIVE DONE","RECEPTION ERROR","TIME UP"}

7)  if "RECEIVE DONE"
        then return PARAMO.Status.Success
        else if "RECEPTION ERROR"
                then return PARAMO.Status.Receive_Error

8)  if "PHASE 2 IN PROGRESS"
        then return PARAMO.Status.Timeout_Error
        else mark "PHASE 2 IN PROGRESS"

9)  set TIMER = (1/2) * (PARAMI.Timeout_Count)

10) execute Sublevel.Send_Control(ICP.Ready_To_Receive)

11) wait {(ICP.Sender_Not_Present),"TIME UP",
          (ICP.Clear_To_Receive)}

12) if (ICP.Sender_Not_Present)
        then return PARAMO.Status.Sender_Not_Present
        else if "TIME UP"
                then return PARAMO.Status.Timeout_Error
                else continue at (5)
```

B.5 SEND FUNCTION

```
Primitive.SEND
        (PARAMI.Buffer_Address,
         PARAMI.Buffer_Size,
         PARAMI.Timeout_Count,
         PARAMO.Status.modifier):

1)  if "IPLIMP NOT IN USE"
        then return PARAMO.Status.Not_Associated

2)  if "LINK DOWN" or "INITIALIZING"
        then return PARAMO.Status.Link_Not_Ready

3)  if "INVALID INPUT"
        then return PARAMO.Status.Bad_Parameters

4)  set TIMER = (1/2) * (PARAMI.Timeout_Count)

5)  wait {"RECEIVER READY","TIME UP"}

6)  if "TIME UP"
        then continue at (10)

7)  execute Sublevel.Transmit_Block(PARAMI.Buffer_Address,
                                    PARAMI.Buffer_Size)

8)  wait {"TRANSMIT DONE","TRANSMIT ERROR","TIME UP"}

9)  if "TRANSMIT DONE"
        then return PARAMO.Status.Success
        else if "TRANSMIT ERROR"
                then return PARAMO.Status.Send_Error

10) if "PHASE 2 IN PROGRESS"
        then return PARAMO.Status.Timeout_Error
        else mark "PHASE 2 IN PROGRESS"

11) set TIMER = (1/2) * (PARAMI.Timeout_Count)

12) execute Sublevel.Send_Control(ICP.Ready_To_Send)

13) wait {(ICP.Receiver_Not_Present),"TIME UP",
          (ICP.Clear_To_Send)}

14) if (ICP.Receiver_Not_Present)
        then return STATUS.Receiver_Not_Present
        else if "TIME UP"
                then return STATUS.Timeout_Error
                else continue at (5)
```

B.6 SEND COMMAND FUNCTION

```
Primitive.SEND_COMMAND
        (PARAMI.Command,
         PARAMI.Timeout_Count,
         PARAMO.Status.modifier):

1)  if "IPLIMP NOT IN USE"
        then return PARAMO.Status.Not_Associated

2)  if "LINK DOWN" or "INITIALIZING"
        then return PARAMO.Status.Link_Not_Ready

3)  if "INVALID INPUT"
        then return PARAMO.Status.Bad_Parameters

4)  case PARAMI.Command:

      UCP.Send_Control_Block then continue at (SC1.1)

      UCP.Notify_Adjacent_Receiver then continue at (SC2.1)

      UCP.Notify_Adjacent_Sender then continue at (SC3.1)

      UCP.Load_Request then continue at (SC4.1)

      UCP.Download_Request then continue at (SC5.1)

    else return PARAMO.Status.Illegal_Command


SC1.1)  execute Sublevel.Send_Control(PARAMI.Command)

SC1.2)  return PARAMO.Status.Success



SC2.1)  set TIMER = (PARAMI.Timeout_Count)

SC2.2)  execute Sublevel.Send_Control(PARAMI.Command)

SC2.3)  wait {(ICP.Receiver_Not_Present),"TIME UP",
              (ICP.Clear_To_Send)}

SC2.4)  if (ICP.Receiver_Not_Present)
           then return PARAMO.Status.Receiver_Not_Present
           else if "TIME UP"
                   then return PARAMO.Status.Timeout_Error
                   else return PARAMO.Status.Success
```

```
SC3.1)  set TIMER = (PARAMI.Timeout_Count)

SC3.2)  execute Sublevel.Send_Control(PARAMI.Command)

SC3.3)  wait {(ICP.Sender_Not_Present),"TIME UP",
              (ICP.Clear_To_Receive)}

SC3.4)  if (ICP.Sender_Not_Present)
           then return PARAMO.Status.Sender_Not_Present
           else if "TIME UP"
                   then return PARAMO.Status.Timeout_Error
                   else return PARAMO.Status.Success



SC4.1)  if P_STATUS indicates "PEER DOWNLOADER NOT RESIDENT"
           then return PARAMO.Status.Command_Not_Supported

SC4.2)  if "LOADER NOT RESIDENT"
           then return PARAMO.Status.Command_Not_Supported

SC4.3)  set TIMER = (PARAMI.Timeout_Count)

SC4.4)  execute Sublevel.Send_Control(PARAMI.Command)

SC4.5)  wait {"TIME UP",(UCP.Send_Control_Block)}

SC4.6)  if "TIME UP"
           then return PARAMO.Status.Timeout_Error
           else return PARAMO.Status.Success



SC5.1)  if P_STATUS indicates "PEER LOADER NOT RESIDENT"
           then return PARAMO.Status.Command_Not_Supported

SC5.2)  if "DOWNLOADER NOT RESIDENT"
           then return PARAMO.Status.Command_Not_Supported

SC5.3)  set TIMER = (PARAMI.Timeout_Count)

SC5.4)  execute Sublevel.Send_Control(PARAMI.Command)

SC5.5)  wait {"TIME UP",(UCP.Send_Control_Block)}

SC5.6)  if "TIME UP"
           then return PARAMO.Status.Timeout_Error
           else return PARAMO.Status.Success
```

B.7 COMMAND HANDLING PROCESS

```
Command Handler:
1)  wait {(P_COMMAND)}
2)  case P_COMMAND:
       ICP.Ready_To_Send then execute (CH2.1)
       ICP.Ready_To_Receive then execute (CH3.1)
       ICP.Clear_To_Send then execute (CH4.1)
       ICP.Clear_To_Receive then execute (CH5.1)
       ICP.Receiver_Not_Present then execute (CH4.1)
       ICP.Sender_Not_Present then execute (CH5.1)
       ICP.Abort_I/O then execute (CH1.1)
       ICP.Status_Request then execute (CH11.1)
       ICP.Invalid_Function then execute (CH6.1)
       UCP.Notify_Adjacent_Receiver then execute (CH2.1)
       UCP.Notify_Adjacent_Sender then execute (CH3.1)
       UCP.Load_Request then execute (CH7.1)
       UCP.Download_Request then execute (CH8.1)
       UCP.Send_Control_Block then execute (CH9.1)
       STATUS_RESPONSE.modifier then execute (CH13.1)
       STATUS_EXCHANGE.modifier then execute (CH12.1)
    else execute CH10.1
3)  continue at (1)


CH1.1)  if "SEND/RECEIVE/SEND COMMAND PRIMITIVE IN PROGRESS"
           then execute System.Abort_Primitive
CH1.2)  return to caller
```

```
CH2.1)  if "IPLIMP IN USE"
           then continue at (CH2.4)

CH2.2)  execute Sublevel.Send_Control
                        (ICP.Receiver_Not_Present)

CH2.3)  return to caller

CH2.4)  execute System.Notify_Process(R_ACCESS)

CH2.5)  execute Sublevel.Send_Control(ICP.Clear_To_Send)

CH2.6)  continue at (CH2.3)



CH3.1)  if "IPLIMP IN USE"
           then continue at (CH3.4)

CH3.2)  execute Sublevel.Send_Control
                        (ICP.Sender_Not_Present)

CH3.3)  return to caller

CH3.4)  execute System.Notify_Process(S_ACCESS)

CH3.5)  execute Sublevel.Send_Control(ICP.Clear_To_Receive)

CH3.6)  continue at (CH3.3)



CH4.1)  if not "WAITING FOR (ICP.Clear_To_Send)"
           then continue at (CH10.1)

CH4.2)  execute System.Wake_Up_Waiting_Primitive

CH4.3)  return to caller



CH5.1)  if not "WAITING FOR (ICP.Clear_To_Receive)"
           then continue at (CH10.1)

CH5.2)  execute System.Wake_Up_Waiting_Primitive

CH5.3)  return to caller
```

```
CH6.1)  execute System.Error_Report
                    ("Invalid Function Flagged By Peer")

CH6.2)  return to caller

.


CH7.1)  if P_STATUS indicates "PEER LOADER NOT RESIDENT"
           then continue at (CH10.1)

CH7.2)  if "DOWNLOADER NOT RESIDENT"
           then continue at (CH10.1)

CH7.3)  execute System.Activate(DOWNLOADER,DNL.1)

CH7.4)  return to caller


CH8.1)  if P_STATUS indicates "PEER DOWNLOADER NOT RESIDENT"
           then continue at (CH10.1)

CH8.2)  if "LOADER NOT RESIDENT"
           then continue at (CH10.1)

CH8.3)  if "LOADER WAITING"
           then execute System.Wake_Up_Waiting_Process
           else execute System.Activate(LOADER,LDR.8)

CH8.4)  return to caller


CH9.1)  if not "WAITING FOR (UCP.Send_Control_Block)"
           then continue at (CH10.1)

CH9.2)  execute System.Wake_Up_Waiting_Primitive

CH9.3)  return to caller


CH10.1)  execute Sublevel.Send_Control(ICP.Invalid_Function)

CH10.2)  execute System.Error_Report
                    ("Peer Function Is Invalid")

CH10.3)  return to caller
```

```
CH11.1)  if "WAITING FOR STATUS RESPONSE"
            then continue at (CH11.2)
            else if "WAITING FOR STATUS EXCHANGE"
                    then continue at (CH11.2)
                    else if "WAITING FOR LINK UP"
                            then continue at (CH11.4)
                            else continue at (CH11.6)

CH11.2)  execute Sublevel.Send_Control
                          (STATUS_RESPONSE.modifier)

CH11.3)  return to caller

CH11.4)  clear "WAITING FOR LINK UP"

CH11.5)  reset TIMER

CH11.6)  set "WAITING FOR STATUS EXCHANGE"

CH11.7)  execute Sublevel.Send_Control
                          (STATUS_RESPONSE.modifier)

CH11.8)  set TIMER = "RETRY TIME"

CH11.9)  wait {"TIME UP"}

CH11.10) if "LINK UP"
            then continue at (CH11.7)
            else continue at (Link Initializer)



CH12.1)  if "INVALID STATUS MODIFIER"
            then continue at (CH10.1)

CH12.2)  set P_STATUS = STATUS_EXCHANGE.modifier

CH12.3)  if "WAITING FOR LINK UP"
            then continue at (CH12.4)
            else if "WAITING FOR STATUS EXCHANGE"
                    then continue at (CH12.6)
                    else continue at (CH12.8)

CH12.4)  clear "WAITING FOR LINK UP"

CH12.5)  continue at (CH12.7)

CH12.6)  clear "WAITING FOR STATUS EXCHANGE"

CH12.7)  reset TIMER

CH12.8)  return to caller
```

```
CH13.1)  if "INVALID STATUS MODIFIER"
            then continue at (CH10.1)

CH13.2)  set P_STATUS = STATUS_RESPONSE.modifier

CH13.3)  execute Sublevel.Send_Control
                          (STATUS_EXCHANGE.modifier)

CH13.4)  if "WAITING FOR LINK UP"
            then continue at (CH13.5)
            else if "WAITING FOR STATUS RESPONSE"
                    then continue at (CH13.7)
                    else continue at (CH13.9)

CH13.5)  clear "WAITING FOR LINK UP"

CH13.6)  continue at (CH13.8)

CH13.7)  clear "WAITING FOR STATUS RESPONSE"

CH13.8)  reset TIMER

CH13.9)  return to caller
```

B.8 LINK INITIALIZATION PROCESS

```
Link Initializer:

1)  execute Sublevel.Enable_Communications

2)  if "LINK UP"
        then continue at (6)
        else set "WAITING FOR LINK UP"

3)  set TIMER = "RETRY TIME"

4)  wait {"TIME UP"}

5)  continue at (2)

6)  clear "WAITING FOR LINK UP"

7)  set "WAITING FOR STATUS RESPONSE"

8)  execute Sublevel.Send_Control(ICP.Status_Request)

9)  continue at (3)
```

B.9 DOWNLOAD PROTOCOL DATA STRUCTURES AND PROCESSES

LOAD CONTROL BLOCK

LOAD ADDRESS	(4 bytes)
LOAD SIZE	(4 bytes)
FRAGMENT SIZE	(4 bytes)
TRANSFER ADDRESS	(4 bytes)

LCB ACK BIT DEFINITIONS

BIT MASK	MEANING
000001	ILLEGAL LOAD ADDRESS
000002	ILLEGAL LOAD SIZE
000004	ILLEGAL FRAGMENT SIZE
000010	ILLEGAL TRANSFER ADDRESS
000020-100000	RESERVED

NOTE: The following LOADER and DOWNLOADER algorithms assume successful execution of each primitive IPLIMP function. An actual implementation should check the output status parameter for successful completion as each function terminates. Also, local variables and constants are represented using mnemonics. For instance, the two symbols Load_Control_Block_Addr and Load_Control_Block_Size are determined by the local LCB data structure for each algorithm.

```
Loader:

LDR.1)  execute Primitive.ASSOCIATE(Dummy.PARAMI,
                                    Dummy.PARAMI,
                                    Dummy.PARAMO)

LDR.2)  execute Primitive.SEND_COMMAND(UCP.Load_Request,
                                       Dummy.PARAMO)

LDR.3)  execute Primitive.SEND(Filename_Block_Addr,
                               Filename_Block_Size,
                               Timeout_Count,
                               Dummy.PARAMO)

LDR.4)  execute Primitive.DISSOCIATE(Dummy.PARAMO)

LDR.5)  set TIMER = "RESPONSE TIME"

LDR.6)  wait {"TIME UP",(UCP.Download_Request)}

LDR.7)  if "TIME UP"
           then continue at (LDR.17)

LDR.8)  if "WAITING FOR (UCP.Download_Request)"
           then reset TIMER

LDR.9)  execute Primitive.ASSOCIATE(Dummy.PARAMI,
                                    Dummy.PARAMI,
                                    Dummy.PARAMO)

LDR.10) execute Primitive.SEND_COMMAND
                         (UCP.Send_Control_Block)

LDR.11) execute Primitive.RECEIVE(Load_Control_Block_Addr,
                                  Load_Control_Block_Size,
                                  Timeout_Count,
                                  Dummy.PARAMO)

LDR.12) initialize (Fragment_Addr, Fragment_Size,
                    Fragment_Count, Last_Fragment_Size)

LDR.13) if "INITIALIZED PARAMETERS VALID"
           then continue at (LDR.18)
```

```
Loader Continued:

LDR.14) flag appropriate LCB_ACK NAK bits

LDR.15) execute Primitive.SEND(LCB_ACK_Addr,
                               LCB_ACK_Size,
                               Timeout_Count,
                               Dummy.PARAMO)

LDR.16) execute Primitive.DISSOCIATE(Dummy.PARAMO)

LDR.17) terminate Loader process

LDR.18) clear LCB_ACK NAK bits

LDR.19) execute Primitive.SEND(LCB_ACK_Addr,
                               LCB_ACK_Size,
                               Timeout_Count,
                               Dummy.PARAMO)

LDR.20) if Fragment_Count = 1
           then set Fragment_Size = Last_Fragment_Size
           else if Fragment_Count = 0
                   then continue at (LDR.17)

LDR.21) execute Primitive.RECEIVE(Fragment_Addr,
                                  Fragment_Size,
                                  Timeout_Count,
                                  Dummy.PARAMO)

LDR.22) update (Fragment_Count, Fragment_Addr)

LDR.23) continue at (LDR.20)



Downloader:

DNL.1)  execute Primitive.ASSOCIATE(Dummy.PARAMI,
                                    Dummy.PARAMI,
                                    Dummy.PARAMO)

DNL.2)  execute Primitive.SEND_COMMAND
                         (UCP.Send_Control_Block,
                          Dummy.PARAMO)

DNL.3)  execute Primitive.RECEIVE(Filename_Block_Addr,
                                  Filename_Block_Size,
                                  Timeout_Count,
                                  Dummy.PARAMO)

DNL.4)  continue at (DNL.6)
```

Downloader Continued:

```
DNL.5)  execute Primitive.ASSOCIATE(Dummy.PARAMI,
                                    Dummy.PARAMI,
                                    Dummy.PARAMO)

DNL.6)  execute Primitive.SEND_COMMAND(UCP.Download_Request,
                                       Dummy.PARAMO)

DNL.7)  initialize (Fragment_Addr, Fragment_Size,
                    Fragment_Count, Last_Fragment_Size)

DNL.8)  execute Primitive.SEND(Load_Control_Block_Addr,
                               Load_Control_Block_Size,
                               Timeout_Count,
                               Dummy.PARAMO)

DNL.9)  execute Primitive.RECEIVE(LCB_ACK_Addr,
                                  LCB_ACK_Size,
                                  Timeout_Count,
                                  Dummy.PARAMO)

DNL.10) if "LCB ACKNOWLEDGED"
           then continue at (DNL.13)

DNL.11) execute Primitive.DISSOCIATE(Dummy.PARAMO)

DNL.12) terminate Downloader process

DNL.13) if Fragment_Count = 1
           then set Fragment_Size = Last_Fragment_Size
           else if Fragment_Count = 0
                   then continue at (DNL.12)

DNL.14) execute Primitive.SEND(Fragment_Addr,
                               Fragment_Size,
                               Timeout_Count,
                               Dummy.PARAMO)

DNL.15) update (Fragment_Count, Fragment_Addr)

DNL.16) continue at (DNL.13)
```

NOTE: There are two entry points for both the LOADER and the DOWNLOADER. (LDR.1) is entered by a user on his own system to obtain a load module from a peer system. (LDR.8) is entered by IPLIMP to finish processing a download request received from a peer system that wishes to transmit a load module. (DNL.5) is entered by a user on his own system to transmit a load module to a peer system. (DNL.1) is entered by IPLIMP to finish processing a load request from a peer system that wishes to receive a load module.