1-1-1986

A Network Package for the Macintosh Using the DoD Internet Protocols

Mark Sherman
Dartmouth College

A NETWORK PACKAGE FOR THE MACINTOSH USING THE DoD INTERNET PROTOCOLS

Mark Sherman

Technical Report PCS-TR86-124

A Network Package for the Macintosh using the DoD Internet Protocols

Mark Sherman
Computer Network Laboratory
Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755

(603) 646-2415
mss@Dartmouth.CSNet

Table of Contents

List of Figures

1. Introduction

This report describes the initial construction of a set of IP protocol packages for the Macintosh. The report includes a description of the packages' programs and libraries, along with some maintenance information.

The construction of the protocol package is part of a larger project to connect Macintoshes with networks that support TCP/IP protocols, for example, an Ethernet that runs between Suns and Vaxes. The project is broken down into two phases: providing a gateway[1] between Applebus and another network medium (such as Ethernet), and providing the necessary protocol support on the Macintoshes. Because several groups seemed to be investigating the first phase (for example, Bill Croft at Stanford University's Medical Exchange and Lawrence Butcher at Carnegie-Mellon University's Computer Science Department) and because the first phase seemed most subject to change (because of its dependency on the final hardware available for the gateway), I chose to work on the second phase: a set of protocol packages to be used on the Macintosh. The initial pieces of these packages were constructed at Carnegie-Mellon University during the Summer of 1984 under a grant from the Apple Computer Company. At the time of writing this report, the software is still under development. Although distributions are available, the software should be considered experimental and not production quality.

The design of the system was taken from the TCP/IP implementation written for the IBM PC (by various authors) that is distributed by MIT [17, 18, 19]. The lower and higher levels of the design were reimplemented from scratch. The middle layers of the design were transliterated from the C sources that MIT distributed, into Pascal sources for the compiler that runs under the Lisa Workshop. Most sources are available to the general public (see Section 5).

In the description that follows, I assume that the reader has access to or has read the documents describing the following protocols and systems:

ABLAP — Applebus Link-Layer Protocol, [1]. This protocol describes the broadcast scheme used by the Applebus and defines node numbers within an Applebus.

DDP — Datagram Delivery Protocol, Applebus unreliable (best efforts) datagram protocol, [4]. This protocol describes the datagram mechanism provided by Apple. This protocol also defines network numbers and socket numbers for Applebus. Most other protocols defined by Apple are built using the DDP layer.

IP — Internet Protocol, [10]. This protocol, developed for the Department of Defense, describes how packets are formatted, how IP addresses are interpreted, and various performance options that can be specified for the underlying link layer. Header integrity is provided by a checksum; there is no

[1] Some use the terminology "bridge", "router", "interface" or "link".

provision for data integrity.

ICMP — Internet Control Message Protocol, [11]. This protocol, which is inextricably related to IP, describes the control messages that should be sent during various error conditions when using the IP protocol.

GGP — Gateway to Gateway Protocol, [5]. This protocol is used between internet gateways for updating table information and testing.

ARP — Address Resolution Protocol, [6]. This protocol is used by a host to translate an IP address into a local hardware address, for example, from IP to Applebus.

UDP — User [sic] Datagram Protocol, [8]. This protocol provides an unreliable datagram service on top of IP, and defines sockets. Header and data have independent checksums. The delivery is unreliable since there is no retrial if a datagram is corrupted or not received.

Time Service — Time Service Protocol, [13]. This protocol provides a way to ask a server machine for the current time in seconds since midnight, Jan 1, 1900, GMT. It usually uses UDP.

INS — Internet Name Service Protocol, [7]. This protocol provides a way to ask a server machine to translate a host name (string) into an IP address (integer). It uses UDP.

Logging Service — Log Service Protocol, [3]. This protocol provides a way for UDP processes to log their activities on a remote host. It uses UDP.

TFTP — Trivial File Transfer Protocol, [20]. This protocol provides a reliable file transfer protocol, usually using UDP. It has very simple flow control and transfer modes.

TCP — Transmission Control Protocol, [12]. This protocol provides a reliable data stream protocol using IP. It has sophisticated flow control and transfer modes, and corresponding amounts of overhead.

FTP — File Transfer Protocol, [9]. This protocol provides reliable file transfer using TCP.

Telnet — Remote Terminal Protocol, [14]. This protocol provides a reliable, remote terminal protocol using TCP.

PCIP — IBM PC Internet Protocol Package, [17, 18]. This package provides minimal implementations of the Internet protocols above (except FTP) for the IBM PC. It was written by many authors and uses designs from many implementations of the protocols. The Macintosh design was based on this package. Some of the code from this package was transliterated from C into Pascal for use on the Macintosh.

Inside Macintosh — This document [2] gives a description of the Macintosh User Interface Toolbox and the Macintosh Operating System.

Workshop — This document [21] gives a description of the Pascal language used for programming the Macintosh, the resource compiler for Macintosh resources and the development environment for creating Pascal programs and libraries.

This document is not intended to be a complete guide to DoD's Internet protocols, to the Macintosh, or even to the network programs for the Macintosh. This document is intended to be used by someone who is familiar with most of the material listed above. Because much of the code written for the Macintosh was based on the design of the IBM PC system, the discussion is usually one of comparing the two implementations and not providing all of the details for the Macintosh implementation. Where a comparison is being made, the IBM PC system will be referred to as *PCIP* and the Macintosh system will be referred to as *MacIP*. Where a new design has been added to system, more details will be given.

The next section (Section 2) gives both user and maintainer information for the Macintosh programs that are provided on the Macintosh disk: *TIME*, *TFTP* and *CUSTOMIZE*. The following section (Section 3) gives maintainer information about the libraries in MacIP. Section 4 gives an overview of some important design differences between PCIP and MacIP. Following the design discussion, Section 5 describes how copies of MacIP can be obtained. The two appendices give the file names of all sources and the "magic numbers" defined in MacIP that are visible to interacting programs.

2. User Programs

There are three programs written for the Macintosh: *TFTP*, *TIME* and *CUSTOMIZE*. For each program, a brief description is given on how to use the program, followed by a section giving maintenance information.

2.1. TFTP

2.1.1. User Information

The TFTP program uses the trivial file transfer protocol [20] to implement reliable file transfer between two Macintoshes (or between a Macintosh and any other system that supports TFTP). The program can act as either a server or a user, but not both at the same time. The choice is made by the use of a pull down menu (see Figure 2-1). When the program is acting as a server, each request is displayed as it is processed (see Figure 2-2). When the program is acting as a user, a dialog is provided for specifying transfer characteristics. During the file transfer, statistics and diagnostics describing the transfer are displayed for the operator[2] (see Figure 2-3).

---

[2] "Operator" denotes the user of the TFTP program. The word "user" denotes, in this discussion, "user mode of TFTP".

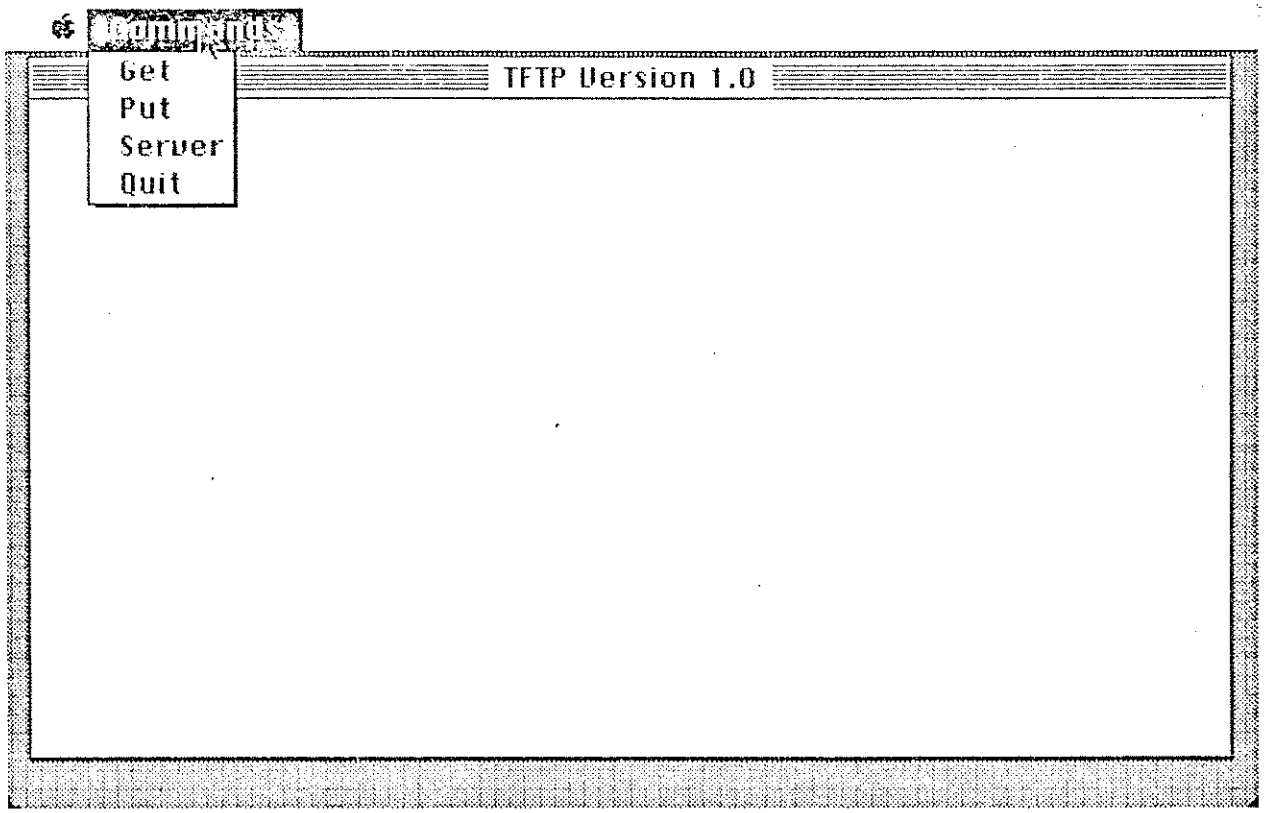

Figure 2-1: TFTP Command Menu

When the TFTP program is operating in server mode, the operator can only stop the server by the *QUIT* command in the command menu. All other actions performed by the server are in response to requests on the network. When the TFTP program is operating in user mode, and the operator requests a file transfer, the program provides a dialog by which an operator can choose the file part, transfer mode, transfer direction, file names and host name (see Figure 2-4). Each of these choices is described below.

The "file part" selection is provided because Macintosh files are not monolithic sequences of data. Instead, each file has three parts: finder information, a data fork and a resource fork. The finder information is similar to directory information on other systems and includes type information about files (Macintosh files are typed). The data fork typically holds application defined information, such as the text of a MacWrite document, and corresponds to a data file on most systems. The resource fork holds programs, icon definitions, and other Macintosh resources, and corresponds to executable files on other systems. When retrieving or sending a file with a non-Macintosh system, usually the data fork is being exchanged (such as text created with a text

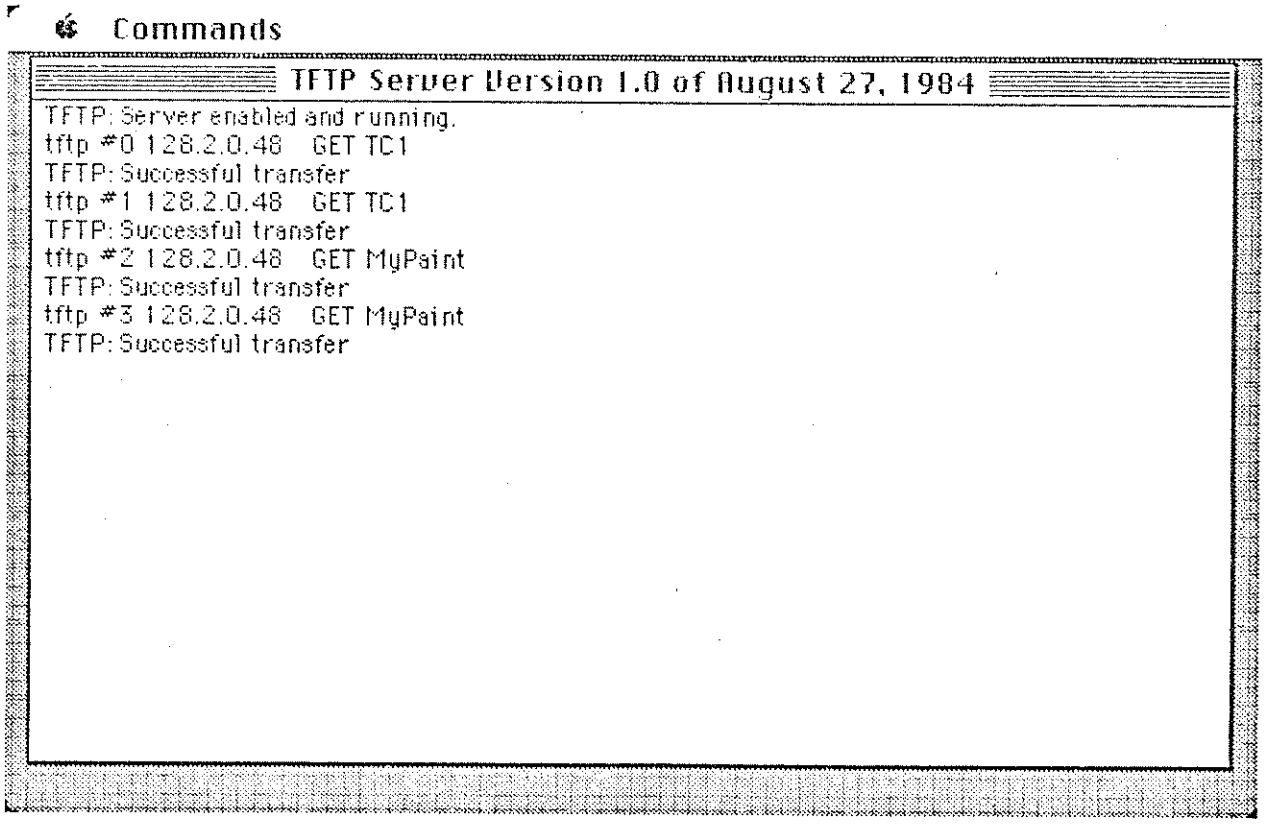

Figure 2-2: Output from TFTP Server

editor or with MacWrite). When exchanging files between two Macintoshes, all three parts of a file need to be moved. However, the operator may wish to control explicitly which pieces of a file are transferred. The file part selection provides the operator some control over the parts of the file to be transferred. The *Whole File* selection will cause all three parts to be moved, while the *Data Fork Only* selection will cause only the data fork to be transferred.

The available transfer modes are *Ascii*, *Octet*, *Image* and *Macintosh*. In all four modes, bytes are treated as eights of data without further interpretation. For example, carriage returns, line feeds and tabs are not added, deleted or transformed into other character sequences. In *Ascii*, *Octet* and *Image* modes, the data fork of a file is transferred as per the protocol specified in RFC783 [20]. However, when finder information is transferred, the TFTP program will append the characters ".info" to the remote file name. Similarly, when resource fork data are transferred, the TFTP program will append ".rsrc" to the remote file name. For example, if *TFTP* is requested to place the local file *LF* onto another machine with the name *DF* using whole-file, ascii-transfer mode, then the data fork of *LF* will be transferred to the other machine as *DF*, the resource fork will be transferred to the other machine as *DF.rsrc*, and the finder information will

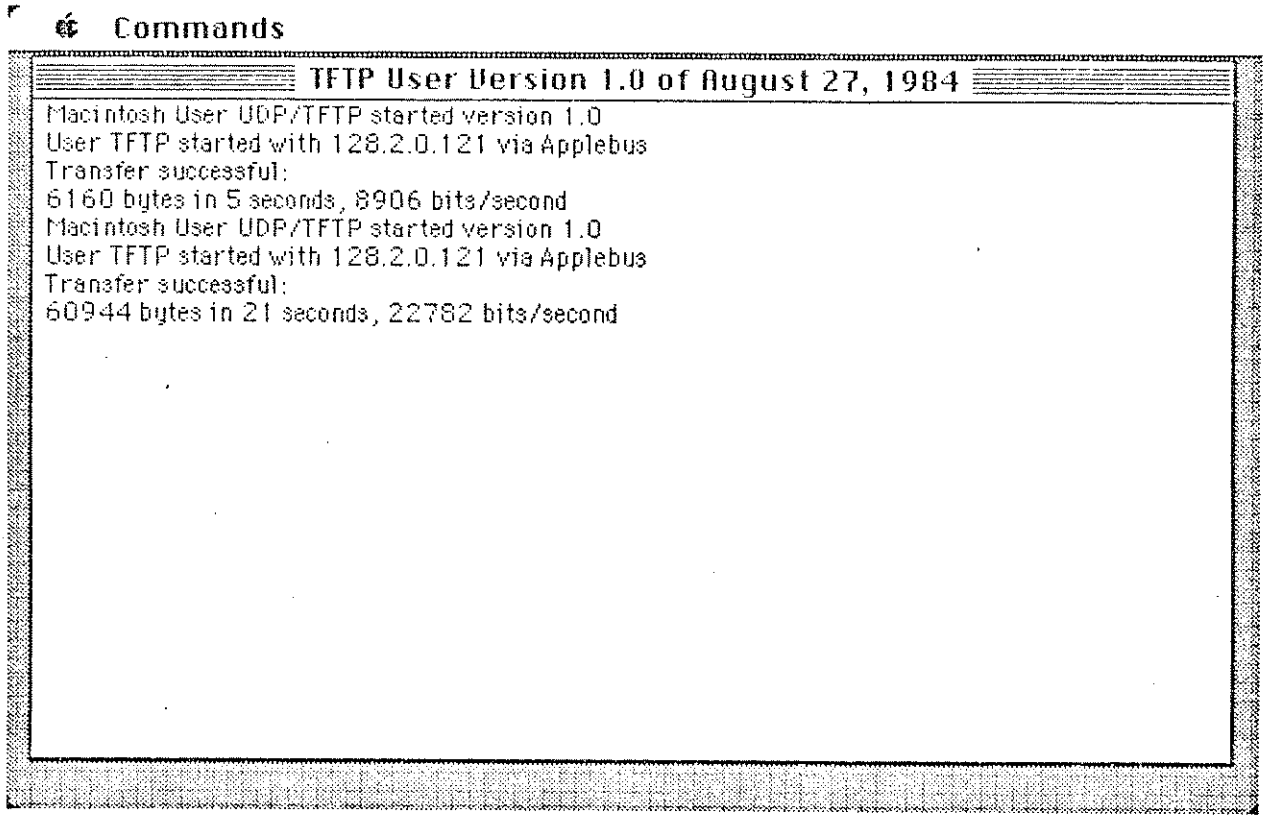

Figure 2-3: Output from TFTP User

be transferred to the other machine as *DF.info*.

Macintosh mode uses a slightly different protocol for transferring files. According to RFC783, request packets contain an opcode (either Read Request or Write Request) followed by a C-style string[3] specifying the filename followed by a C-style string that specifies the transfer mode. The specification given in RFC783 is shown below:

2 bytes	string	1 byte	string	1 byte
Opcode	Filename	0	Mode	0

When the transfer mode is *Macintosh*, a C-style string follows the transfer mode which specifies which part of the Macintosh file is being transferred: 'data', 'resource' or 'finder', as specified below:

[3] A string in C is a sequence of non-zero bytes terminated by a 0 byte.

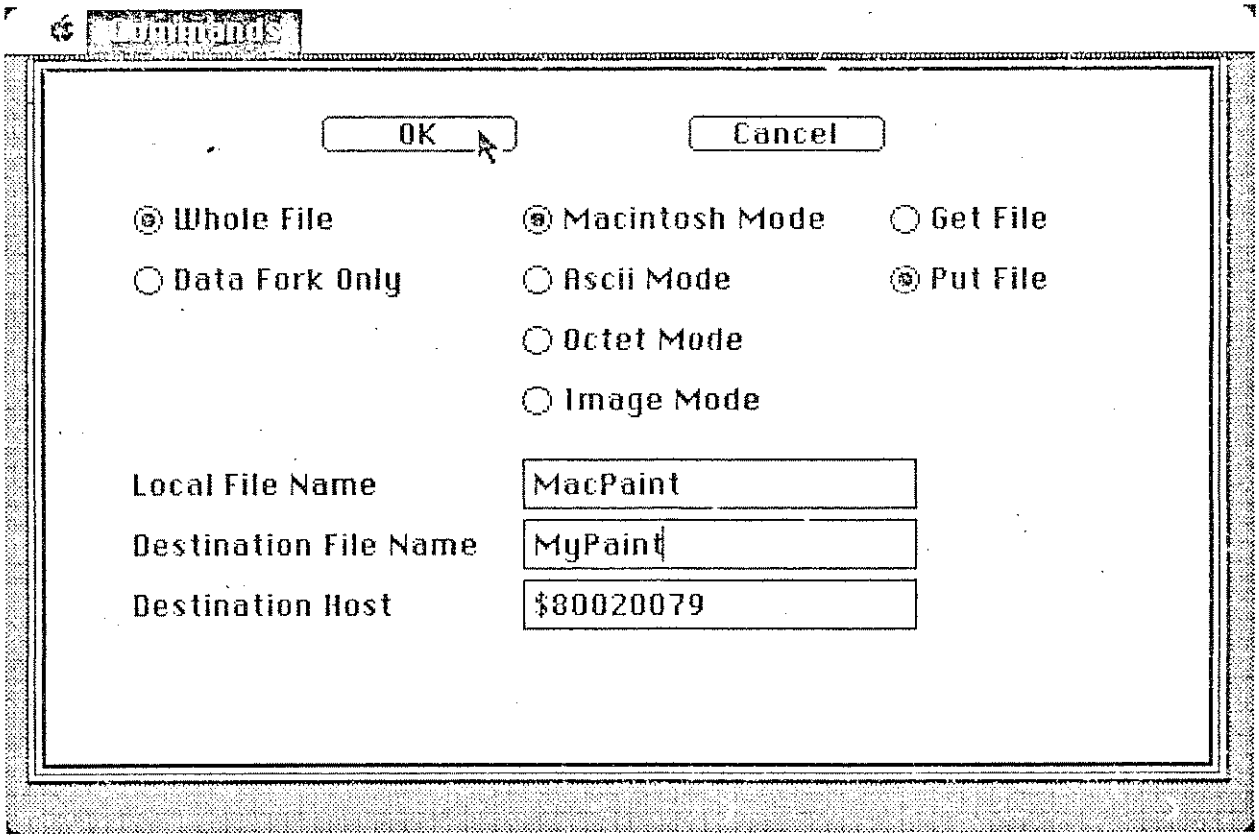

Figure 2-4: TFTP User Dialog

2 bytes	string	1 byte	string	1 byte	string	1 byte
Opcode	Filename	0	Mode	0	File Part	0

When the operator specifies that *Macintosh* transfer mode should be used, the file part that is being transferred will be identified by the third string in the request. Note: In all transfer modes, a whole file transfer makes three TFTP-level file transfers (one for each part of the file), while a data-fork-only file transfer makes only one TFTP-level file transfer.

Because I expect that whole files will be transferred in *Macintosh* mode while only data forks will be transferred in other modes, the selection of *Ascii*, *Octet* or *Image* mode will cause *Data Fork Only* to be selected as the file part. Similarly, choosing *Macintosh* mode will cause *Whole File* to be selected as the file part. However, the operator may select some other file part after selecting the desired transfer mode.

The same file transfer dialog is presented to the operator regardless of whether a *Get* or *Put*

command is selected from the *Command* menu. However, the dialog will have the requested transfer direction filled in, The operator may change the direction of the file transfer by clicking the appropriate button.

Because the name service software is not debugged, the current version of *TFTP* requires that the host be given as an IP address, which is an integer. Both hexadecimal and decimal number notation is supported. (A hexadecimal number is preceded by a $.)

The file transfer does not take place until the *OK* button is clicked; clicking the *Cancel* button cancels the request.

The performance of the file transfer is hard to measure. The following table was constructed by making copies of the *MacPaint* program and a sample picture from an internal Sony disk onto the same disk, from an internal Sony disk onto an external Sony disk, and across the network between two internal Sony disks. All times are seconds as measured on my watch; the times represent the interval from when I released the mouse to start the action until the action was completed. The numbers are very soft, but give some idea of the times involved.

Test	Copying MacPaint	Copying a Picture
Same Disk	12	7
External Disk	25	17
TFTP	25	8

The *MacPaint* program contains about 60.9 kilobytes of data, while the picture contains about 4.6 kilobytes of data. A time of 25 seconds for copying MacPaint represents a data transfer rate of about 20 kilobits/second, or about 10% of the Applebus bandwidth.

2.1.2. Maintenance Information

The Pascal source file for the TFTP program is APPL-TFTP.TEXT; the resource file is APPL-TFTPR.TEXT. The same resource file can be used when compiling the TFTP program in debugging mode.[4] After all source files have been compiled, they can be linked by using the command file T-LINK.TEXT with the argument APPL-TFTP. The resulting resource file will be named APPL-TFTP.RSRC.

There are some small changes that should be made to the TFTP program which would help operators.

First, the TFTP program should use the parts of the name server that do not access the network to allow operators to express IP addresses in any of the standard formats. In particular, the

[4] All source files contain a compile-time variable DEBUG that controls the enabling of debugging features.

TFTP program should use the routines that parse and translate various standard IP network number representations (#<digits>, <net>.<subnet>.<host>.<host>, <net>,<subnet>,<host>,<host>) into integers. However, one should be careful about implementing the name resolution protocol, since RFC901 [15] specifically warns that the name protocol is ambiguous and under revision.

Second, there should be some way of using the standard file system package for getting file names. When sending a file, the operator should be able to pick the file using the standard file menu, which allows, for example, the operator to pick a file from another disk. Similarly, when getting a file, the operator should be able to use another standard file menu for defining a new file (like the *Open* and *Save-as* commands in *MacWrite*). However, all of the other fields in the dialog must still be present: transfer mode, transfer direction, file part, remote file and host names, and confirmation buttons.

Third, the IP addresses of the user and the server should be displayed in the window's title.

Fourth, all of prompts and menu entries that use the word *Destination* should be changed to *Remote*, a more accurate designation. Thus the user dialog would ask for the local file and and remote file name, instead of the local and destination file name.

Two additional problems with the TFTP program may require a bit more work.

First, there is a definite space problem on the 128K Macintosh. When many debugging flags are set to true during compilation, a significant amount of additional code is generated, and I have, on occasion, run out of memory. Using the debugger at the same time exacerbates the situation. Using desk accessories, even with debugging disabled, seems to use too much space. However, I have yet to run out of space when the debugging switches are set to false and the desk accessories are disabled.

Second, the TFTP program makes an incorrect separation of concerns when modifying file names during file transfer in modes other than *Macintosh*. Currently, the interface presented by the TFTP library to a program wishing to send a file is:

```
FUNCTION tftpuse(fhost: in_name; fname: StringPtr; rmfile: StringPtr;
         dir: integer; mode: integer; FPart: tf_FilePart; size:LongInt;
         VAR deltat:LongInt): LongInt;
```

The programmer specifies the part of a file is to be transferred and the remote file name through appropriate parameters. The current implementation has the remote file name being modified by the TFTP library during the construction of the request packet. This is a poor design decision; there is no need for the request-packet algorithm to process details of the remote file name. The remote file name *rmfile* given in the *tftpuse* call should be used without transformation. The caller of *tftpuse* should make whatever file name transformations are appropriate, since the

different hosts could have wildly different file name conventions or file name lengths. By obvious corollary, the scheme implemented in the TFTP library will not work if the remote file name is too long (the exact restriction depends on the remote host -- it is the case that the implemented file naming conventions will fail for a remote Macintosh if the remote file name is at least 250 characters long). Therefore, the file-name transformation should be done in the TFTP program and not in the TFTP library routines.

Finally, the TFTP library used by the TFTP program is not bug free. Two known bugs are 1) that all transferred files appear to be placed in the upper left hand corner of the disk's window, and not in an empty place, and 2) under certain circumstances, a user TFTP will freeze completely (mouse dead, interrupt button dead, etc.) after sending a third file to a server. Although incredibly inconvenient, you are recommended to send only two files at a time with the current release of TFTP (though you should be able to receive any number). Section 3.4.6 contains more details on these problems.

2.2. TIME

2.2.1. User Information

The program *TIME* illustrates the time server and user code. With changes, it could be used to remotely set a Macintosh clock.

Figure 2-5 shows the commands available for this program. The program can act as a time server or as a time user, but not both. When the *Start Server* command is selected, the Macintosh will process and answer requests for the time. When in server mode, the program will also try to display the system clock by using the *Alarm Clock* desk accessory (see Figure 2-6).

When the TIME program acts as a user, it will query the time server every time the *Get Time* command is selected. The retrieved time will be printed on the screen (see Figure 2-7).

2.2.2. Maintenance Information

The Pascal source file for the TIME program is APPL-TIMES.TEXT; the resource file is APPL-TIMESR.TEXT. The same resource file can be used when compiling the TIME program in debugging mode. After all source files have been compiled, they can be linked by using the command file T-LINK.TEXT with the argument APPL-TIMES. The resulting resource file will be named APPL-TIMES.RSRC.

According to the Time Service Protocol [13], the reported time is the number of seconds since Jan. 1, 1900, midnight GMT. The Macintosh provides local time (in seconds) relative to Jan. 1,

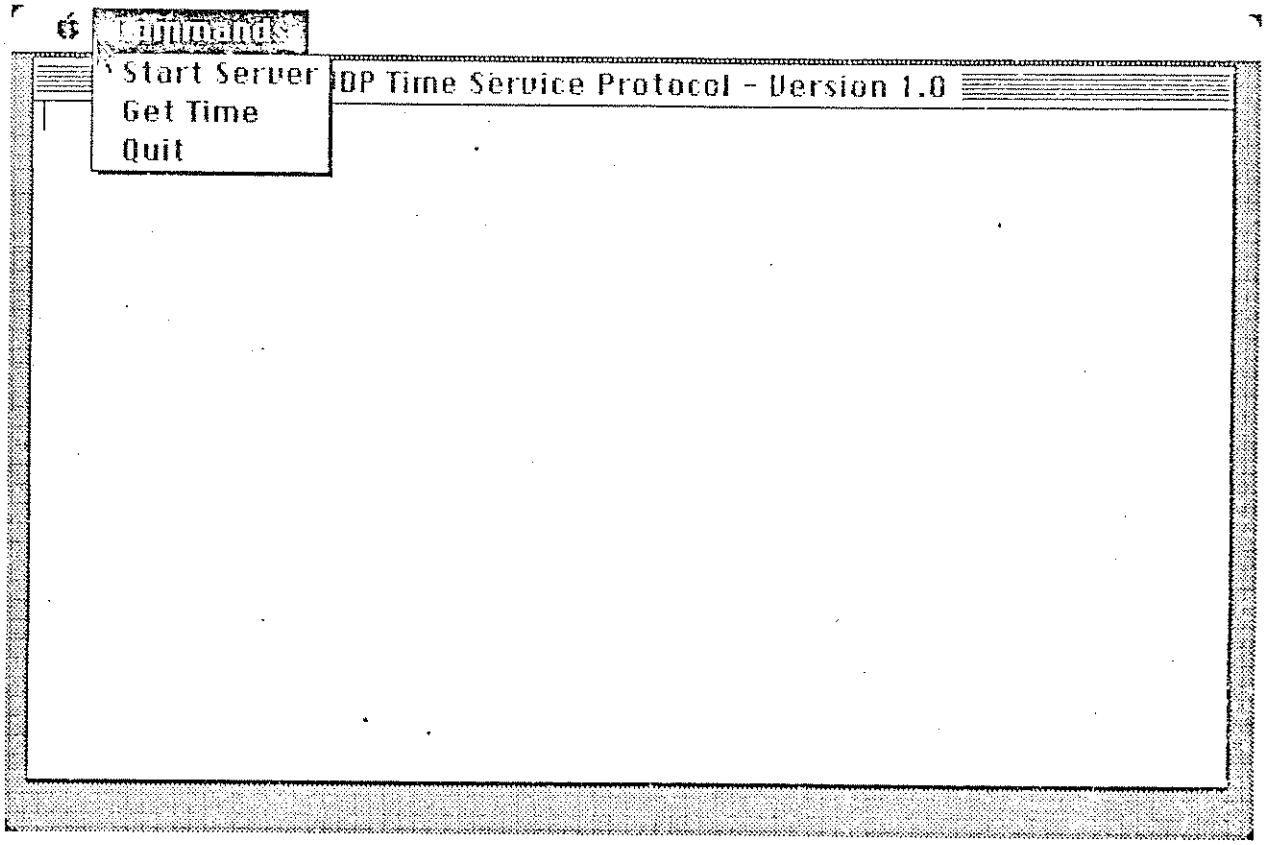

Figure 2-5: Time Server Command Menu

1904, midnight. There is a conversion factor in the program to convert between 1900 GMT and 1904 EDT which appears to work, but since the implementation only interacts with itself, any miscalculation of the conversion offset would not be detected. There are some sample values given RFC868 which should be sent to the implementation to test its correctness, but I have not done so.

Another problem is that the conversion factor must be calculated at run time, since I do not know a way to calculate a long integer value at compile time. If a way can be found to change this, the conversion factor should be a constant exported from the UDP package instead of calculated explicitly by the time server.

2.3. CUSTOMIZE

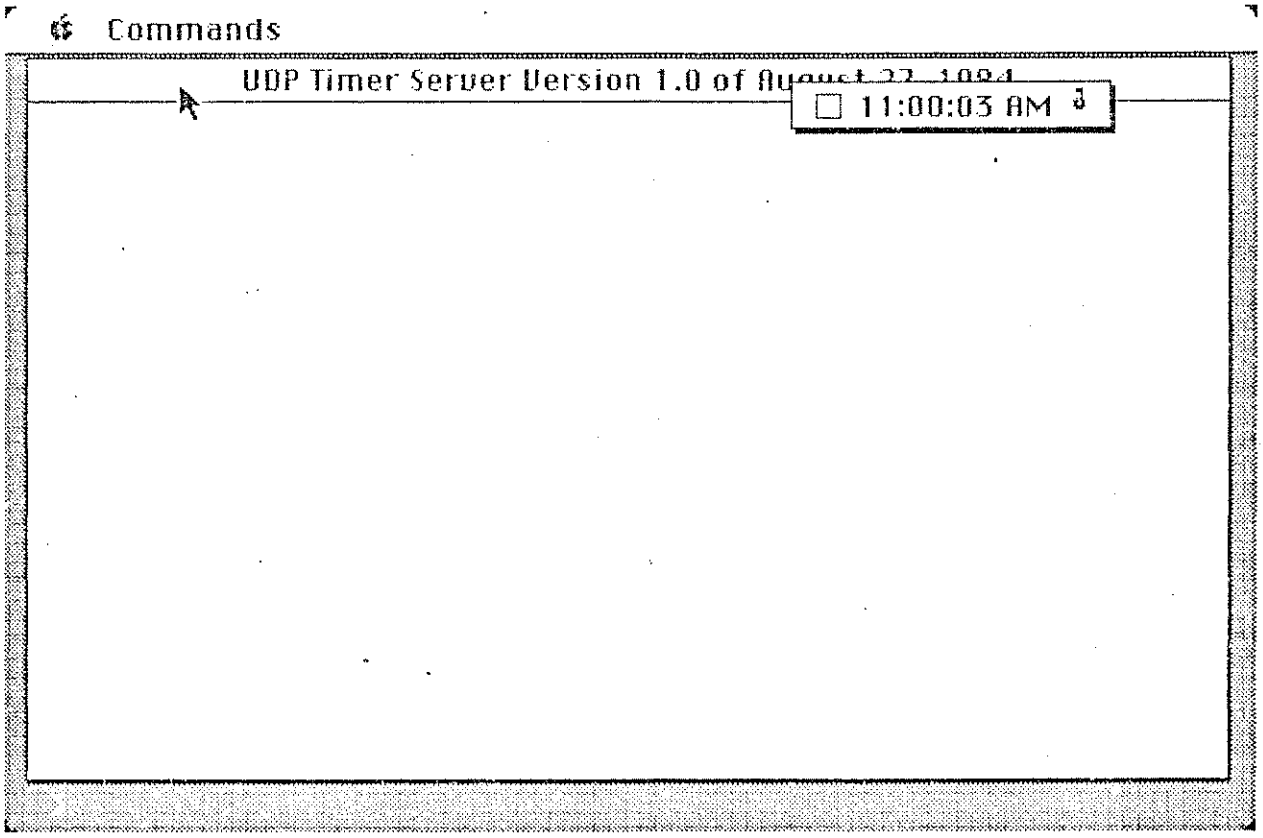

Figure 2-6: Time Server Running

2.3.1. User Information

There are a variety of parameters that are used by the network packages to control the configuration of buffers, host numbers, sockets and names. All programs that need this information read it from a file called *Customization Values*. The program which creates and edits this file is called *CUSTOMIZE*.

When launched, the CUSTOMIZE program presents the user with a dialog which is used for showing and editing the values in the customization file (see Figure 2-8). The user may alter six parameters: the local IP address, the IP address of the (IP) gateway, the IP address of the Time Server, the IP address of the Name Server, the User name at this IP address and whether to alter the local IP address by some function on the IP address and the Applebus Node address. In the original customization file for PCIP, there are many, many parameters that can be set. The current network implementation freezes some of the parameters at compile time as constants in the appropriate files, and eliminates many of the other customization features.

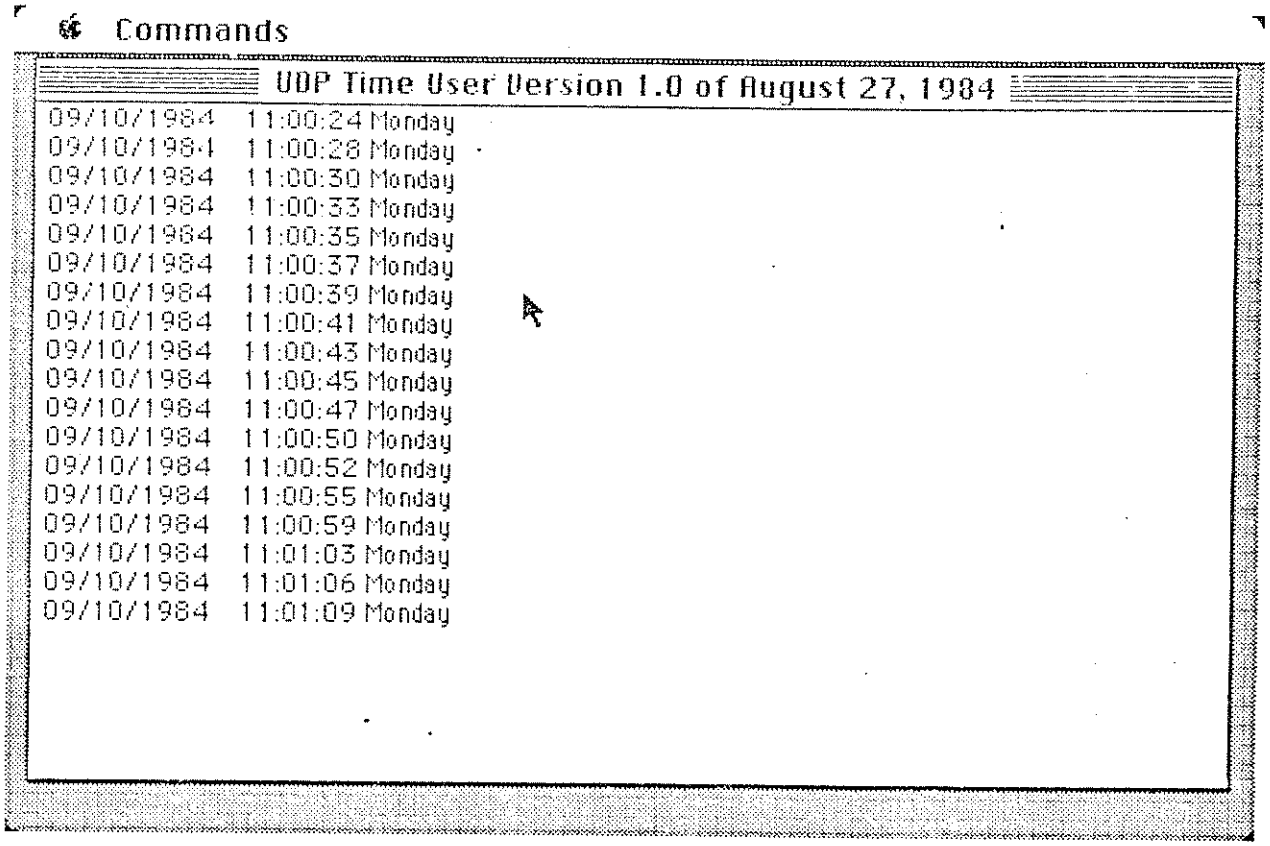

Figure 2-7: Time User Running

Because name service is not available, all IP addresses must be entered as integers. When creating a customization file, default values are selected for these addresses as being within the CMU class B network.

The user may specify that the local IP address be a function of the current Applebus Node number. The current transformation of the IP name that is available is quite simple: the least significant byte of the IP address is set to the current Applebus Node number. This replacement occurs when the customization file is read by the program using it, however, the customization program will attempt to show that value in the *Local IP Address* field.

When the user clicks *OK*, and then confirms the wish to leave the customizer, the program checks the values to make sure that they have some meaning. Any diagnostic messages are provided at the bottom of the dialog box. The user may also cancel any changes and exit the program.

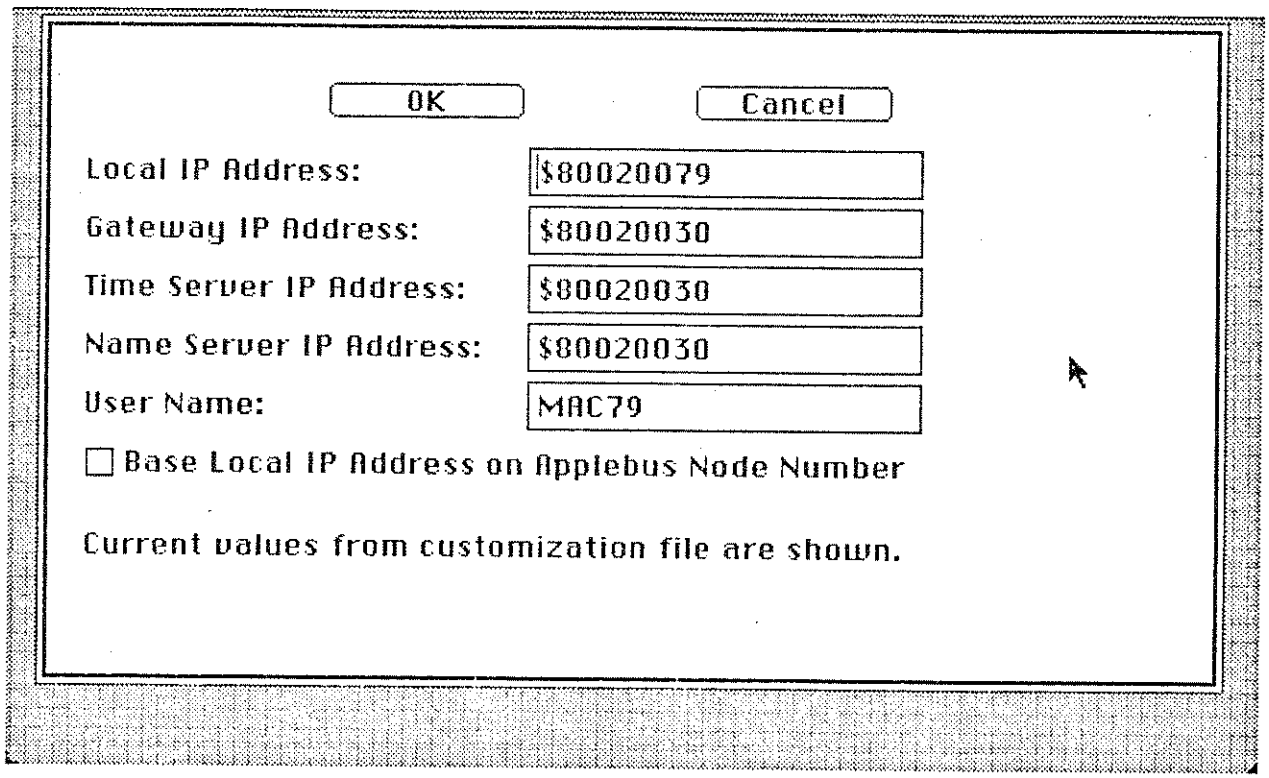

Figure 2-8: Customizer Dialog

2.3.2. Maintenance Information

The Pascal source file for the CUSTOMIZE program is APPL-CUST.TEXT; the resource file is APPL-CUSTR.TEXT. The same resource file can be used when compiling the CUSTOMIZE program in debugging mode. After all source files have been compiled, they can be linked by using the command file T-LINK.TEXT with the argument APPL-CUST. The resulting resource file will be named APPL-CUST.RSRC.

The customizer should work with a family of dialogs, one for each kind of customization that is to be performed. The user would pick the correct customization by a command in a pull-down menu. A more advanced version might also perform name resolution on the input IP values, and allow for multiple customization files.

3. Available Libraries

Several collections of packages or libraries were written for the Macintosh. They can be broken down into three sets: basic operating system support, basic C language support and protocol packages. Some common points for all of the library packages are discussed below, after which each library group is described.

3.1. Common Properties of Libraries

Since many of the libraries were transliterated from C, a number of common transformations are present in the files. These are

- Use of explicit pointer calculations in place of "&" operators.
- Use of explicit pointer calculations in place of autoincrement and indexing operations.
- Use of explicit initialization of global variables.
- Use of new pointer types to replace the anonymous pointer types specified by the "*" type constructor.
- Use of the new transput (io) package in place of the "printf" calls.

The use of explicit pointer calculations in place of "&" operators was motivated by the inability of Lisa Pascal to process "@" operators for packed records. Because packed records best matched C structures, packed records were used almost exclusively in place of C structures. When an address of a field was needed, the C program usually used the & operator; the Pascal version contains an explicit pointer calculation. As an example, a typical transformation is from

```
data = &p->buf
```

to

```
CONST BuffOffset = 4;
...
data := { @p^.buf } POINTER(ORD4(p) + BuffOffset);
```

The use of pointer calculation in place of autoincrement and indexing operations was caused by the lack of pointer arithmetic in Pascal. Thus C statements such as

```
data[index] = 0;
data ++;
```

were translated into

```
RawPtr := POINTER(ORD4(data) + index);
RawPtr ^ := 0;
data := POINTER(ORD4(data) + 1);
```

One should note that this transformation is required only because PCIP was written using a certain style of C programming. If one were starting in Pascal, one would use a different approach in many of these circumstances and would not require as much direct pointer

manipulation. For example, one could make the type of the variable be a pointer to a packed array and then use Pascal indexing (and increment the index, not the pointer). I decided that a direct mimicking of the C code was a less-risk approach than rewriting parts of the programs in a Pascal paradigm instead of a C paradigm.

Throughout the C libraries, the global variables are initialized where they are declared, resulting in the correct values being loaded when the program is loaded. The Pascal for the Macintosh does not provide this kind of initialization. Therefore, all initialization of global variables occurs explicitly in the various *init* routines of each library.

A common practice in C is to declare pointer variables by a syntax which specifies the type to be pointed-at by the variable and not by creating a new pointer type. For example, a typical C pointer variable declaration is

```
integer * Data;
integer * Data2;

Data = Data2;            /* Legal in C */
```

A strict conversion to Pascal would result in two data types, as shown below:

```
Data: ^ Integer;
Data2: ^ Integer;

Data := Data2;           { Illegal in Pascal }
```

This transformation creates two incompatible types in the Pascal program. Therefore, most data structures in the Pascal programs have an additional type declaration for a pointer to that data structure. The additional type declaration is used in place of the anonymous pointer types of C.

3.2. Operating System Packages

Three operating system packages are provided: a queuing package, a tasking package and a timer package. Each is described in turn.

3.2.1. Queuing Package

The queuing package can be found in NET-Q_LIB.TEXT. This package provides a queue manipulation facility, providing both atomic (uninterruptable) and pedestrian queue operations.

Most of the queue manipulation routines are a straight forward transliteration of the C code. There are four significant differences.

First, the atomic versions of the queue operations in PCIP manipulated interrupts to guarantee that they were atomic. Because Macintosh tasks run synchronously, and in particular, because the network packet retrieval procedures are synchronized with the protocol demultiplexers (by the

tasking system), there was no need to disable or reenable interrupts in the queue package. Hence the procedures that were responsible for enabling and disabling interrupts were made into null procedures.

Second, the original C-language list-manipulation routines frequently cast a value in two way: as a pointer to a queue element, and as a pointer to a pointer to a queue element. This could be done because the first field of a queue element is a pointer to the next queue element. The dual casting took place when a procedure was comparing the "current" queue element with the "next" queue element. To achieve this double meaning of a single pointer, I used a variant record, one variant being a single indirection pointer, the other being a double indirection pointer. Assuming that the pointers would be overlayed in memory, the variant record effectively allows the kind of casting used in the C version.

Third, the original C-language routines were able to generate the addresses of various parts of the the queue element record by use the of the "&" operator. The equivalent operator in Lisa Pascal is "@", which cannot be used with packed records (as all C structures are). Therefore, I explicitly calculated pointer values from the beginning of queue elements rather than using the "@" notation. Fortunately, most of these calculations are safe since they refer to the beginning of the record and will always have an offset of 0.

Fourth, the original C-language routines assumed that queue headers would come properly initialized, perhaps taking advantage of memory allocation convention which zeros out newly allocated storage. Such an assumption would make all pointer values in the queue header 0 (which is NIL). No such automatic clearing of memory is guaranteed by the Macintosh, so a new procedure, *q_init*, was added. This procedure must be called for each new queue before that queue is used. For example, the allocator for the free packet queue calls *q_init* before creating the free packets.

For a long-term measure, it may be possible to use the queue package provided in the Macintosh Operating System, since both MacIP and the Macintosh Operating System use the first 4 bytes as a link field of queue elements. However, this design would require changing the definitions of other fields in the queue element to accommodate the element-type field in Macintosh queue elements, and require changing the network system's queue headers, which differ from the Macintosh's queue headers.

3.2.2. Tasking Package

The tasking package can be found in two files: NET-TASK_LIB.TEXT and NET-TASK_ASM.TEXT. This package provides non-peremptory tasks for the Macintosh. Tasks are scheduled using a round-robin algorithm.

The design is similar to the PCIP tasking package. Each task has its own stack, which is allocated when the task is created. Because Pascal has a different procedure and type structure than C, tasks have a more restricted form. In particular, a task must be a global procedure (outermost lexical level and not a function) with a single pointer parameter. Actually, the parameter can be pointer sized (a LongInt or a VAR parameter), but for definition purposes, a pointer value is passed to the procedure when it is started as a task. The procedure must never return; a task finishes by calling the procedure *tk_exit*. Otherwise, a procedure used as a task may perform the usual operations of a procedure: it may define and call local procedures, it may call other global procedures, it may declare local variables, it may access its parameter and it may manipulate global variables.

A task blocks by calling the procedure *tk_block*, which freezes the task's stack in a predictable state. At some point, another call of *tk_block* will cause the blocked task to be restarted, which to the blocked task looks as if the call to *tk_block* finished. Therefore, the predictable state of the stack must appear to be in the middle of call to *tk_block*. When a task is forked, a fake stack image is created that makes the stack appear as if a procedure had called *tk_block*, after which a call to the procedure which defines the forked task will take place. This stack creation is performed by the assembly language routine *tk_frame*. The stack switching that is performed when stopping one task and starting another is implemented by the assembly language routine *tk_swtch*. Both routines must follow the same conventions for leaving the stack in a predictable state.

In addition to a stack, a task control block is allocated for each stack. The task control block holds a description of the task: the task's name, size and so forth. With one exception, the task control block sits at the end of the stack (low memory -- the stack grows from high memory to low memory). Thus if a task goes beyond its stack limit, the result will be that it will first overwrite its own task control block. Since the task control block of the currently executing task is used for scheduling the next task, the problems of a corrupted task control block should happen immediately and not when a later task is finally scheduled to execute. As an added safety measure, the scheduler checks task control blocks for corruption during scheduling. Many of the problems that I found while debugging the system came from a task using more than its allocated stack space, hence I think the safety checks are worth their cost during scheduling. Because of the possible problems caused by stacks running into one another, the procedure which checks for stack overflow, *Checktask*, is exported from the tasking package for use by other procedures. There is also a procedure *GetSP* which returns the current value of the stack pointer, but unlike *CheckTask*, *GetSP* is not exported from the tasking package.

I commented that one task control block does not reside at the end of its stack: the task control block for the main program. This task control block is allocated as a global variable for

historical reasons. It should be allocated at the end of its stack like the other tasks.

The stacks for tasks in PCIP came from the heap. When a task finished executing, the storage was freed and returned to the heap. On the Macintosh, there is a "stack sniffer" that checks to see if the stack pointer (register A7) refers to a location in the heap. If so, the Macintosh will display a deep shit alert box with error code 28. One can either disable the stack sniffer or allocate space for the task stacks on the user stack. I chose the later, thus the task forking routine and the task initialization routine keep track of a high water mark of the user stack and allocate new stack space starting at that location.[5] This works fine except when a task terminates, since there is no easy way to return stack space. In the current implementation, released tasks have their stack space collected on a list and new tasks first try to get stack space from that list (using a first fit algorithm) before moving the high water mark. I believe this is an adequate technique, since the tasks in MacIP that terminate are those that are associated with a connection that is about to be closed. When a new connection is made, the same task (with same stack requirement) is forked again, hence the first fit algorithm should effectively reuse the just-released space for the newly spawned task.

I know of one possible performance problem with the tasking package: the consistent state of the stack of an inactive task may store too much information. For example, register A6 (global variable pointer) is saved even though it never changes between tasks. There may be other redundant information in the stack as well.

3.2.3. Timer Package

The sources for the timer package can be found in NET-TIMER_LIB.TEXT. The timer package is a straight forward transliteration of the C version of the timer package. There are four significant changes.

First, the same double-indirection pointer vs single-indirection pointer problem found in the queue package is present, with the same solution applied.

Second, PCIP assumes that the clock ticks at 18 beats a second, while the Macintosh clock ticks at 60 beats a second. Therefore I changed the symbolic constants as necessary.

Third, PCIP assumes that a timer is an external device that will generate an interrupt (relayed by a Unix-like Signal call) when the timer counts down to 0. However, implementation of the network package is synchronous. To perform synchronization of the alarm with the rest of the system, three global variables are used: *AlarmValue*, *TimerEnabled* and *tm_task*.

---

[5] The Pascal forking and initialization routines use two assembly language programs, stk_init and stk_alloc, to perform the actual stack manipulations.

TimerEnabled indicates if the alarm value is meaningful. *AlarmValue* holds the time at which the alarm should ring, as set by the timer package; this value is compared against the value returned by *TickCount*. The *tm_task* variable is a reference to the task that controls all timers.

Each time a null event is returned by *GetNextEvent*, *TimerEnabled* is checked, and if set, then the time as provided by *TickCount* is compared against *AlarmValue*. If the alarm time has been passed, the task denoted by *tm_task* is awakened (which will in turn call the procedure associated with the expired timer).

The fourth change is a restriction on the procedures that may be called when a timer expires. Nearly the same restrictions as given for tasks applies to procedures called by timers: the called procedure must have a single pointer parameter and the procedure must be declared at the outermost lexical level (be a global procedure). Unlike the procedures used as tasks, the procedures called when a timer expired should return when finished.

3.3. C Language Support Packages

There are two collections of C support packages: one for input and output, and one for operators in C that are not provided in Pascal. Each collection is described below.

3.3.1. Input and Output Package

Three files, C-IO.TEXT, C-IO_ASM.TEXT and C-CVTNUMSTR.TEXT, provide routines that prepare a scrolling window on the Macintosh, perform string and numeric transput, and perform conversion between numbers and strings. Most of the operations are obvious, for example, *WriteStr(S:String)* takes the passed string and writes it into the scrolling window. However, some tricks are played to minimize the amount of storage used by the i/o routines.

Two of the tricks try to minimize the number of string copies during output. Normally, output procedures specify that their parameters are to be passed by value because string literals cannot be passed by reference (VAR). In the Lisa implementation of Pascal, the conventions for passing a string by value require the caller to place the address of the passed string onto the stack and the called procedure to copy that string into local storage. For the output routines, this causes unnecessary copying. Since the string is never modified by the output routine, a more efficient implementation would use the original copy of the string directly.

The two tricks provide two different solutions to this problem. The first solution provides parallel routines for performing output, for example *WriteStr* and *AWriteStr*. One routine accepts its parameters by value (*WriteStr*) while the other accepts the parameter by reference or address (*AWriteStr*). Because the caller generates the same code for passing a string by value or by reference (just the string's address), the implementation of the output procedure that assumes

by-reference passing will work with a call of the output procedure that assumes by-value passing. The difference in the two procedures is that the by-value procedure will perform a string copy. To prevent the string copy, only the by-reference versions of the Pascal procedures are written in Pascal. Although a Pascal interface for the by-value procedures is provided, the implementation of the by-value procedures is written as a small assembly language program which merely calls the by-reference version of the program thereby avoiding an unnecessary copy operation. This trick works as long as the by-reference procedure does not alter its string parameter.

The second solution assumes that procedures are not passed strings, but are passed pointers to strings (this technique is used extensively in the TFTP package). When using pointers, one does not have to perform any copying. Although one can generate pointers to string variables through the use of the "@" operator in Lisa Pascal, there is no easy way to pass a string literal to a procedure that requires a string pointer. To allow one to pass a string literal, I created a routine called *StrCvt* which converts a string to a pointer to that string. *StrCvt* has the Pascal specification

```
FUNCTION StrCvt(S:STR255): StringPtr;
```

but the actual implementation is a small assembly language procedure which uses the fact that values of STR255 are passed by pushing an address on the stack. Therefore, the return value is the address pushed -- *StrCvt* just slides the parameter into the function return result and returns.

Storage was also wasted in the routines that convert between numbers and strings. Each time one of these procedures was called, it allocated space on the stack to hold the intermediate strings that were generated as well as space for possible input dialogs. Since the stack space was reclaimed when the procedure returned, this allocation of stack space is not, by itself, wasteful. However, each task in the network system usually performed some input or output operation, and thus each task in the system had to have substantial spare stack space to accommodate calls to the conversion routines. Since only one task was using a conversion routine at a time, this circumstance wasted a significant amount of storage. As a solution, I allocated several global variables in the conversion package to be shared by all the conversion routines. Because only one conversion routine at a time can be active (none of them block), the conversion routines and tasks can share the one allocation of the global variables and each task stack can be reduced.

The output procedures write to a scrolling window. This window is provided by the IO package, and must be initialized by calling the *PasIOInit*. It is a rather crude interface, but sufficient for getting the rest of the system working. However, since all output calls are channeled through this window (via the IO package), more sophisticated window manipulation could be used.

Input procedures obtain their values by use of a dialog. Each dialog provides a default value

and a prompt string. Although simple, the input procedures are sufficient for getting the rest of the system working. Note, however, that for the input dialogs to work, the appropriate dialog resource must be present. The easiest way to include the dialog resource is to copy the descriptions of resources DLOG 50 and DITL 5 from the resource file for TFTP (APPL-TFTPR.TEXT).

Besides the usual character and number transput operations, the IO package also provided a procedure to flash the screen: *FlashScreen.* This is a preference of mine for calling attention to the screen. I wrote this procedure before I received documentation on the *SysBeep* procedure (which I assume will flash the screen if the sound volume is set to 0). Calls to *FlashScreen* should probably be replaced with calls to *SysBeep.*

Although an attempt was made to reduce the amount of storage used by the transput and conversion procedures, there was little attempt to make these procedures fast. Therefore, they are not particularly fast. As output is written, you can see the hesitations caused by the simple algorithms. A future release of MacIP may use one of the transput packages being written by other groups, for example, by R. Martin Chavez at Harvard.

The dependencies between the three input-output packages are hierarchical when the debugging switches are set to false. However, when the debugging switches are set to true, the libraries CVTNUMSTR and IO are mutually "USE"d. The way to compile these package for debugging is to compile both with the debugging switch set to false: first CVTNUMSTR, then IO. Then the debugging switch can be set to true and the two libraries can be recompiled in either order.

3.3.2. C Operations

Pascal is missing several operators that are available in C and used in the C version of the protocol packages. To aid in the transliteration, several of the C operators were written as Pascal procedures. Further, there were several common macros in the protocol packages that needed to be simulated by procedures. Implementation of these macros as procedures were also placed in the C operations package. The files that contain these procedures are C-OPERATIONS.TEXT and C-CALL.TEXT.

A list of the C operations and the Pascal procedures that replaced them are given below. Where necessary, more explanation follows the table:

C Operation	Pascal Procedure
<<	LeftShift
>>	RightShift
&	CBitAnd (and BCBitAnd)
|	CBitOr (and BCBitOr)
~	CBitInverse

The bit operations come in two forms: C<operation> and BC<operation>. The notation <operation> was not used since the Macintosh Operating System already had procedures named *BitAnd* and *BitOr* which work on long (4-byte) integers. The C operations are different, working on 2-byte integers. The prefix C denotes a procedure that returns an integer value; the prefix BC denotes a procedure that returns a boolean value. The motivation for the "BC" procedures is that the use of bit operations frequently occurred in a boolean context, for example,

```
IF (NDEBUG & NETERR ) printf(....);
```

To avoid writing this expression as

```
IF CBitAnd(NDEBUG,NETERR) <> 0 THEN printf(...);
```

I created the parallel set of bit operations which implicitly performs the check against 0: a nonzero result causes True to be returned, a zero value causes False to be returned.

Although the approach of writing Pascal procedures to simulate C operators may not be the best for the long run, it seemed the least risk approach for getting the transliterated code to work. In a later revision of MacIP, it may be better to use different kinds of operations for the C operations. For example, many of the shifting operations are multiplications and divisions in disguise. Many of the bit operations are testing for conditions which might be done faster by using sets or boolean variables rather than logical operations on integers.[6]

Three additional operations which are not mentioned in the table are: calling procedure variables, which was implemented by the assembly language routine *Call*, pointer casting, which was done by use of the POINTER and ORD4 predefined functions, and conversions between C string formats and Pascal C string formats, which were implemented by the routines *PStr2CStr* and *CStr2PStr*. Since POINTER and ORD4 are part of Lisa Pascal, I will not describe them further. The other routines are described below.

In C, one can call any procedure variable with any number and kind of parameters. The programmer must ensure that the calling and called procedure agree on the information being passed. By contrast, Pascal requires complete information about the called procedure at compile time. Because the protocol package uses procedure variables that have different sets of parameters, one cannot write a single Pascal program or interface that can be used for all calls of C's procedure variables. My tactic was to create a single assembly language routine that would simulate any call of a C procedure variable. The convention is simple: a call of a C procedure variable is translated into a call of the special procedure *Call* with the procedure variable as the last parameter. For example, the C program might have the code

---

[6] I tried some sample programs with the Lisa Pascal compiler; sometimes the compiler produced very good code, but sometimes the code generated seemed rather long to me.

```
(*p)(a,b,c);
```

which would be translated into the Pascal code

```
CALL(a,b,c,p);
```

The implementation of *Call* is straightforward. The return address is popped from the stack and held in a temporary. The stack is then popped again into another temporary; this temporary now holds address denoted by the procedure variable. The return address is pushed back on the stack, and a jump is made to the address in the second temporary, that is, to the procedure specified by the procedure variable. Because there is no way to know the context in which a procedure variable was loaded, this technique works only for global procedures.

The main problem with this strategy is that there is no single Pascal specification for *Call* that simultaneously will match all calls of procedure variables. To solve this, I made as many Pascal interfaces as I needed (usually, one per "type" of procedure variable) and had all of the interfaces bound to the same assembly language routine.

The conversion between C strings and Pascal strings is needed for the TFTP protocol. The TFTP protocol specifies strings which must appear in its packets, and unlike other protocols which provide a count and the data, the strings are implemented using C conventions: a sequence of nonzero bytes terminated by a 0 byte. Although the TFTP program alone had the specific need to translate between the two forms, I made these procedure generally available in the operations package.

Two operations which are technically part of Unix and not C, *calloc* and *cfree*, are used by PCIP and MacIP. These were implemented as calls to the memory manager of the Macintosh.

Two macros from the C programs that were converted to procedures are *Max* and *Min*. There functions are self explanatory.

Because the 8086 in the IBM PC uses byte-swapped integers, integer values had to have their bytes reversed into a conventional form when used as values in packet headers. The 68000 does not byte-swap integers, so the representation of numbers in the 68000 can be transferred directly to a packet when needed. The original protocol package contain the procedures *bswap* and *wswap* to perform byte swapping when necessary. Although I tried to comment out all calls of these functions, the function definitions were left in the operations package. They are implemented with the use of a variant record to swap pieces of the representation.[7]

---

[7] There may be a bug in the wswap routine. I did not have an 8086 manual, so I do not know at what level the byte reversal should occur when byte swapping a long integer — word at a time or reverse all of the bytes. I merely reversed the words but not the bytes within the words.

3.4. Network Packages

Six protocol packages have been implemented: one by Apple, one from scratch and the rest transliterated from PCIP. These packages is described in the next six sections.

3.4.1. Applebus

The Applebus interface is provided by the files which make up the Macintosh Protocol Package. The code that manipulates the network is a desk accessory called *.MPP* that is distributed by Apple. Apple also provides a Pascal interface that is used by MacIP. This interface is found in the files AB-ABPasIntf.TEXT and AB-ABPasCalls.Obj.[8]

Five routines from the package provided by Apple are used in MacIP: *DDPRead*, *DDPWrite*, *DDPOpenSocket*, *DDPCloseSocket* and *GetNodeNumber*. The first two procedures read and write a DDP packet on the network. The operation can be performed asynchronously. The two socket procedures open and close a local DDP socket respectively. The GetNodeNumber function returns the Macintoshes node number.

The details of these routines can be found in the MPP supplement to *Inside Macintosh* [2]. The important features of this package are:

- When an asynchronous call is made to either *DDPRead* or *DDPWrite*, a flag in a passed parameter block will be set to 1; when the operation is completed, the flag will be set to some status value and an attempt will be made to post a network event onto the system's event queue.
- Each call of *DDPRead* will wait for a packet addressed to a particular socket and using a particular (DDP level) protocol.
- More than one read request may be outstanding at a time.

These three features will control the design of the DDP package discussed in Section 3.4.2.

I added one function missing to the Pascal interface provided by Apple: *GetNetNumber*. This function returns the integer value of the network that the Macintosh is connected to. To minimize the number of interface files, I included the Pascal specification of *GetNetNumber* in the file AB-ABPasIntf.TEXT. The implementation of the function is an assembly language routine found in AB-GetNet.TEXT.

3.4.2. DDP Package

The DDP package for MacIP is found in NET-DDP_LIB.TEXT. The structure of the DDP package was inspired by the Ethernet package in PCIP. It contains the code and data structures for sending and receiving DDP packets for use by higher level protocols, and for performing the

---

[8] No source is available for AB-ABPasCalls.

address resolution protocol. It does not do any direct manipulation of the Applebus hardware; such manipulations are performed by the *.MPP* desk accessory.

Two procedures form the basis of the package: *DDPSend* and *DDPDemux*. *DDPSend* is the "down call" side of a protocol. When a higher protocol wishes to use DDP to send a packet, *DDPSend* is called. *DDPDemux* is the "up call" side of a protocol. When a packet is received for DDP, the type field of the packet will direct *DDPDemux* to call the appropriate receiver for the higher level protocol.

DDPDemux is currently setup to receive packets intended for the following higher level protocols: DoD Internet Protocol, Address Resolution Protocol, Name Binding Protocol, Routing Table Maintenance Protocol, Applebus Transaction Protocol, Data Stream Protocol and Device Control Protocol. Of these protocols, only the DoD Internet and Address Resolution protocols have any higher level handlers. Therefore, only those two protocols are "enabled".

Within the DDP package, one must "enable" a protocol in order to receive it. To enable a DDP-level protocol, MacIP sets aside a pair of global variables which point at the current input buffer for packets of that protocol and at the DDPRead parameter block for that protocol, and then makes a call of *DDPRead* to (asynchronously) get a packet of that protocol type. Because *DDPRead* will accept packets only of a specified protocol, only *IP* and *ARP* packets should be processed by *DDPDemux*. However, if any of the other protocols are received, *DDPDemux* records the fact and then discards the packet. After enabling these other protocols, and by including a call to the appropriate procedure, any of these other protocols could be added to the current system.

At the moment, the DDP package contains exactly two sets of variables, for for enabling IP and one for enabling ARP. Two more flexible could be substituted. First, DDP could use an array of enabled connections, thereby allowing the "registration" mechanism discussed earlier. A second approach is to substitute a custom DDPRead handler for the one provided by the MPP. The substitute handler could perform demultiplexing for a number of protocols. The problem with the former is that too much overhead may be involved while looping through an array every time DDPReadDone checks for a packet. The problem with the second suggestion is that it is not well thought out. One must still make a separate DDPRead call for each protocol type. Although I think the second approach has promise, I have not given it enough consideration to suggest that the second approach replace the implemented code.

Because *DDPDemux* performs the upcalls of other protocols, the stack for the task running *DDPDemux* must be large enough to upcall the highest level protocol for which a packet might be received. The size of the stack for *DDPDemux* can be controlled by setting the global

variable *DDPStack* before initializing the network.

The bulk of the rest of the DDP package is concerned with the implementation of the address resolution protocol. (There is a good argument to be made that the ARP implementation should be in a different package.) The algorithm for the address resolution protocol that is implemented is used by many machines that provide IP on Ethernet for determining which Ethernet address should be associated with a particular IP address. I made two changes to the protocol to implement it on the Macintoshes, and left out one piece of information. First, I created a new hardware type to be used in the ARP protocol -- namely the hardware type for Applebus (I just selected the next available number, 3). Second, I had to create a new protocol type number for DDP. I would have preferred to use the published type value, 2054, but that value is too large to fit into a DDP type field. So I arbitrarily chose 23. At some point, some values for these fields will have to be standardized, and when that happens, Jon Postel at ISIF should be informed so he can revise the Internet protocol documentation [16]. Similarly, someone at Apple will have to register the protocol numbers that are introduced by the use of IP and address resolution on Applebus. (Again, I needed a new number for IP, so I arbitrarily chose 22, which seemed far enough away from the numbers given in the documentation for Version 1.0 of the MPP.)

A future release of MacIP may use NBP to translate names directly into Applebus addresses instead of using a name server to translate a name into an IP address and then ARP to translate an IP address into an Applebus address. Another possibility is that name resolution will occur at more than one level: first try NBP, and if that fails, then try ARP.

There are four other procedures which are exported from the DDP package: *DDPInit*, *DDPGotPacket*, *DDPReadDone* and *DDPDone*. Each of these is described below.

The DDPInit procedure performs initialization of the Applebus entry of the network table used by the network package. *DDPInit* contains a break in the modularity of the design (the break is present in both PCIP and MacIP) which causes an unusual interaction between modules. In PCIP, the interaction is merely confusing; in this system the break can cause the system to crash unless careful initialization is performed. The exact fault is discussed below.

The entire upcall architecture of the network system is based on the idea of "registration". A higher level protocol registers with a lower level protocol by providing the lower level protocol with an identification of the higher level protocol and a procedure to call when a packet for that protocol is received. This is easy to observe in the IP package: *in_open* causes a table entry to be made which associates a protocol number with a procedure. When a packet has that protocol number as its type, the corresponding procedure is called by *indemux*. This design is violated at the hardware level, that is, at the DDP level. The higher level protocols that are called by DDP,

which include IP and ARP, are wired into the packet reception function of the DDP package (*DDPDemux*). The IP protocol never "registers" with DDP. This means that once the DDP package is initialized, it could receive a packet tagged for IP and attempt to upcall IP. But IP may have never been initialized; the results could be (and were) disastrous. Therefore, IP must be initialized before DDP is initialized (or at least before DDP receives a packet for IP). By a similar reasoning, but going in the down-call direction, DDP must be initialized before IP. The way this mutual initialization situation is solved by PCIP relies on the fact that C initialization is performed statically, therefore all global variables are loaded with initial values before any procedure starts running. This situation does not occur in Pascal. The MacIP solves this problem by carefully including the initialization of the internet protocol while performing the initialization of the DDP protocol. Unlike the original network system, the user need not (and in fact should not) call *in_init* for initialization of the IP layer.

The DDPGotPacket procedure is a bridge between the "interrupt handler" function that the main program provides (see Section 4.2) and the actual process of reading a packet from the Applebus. After verifying that a packet has actually been received, *DDPGotPacket* places the received packet into the queue for further processing, reenables the hardware to read another packet of the received prototype and wakes the receiving task for incoming packets, i.e., *DDPDemux*. Note that *DDPGotPacket* must verify that a packet has arrived. The "interrupt handler" will call *DDPGotPacket* whenever a "Network Event" is returned by the *GetNextEvent* function. However, this event only means that an asynchronous call to *DDPRead* or *DDPWrite* has completed, therefore the event could signal a "write" completion and not a "read" completion. Hence, *DDPGotPacket* verifies a packet has really been received.

The discussion of *DDPRead* noted that a packet reception would cause the MPP to attempt to post a network event. However, the attempt could fail. The "interrupt handler" cannot depend on receiving a network event before calling *DDPGotPacket*. Although the MPP may fail to post a network event, the MPP is guaranteed to set the flag *abResult* in the parameter block for each read to an appropriate value. Therefore the value of *abResult* in all outstanding read-packet requests can be tested to see if any packet has arrived. As explained in Section 4.2, such a test is performed by the function *DDPReadDone* every time a null event is returned by *GetNextEvent*.

The *DDPDone* procedure is used to leave the Applebus network in the state the program found it. Therefore the DDPDone procedure closes the socket that was opened by the DDPInit procedure. Failure to close this socket would cause the program to abort next time it starts, since the attempt to open the socket would fail ("socket already open" error).

3.4.3. Network Package

PCIP was designed to work with many networks at once, not a single network. Therefore it included a package for organizing all of the networks for use by the IP protocol package. This package is called the network package, and its source code is largely transliterated from the C implementation. The source is contained in the file NET-NET_LIB.TEXT.

Most of the changes to the network package are simplifications. For example, the number of buffers is now a compile time constant. So too are the maximum packet size, maximum packet header size, the number of name servers, the number of networks and the number of time servers. The records describing each of the servers and the packets buffers for the packet records are all allocated as global variables and not from the heap. This was done to minimize possible heap fragmentation and to minimize the overhead imposed by the heap. However, the manipulations of buffers and packets is still done by pointers -- the addresses are generated by the Pascal "@" operator.

The initialization of individual networks has also been simplified. Because individual networks are initialized through the use of a procedure variable that has been provided to *NetInit* (via the *n_init* field of the network record), the initialization routine for each network must have the same Pascal interface, namely

```
PROCEDURE Init(Ref_Net);
```

The parameter points at the entry in the table of networks for that network. Each network is responsible for filling in the appropriate values in that table.

One of the entries in the network table, *n_stats*, is a procedure variable that denotes a procedure that can provide statistics about that network's usage. In MacIp, each procedure that a network provides for printing statistics must be parameterless. Another entry in the table, *n_close*, is a procedure variable that denotes a procedure to be used to close a network. It too must be parameterless.

3.4.4. IP Package

The Internet Protocol package can be found in the file NET-IP_LIB.TEXT. It is almost entirely transliterated C code. There are only three substantial differences: the use of procedures to manipulate bit fields in the packet header; the use of procedures to replace some macros; and the ubiquitous restrictions on procedure variables used for upcalls.

The IP package was among the first pieces of C code I transliterated. At that time, I did not believe that the Pascal compiler would generate the appropriate code for manipulating bit fields that crossed byte boundaries. Some later experiments seem to show that the Pascal compiler would produce appropriate code, but by that time I had already introduced a set of procedures

that are used to read and set bit fields in the IP header. In a later revision of the Macintosh network system, some bit fields in the record should be redeclared and the compiler should generate the necessary masking operations.

The macros were easily changed to procedures and functions, though I do not know the performance penalty that the program pays for the extra procedure calls. This penalty could be substantial, since the macros only performed field selection and were usually nested. The originally nested macros calls could be evaluated almost completely at compile time whereas MacIP will perform the extra procedure calls.

The original internet package allowed arbitrary upcall procedures. Because of Pascal's restrictions on procedures, I changed the use of the procedure variable so that only global procedures meeting the following specification are allowed to be used as upcalled routines:

```
Procedure IPUpCall(P:Packet;DataLength:Integer;Host:in_name);
```

Two minor changes from PCIP to MacIP are a recoding of the CkSum routine and a simplification of the SameNet routine.

Cksum used to be an assembly language routine because of the need to byte-swap integers before adding them. The 68000 does not byte swap its representation of integers, so an ordinary Pascal routine will work as well. Further, this checksum routine should be moved out of the IP package, since it is not unique to IP.

The SameNet procedure used to contain special code for the MIT local network. This special code was removed.

Although the IP package was tested, these tests were not exhaustive. Several common circumstances were not tested, so a few caveats about the working of the package are listed below.

The ICMP and GGP parts of the IP package were transliterated into Pascal and were changed only to be compatible with the previously mentioned alterations. No other changes were performed. Since the Macintoshes were never connected with a real IP gateway, these parts of the transliterated code were never really tested. As a maintenance procedure, I would also suggest the separation of the ICMP and GGP parts of the IP package into separate packages.

The testing of the basic IP package ignored sending packets to itself (either by the same IP address or sending a packet to itself from one network to another when acting as a gateway). Further, there are probably "symmetric" bugs in the implementation. These bugs result from an

(as yet, undiscovered) incorrect assumption that is built in both the sending and receiving implementations. For example, some IP header offset may be incorrect. Because both parts of the MacIP package make the same mistake, the package appears to work. Once MacIP is connected to another IP implementation, it may start to fail.

3.4.5. UDP Package

The UDP package can be found in the file NET-UDP_LIB.TEXT. It is nearly all transliterated C code. The package actually contains four packages: an implementation of UDP, an implementation of a time-service built on top of UDP (as per RFC868), an implementation of a name-service built on top of UDP (as per IEN116) and an implementation of a logging-service, which I suppose is a local MIT protocol [3] since I do not see a reference to such a protocol in the NIC documents [15, 16].

Because the code for the UDP package is so large, it is separated into the following segments:

InitSeg	The only procedure from the UDP package that is placed into the InitSeg segment is UDPInit. This is the same segment that all of the network initialization procedures belong to (see Section 4.1.2).
UDPNameS	This segment contains the routines for handling the name-service protocol.
UDPLog	This segment contains the routines for handling the logging-service protocol.
UDPTime	This segment contains the routines for handling the time-service protocol.

All other routines are in the main segment.

The basic UDP package has been tested and seems to work. Like the IP package, the Pascal version of UDP package constrains the procedures that can be used in upcalls. The specification of a procedure called by the UDP demultiplexer is

```
procedure UDPUpcall(p:Packet; DataLength: Integer; Host: in_name;
                    DataPtr: PTR);
```

The first three parameters are obvious. The fourth parameter is a special pointer-sized piece of information that is associated with a UDP connection when opened (see the specification for *udp_open*). The value passed to *udp_open* is passed back in the upcall in the fourth parameter. As an editorial comment, I suggest that a better approach would not include the special parameter to *udp_open* but instead have the upcall include the UDP connection on which the packet was received (rather than the unspecified pointer value). Any special values passed to *udp_open* are already stored in the record describing the corresponding UDP connection.

The time-service part of the package has been tested and seems to work, with two caveats. First, the Macintosh uses a different interpretation for a time of "0" (Jan. 1, 1904, Local Time, by the Macintosh vs Jan. 1, 1900, GMT, specified by the protocol). There is a constant in the

UDP library to compensate for the difference, but this constant is set for Eastern Daylight Time. Therefore it is incorrect outside of the Eastern United States during the summer and incorrect inside of the Eastern United States during the winter. Further, my calculation of the constant could be wrong. Since the protocol was tested against itself, the same compensating constants were used, thus an erroneous constant would not be detected. Finally, as mentioned in Section 2.2.2, this constant is actually calculated because I did not know how to create a long integer constant declaration in Pascal. The second caveat concerns the reuse of packet and connection storage. PCIP is intended to be run only once, and for some undocumented reason, the time server does not recover the space it allocates for its connection. (The code is present but commented out.) Since the PCIP program will end as soon as the time is retrieved, the unrecovered storage will not adversely affect program performance. However, the Macintosh program is intended to run "forever", and therefore cannot afford to waste storage. I admit that I have yet to see the system run out of storage in the TIME program, but the possibility exists if the timer routines are used as part of another package that runs for a substantial period of time.

The name-service part of the package has been tested and seems not to work. I have not spent any time debugging the name-service routines. They do not look complicated and I believe that they could be made to work in a very short time. However, like the time-service part of the UDP package, the name-service parts contain code to recover used storage, but for some reason, that code has been commented out. Therefore, frequent reuse of the name-service may cause the heap to run out of storage.

The logging-service parts of the package have not been tested. The server parts of the package have been compiled, but not the all of the user parts. I noticed that most of the logging user code that was included throughout the package was commented out. I did not reinstate the user code. However, I warn future administrators: this feature is a deactivated trap-door function. Should those comments be removed, it would be a simple technical matter to have the logger record selected pieces of traffic sent or received by the rest of the UDP package -- wonderful for debugging, not so wonderful for confidential (and unencrypted) mail.

3.4.6. TFTP Package

The TFTP protocols are implement by the routines in the file NET-TFTP_LIB.TEXT. One can divide each routine in the TFTP package into two sections: the network manipulation piece and the file manipulation piece. Most of the network parts of these routines were transliterated from the C version. The file system parts of these routines were written from scratch. Therefore, the file handling pieces of code contain the major departures from the C version: the use of different file system calls, the knowledge that Macintosh files have three parts and the elimination of character transformations when moving data between a packet and a file.

For each of the three parts of a file, three different sets of file system calls are used to get the information. Usually, each operation is surrounded by a CASE statement which controls where file data should be found.

The most significant difference between the C version and the Pascal version for handling the network operations comes from the extension of the TFTP protocol to allow for the three parts of a Macintosh file. These changes are manifested in:

Request Packets — The routines which send request and receive request packets must either use the extended TFTP protocol (see Section 2.1.1) or, when a user process, perform the necessary file name transformations. (When acting as a server, the program never performs file name transformations. Either the extended protocol is used or the program will transfer the data fork of the file.)

User-Process Interface — The procedure responsible for acting as the TFTP user, *tftpuse*, includes an extra parameter for describing which part of the file should be transferred. Depending on the transfer mode selected, this parameter will be transformed into a file part in the request packet or will cause remote file name transformations.

Because Macintosh files use all 8 bits in a character to represent a datum, no translations of the bytes are done during transfer. Therefore the code in PCIP for processing the end of lines and end of files was removed.

A minor difference in the code which causes a significant difference in performance is the changing of time-out constants. Using the original C values (even scaled to make them agree with the faster Macintosh clock) almost always causes a time-out. The TFTP package can become very confused when too many time-outs happen at the wrong time. Because the package leaves its UDP connection open and starts receiving multiple replies (or acknowledgements), it starts rejecting legitimate replies as being messages for an "old" connection. The problem is that the server cannot process the packet, spin the disk to verify the necessary file information and send an outgoing packet within 1 second. Therefore I arbitrarily increased the timeout period to a very large value (18 seconds). The system times out infrequently, but when it does, there is a noticeable pause until the handshaking required by TFTP resumes.

Because the TFTP routines are quite large, I divided the package into several segments:

TFTPServ — The TFTPServ segment contains the routines for the server process for TFTP.

TFTPUser — The TFTPUser segment contains the routines for the user process for TFTP.

<Blank> — The main segment contains the routines shared by both the user and the server.

Although the TFTP Macintosh program can act as a server or a user process, it cannot act as

both at the same time. Therefore, the division of the user code and the server code into two segments permits only the needed code to be loaded instead of the entire TFTP package.

Although the initial version seems to work, there are several changes that should be implemented. These are outlined below.

Because data forks are opened with the *FSOpen* call instead of *PBOpen*, if the file was already opened with write access, the *FSOpen* call will fail. I do not know what will happen if the file to be read cannot be opened for writing at all. Therefore data forks should be opened with *PBOpen* instead of *FSOpen*.

Zero-length files present another problem. In principle, an empty file should not be transferred and an attempt to do so should be flagged as an error. In PCIP, the TFTP package made this check and printed the error message. Although this approach is incorrect, it is unchanged in MacIP. The incorrect error message is generated when a data fork or resource fork is empty without the file being empty, since a Macintosh-mode file transfer (which is three TFTP file transfers) may actually attempt to send or receive a zero length fork.

An observable bug in the current TFTP package is the placement of newly transferred file. New files appear at folder location 0,0. Both the Finder and a simple copy-file program I wrote seem to pick an empty location in which to place a new icon. The TFTP program somehow creates a file with the icon always at location 0,0. By printing out the icon's location after every file system call, it seems that the 0,0 location is chosen during file creation. I speculate that some pointer is changing a some file system variable in low memory that specifies where to place the next icon. However, I was unable to find such a bug.

A more serious bug can be observed under the following conditions: Start a server, start a user, send a file (with only a resource fork or only a data fork) three times. In the middle of the last file part (data or resource fork) of the third transfer, the user Macintosh will freeze. Through some experimentation, I noticed that this situation occurs every 6th TFTP-level file reception by the TFTP server. I also noticed that around the 12th file reception, the TFTP server starts printing error messages about its inability to allocate more connection space or packet buffers. Finally, I noticed that this problem happens when using the new (Release 2.0) of the MPP. My speculations are that a place exists in the TFTP server that attempts to allocate storage but does not check to see if a null pointer was returned. The pointer is then used in an explicit pointer calculation, yielding some low memory address. This address in is then used, in some way, to send a packet to the user. The "garbage" packet contains some sort of DDP packet that the MPP processes internally. Because the packet contains worthless information, the procedure that processes the packet goes crazy, and one sees the results: first the mouse goes dead, then the

blinking cursor freezes, then the interrupt button goes dead. One must push the reboot (reset) button several times before the Macintosh will reboot. I admit that I am accusing the MPP without any hard evidence. One could as easily blame that faulty packet as causing disruption in TFTP (for example, maybe a bad packet length is causing TFTP to overwrite some of its pointers.) One way to check this speculation is to hook up a Macintosh as a spy and see what packets are send out by the server when this situation arises.

A problem that is not yet a bug, but which may become a bug, is that TFTP uses only the default volume, that is volume number 0 is always passed to file system calls. However, the file name given in the request is passed directly to the open-file call; to the extent that the open call will handle other volumes, so will TFTP. Otherwise, the current system cannot handle extra disk drives (unless that external disk drive appears to be the default volume).

Another potential problem results from the decision to truncate an existing file when using a name for a file that already exists. One could argue that the file should be deleted, since the truncation only affects the fork being transferred; the other fork is left unmolested. I believe this problem is really one that results from the simplicity of the TFTP protocol -- it just was not designed for fine control of file transfer.

3.4.7. Customization Package

The customization package can be found in NET-CUST_LIB.TEXT. This package provides the declarations for the customization record and a procedure for getting the customization information.

For now, the file name "Customization Values" is built into the package. Maybe a later version will allow the user to use different customizations in different circumstances. Another improvement would be to have the application *CUSTOMIZE* be automatically launched when the customization file is opened.

3.4.8. TCP Package

The package for the TCP library can be found in NET-TCP_LIB.TEXT. This contains a partially transliterated version of the C code; the transliteration is not complete and this package does not compile. Even if the transliteration were complete, parts of the code would still have to be rewritten so that 1) the customization information was gotten from the new customization package, 2) a uniform upcalling was provided and 3) appropriate timer values were included. I include this file in the distribution only for those who feel that they want to finish this piece; it is useless to everyone else.

4. Overall Design Changes to the PCIP Implementation

The previous sections discussed particular pieces of the network implementation in detail. There are some changes in the overall design of MacIP from PCIP. Some of the minor changes are presented followed by a discussion of a rather major design departure.

4.1. Minor Changes -- Initialization, Segmentation

4.1.1. Initialization

Initialization of the network system is different in MacIP than PCIP. First, the initialization procedures have been changed so that some of the functions performed by the initialization routines have been moved to other sections of the Macintosh code. For example, the booting procedure for the Macintosh initializes the network drivers, and the customization package reads in preliminary values for IP addresses. Thus in the old package, the initialization sequence for a network program was:

```
Initl();
Netinit(<main stack value>);
in_init();
IcmpInit();
UdpInit();
LogInit();
```

In the Macintosh version, the following code is a typical initialization sequence:

```
DDPStack := < DDP Stack value >;
NetInit(<main stack value>);
IcmpInit;
GgpInit;
UdpInit;
```

There are three obvious changes: the assignment to *DDPStack*, the omission of the call to *in_init* and the omission of the call to *LogInit*.

As mentioned in Section 3.4.2, the initialization for the internet protocol package is performed as a side effect of calling *NetInit*. Because the programmer may need to control the size of the stack used by the DDP demultiplexer (as explained in Section 3.4.2) the programmer must assign some value to the variable *DDPStack*. Finally, because the logging package is not available, there is no need to initialize it.

Another departure in MacIP from PCIP concerns the location of initialization. The main loop of the a network program using MacIP assumes that the tasking and timer packages have been started. As these packages are initialized as a side effect of initializing the network, the network must be initialized before the main loop of the program is started. This is like initialization for the Macintosh managers. In PCIP, however, the network initialization does not take place until the program is ready to use the network.

4.1.2. Segmentation

The system as a whole is divided into two segments: a segment to hold initialization procedures, and the main segment to hold procedures that will used throughout a program. Some packages (UDP and TFTP in particular) further divide their routines into more specialized segments.

The initialization segment, called *InitSeg*, contains initialization procedures that are executed before the main event loop, such as those listed in the section above. Once the initialization has occurred, the *InitSeg* can be unloaded, freeing its space for other uses.

4.2. Major Changes -- Interrupt Structure

Although the Macintosh does support the addition of interrupt handlers, the software architecture is designed to be used with synchronous events. Rather than fight the basic software design, I tried to fit the application into the basic architecture.[9] The main event loop is viewed as the interrupt handler, where each event is an interrupt. User programs are written as separate tasks. When a command is issued, say by a user selecting a command from a menu, the appropriate task for that command is continued (or started, as the case may be). Each event (viewed as an interrupt) is handled as appropriate. For example, packet receptions are reported as network events, and so cause the task responsible for packet receptions to be continued (which is the network demultiplexing process for the Applebus network, *DDPDemux*). When no event is available to be processed (as documented by a null event being returned by the call to *GetNextEvent*) the timer is checked (and processed as necessary) and the task scheduler is called to permit other tasks to run. Because of the unusual circumstance which allows a read request to be satisfied without an event being posted (see Section 3.4.2), the test of a missed network event is included when null events are processed.[10]

Because every application will have to contain similar code for these special events, the code that the programmer should include in the event loop for every application is shown below:

---

[9] But not entirely: I believe that the inclusion of a tasking package is a good design.

[10] There is a possible race condition here. Suppose that no events are present and a null event is generated. Immediately after the call to GetNextEvent (which just generated the null event), a packet is received and a network event is placed in the event queue. The packet reception also sets a global variable to indicate that a packet has been received. Now the rest of the event loop is executed, which causes the case arm dealing with null events to be run. The null-event arm checks the global variables for packet reception (via DDPReadDone) and finds a packet has already been received. But since a null event has been generated, the system assumes that the event was lost, and so calls DDPGotPacket. When the posted network event is retrieved during a later cycle through the event loop, a spurious call to DDPGotPacket will be made. This race condition is another reason why DDPGotPacket verifies that a packet is available for reading when it first starts executing.

```
NetworkEvt: { We got a packet ! }
   BEGIN
       DDPGotPacket;
   END;

nullEvent: { Nothing pressing, so check on the timers }
   BEGIN
       IF TimerEnabled THEN IF TickCount > AlarmValue THEN BEGIN
         TimerEnabled := FALSE;
         tm_signal(TIMERSIGNAL);
         END;

       IF DDPReadDone THEN BEGIN
         { Reenable the read with the same packet buffer }
         DDPGotPacket;
         END;

       tk_yield; { See if anyone else wants to execute }

   END; { end of null event }
```

Along with the use of the event loop as an interrupt handler, user applications must be written as tasks which are started and continued as necessary. A simple example is provided from the time user program. Within the event loop, a menu event for the "Get Time" command causes the following kind of code to be executed:[11]

```
IF NOT UserStarted THEN BEGIN
    Send_Task := tk_fork(Main_Task,@MySend,3*1024,'MyTSend',NIL);
    { Remember that forking a task merely sets it up to run; forking
      does not actually execute any part of the task }

    { UserStarted flag says whether the task has ever been started }
    UserStarted := True;

    { User_Running flag says whether the task in processing a command }
    User_Running := True;
    END;

IF Send_Task <> NIL THEN
    IF User_Running THEN WriteLnStr('Already executing a command.')
                    ELSE tk_wake(Send_Task)
ELSE WriteLnStr('Sending task died unexpectedly');
```

In this example, the actual application task is called *MySend*, and has a structure similar to the procedure below:

[11] The actual code contains additional statements for various kinds of debugging, which are omitted here.

```
PROCEDURE MySend(dummy:ptr);
VAR     time_answer: time_t;
        DT: DateTimeRec;
BEGIN

    WHILE TRUE DO BEGIN
        User_Running := True;
        time_answer := udptime(0,1);
        { Convert to Macintosh format }
        time_answer := time_answer - GMTOffset;
        { Convert to Time record }
        Secs2Date(time_answer,DT);
        { Print the values }
        PrintTime(DT);

        { Wait for next request }
        User_Running := False;
        tk_block;
        END;

END;
```

5. Availability

The runnable versions of the software (Macintosh programs) are available from me at the address on the front on this report. If you started work on network software using Release 1.0 of the MPP, you should know that the distributed Macintosh programs will probably interfere with your programs, since *TFTP* and *TIME* use Release 2.0 (which is incompatible with 1.0). Source files are available as Workshop (Release 3.0) minidisks and as a Unix tar tape. Hard copies of the sources are also available. The sources are intended to work with the Release 3.0 of the Lisa Workshop with the Macintosh supplement. I spent about a week trying to get the system to work under Sumac without luck. If someone else succeeds, I would like to know.

Because this software is partly provided by Apple, partly by MIT and partly by me, different notices appear in each file identifying copy rights. Material that has an MIT or my copyright notice is available from me at the address given in the notice. I will distribute material that has an Apple copyright notice to consortium schools (since they got it anyway). If Apple lets me know that I can freely distributed their files, I will do so. Until then, people and organizations not associated with the university consortium have to get the files with Apple's copyright from Apple Computer. Sorry. In any case, be aware that you are receiving a piece of software that is not bug free, still under development and subject to change at the author's whim.

For the record, the notices in the files are a subset of the following notices.

MIT's copyright notice is as follows:

My copyright notice is as follows:

Mark Sherman
Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755

Mark.Sherman@CMU-CS-A.ARPA
mss@Dartmouth.CSNet
...decvax!dartvax!mss

Apple's copyright is as follows:

I. Distribution Materials

The current release of MacIP contains Lisa Workshop sources for the packages and programs, and compiled versions of the Macintosh programs. The sources assume that you are using Version 3.0 of the Lisa Workshop (with the appropriate supplement disks for Macintosh development). The network programs assume that they are using Release 2.0 of the MPP (Macintosh Protocol Package).

A copy of the distribution materials contains the following items:

- 2 Lisa minidisks that contain the Workshop sources for MacIP.
- 2 Macintosh minidisks that contain working versions of the *TIME*, *TFTP* and *CUSTOMIZE* with reciprocal IP addresses in the customization files (The disk marked *Host 1* has IP address *$8002001B* assigned to it, while the disk marked *Host 2* has the IP address *$8002000A* assigned to it.)
- A copy of this document.
- A hard copy of the sources on the Lisa minidisks.
- If requested, a Unix tar tape containing copies of the sources on the Lisa minidisks and a copy of the sources for PCIP. (The Lisa sources were moved to Unix by an experimental version of a Lisa file transfer program. Although a quick visual inspection suggests that the files were reliably transferred, any discrepancies between Unix sources and Lisa sources are resolved in favor of the Lisa sources.)

The names of the source files that should be installed on the Lisa hard disk, that come on the Unix tar tape, and that come on the Lisa distribution disk are given below. Note that object files (Obj) are not available on the Unix tape.

Lisa File Name	Unix File Name	Distribution Disk File Name
net-q_lib.text	net/q_lib.text	net/q_lib.text
net-task_lib.text	net/task_lib.text	net/task_lib.text
net-task_asm.text	net/task_asm.text	net/task_asm.text
net-timer_lib.text	net/timer_lib.text	net/timer_lib.text
net-cust_lib.text	net/cust_lib.text	net/cust_lib.text
net-net_lib.text	net/net-lib.text	net/net_lib.text
net-ip_lib.text	net/ip_lib.text	net/ip_lib.text
net-ddp_lib.text	net/ddp_lib.text	net/ddp_lib.text
net-udp_lib.text	net/udp_lib.text	net/udp_lib.text
net-tftp_lib.text	net/tftp_lib.text	net/tftp_lib.text
net-tcp_lib.text	net/tcp_lib.text	net/tcp_lib.text
c-cvtnumstr.text	c/cvtnumstr.text	c/cvtnumstr.text
c-io.text	c/io.text	c/io.text
c-io_asm.text	c/io_asm.text	c/io_asm.text
c-operations.text	c/operations.text	c/operations.text
c-call.text	c/call.text	c/call.text
ab-abpasintf.text	ab/abpasintf.text	ab/abpasintf.text
ab-abpasintf.obj		ab/abpasintf.obj
ab-abpascalls.obj		ab/abpascalls.obj

	ab/abpascalls.text	
ab-getnet.text	ab/getnet.text	
ab-mppdefs.text	ab/mppdefs.text	ab/mppdefs.text
ab-atpdefs.text	ab/atpdefs.text	ab/atpdefs.text
appl-cust.text	appl/cust.text	appl/cust.text
appl-custr.text	appl/custr.text	appl/custr.text
appl-times.text	appl/times.text	appl/times.text
appl-timesr.text	appl/timesr.text	appl/timesr.text
appl-tftp.text	appl/tftp.text	appl/tftp.text
appl-tftpr.text	appl/tftpr.text	appl/tftpr.text
t-link.text	t/link.text	t/link.text

The files should be compiled in the following order:

c-cvtnumstr.text
c-io.text
c-operations
net-q_lib
net-task_lib
net-timer_lib
net-cust_lib
net-net_lib
net-ip_lib
net-ddp_lib
net-udp_lib
net-tftp_lib

After compiling the units above, the following files should be assembled:

c-io_asm.text
c-call.text
ab-getnet.text
net-task_asm

Any of the three application programs can be compiled by using the Workshop "P" command. After compilation, a program can be linked using the command:

```
r<t-link(appl-prog)
```

where "prog" is the program you wish to link. The resulting resource is named appl-prog.rsrc, for example, appl-tftp.rsrc. You can then copy the resource file to a Macintosh disk by using the Maccom utility.

Warning: Almost all of the files in the *ab* directory are distributed by Apple. The file ab-getnet.text is a new file that contains an assembly language procedure *GetNetNumber* which returns the network number of the Macintosh. In addition, the files *ab-abpasintf.text* and ab-abpasintf.obj have been changed to include the Pascal specification of *GetNetNumber.*

II. Registration of Parts

Like all protocol systems, this implementation uses several "magic numbers" which should be registered with appropriate authorities. A list of the numbers and the respective authorities are listed here.

II.1. Apple

Two new DDP protocol types need to be registered with Apple: the Internet Protocol and the Address Resolution Protocol.

IP Protocol number is 22.

ARP protocol number is 23.

Further, MacIp uses DDP socket 72 for receiving and transmitting its packets.

II.2. NIC/ISI

Three numbers need to be registered with the Network Information Center (via Jon Postel at ISI): a third protocol type number for the Address Resolution protocol (same number as for Apple, 23), a new hardware type within the Address Resolution protocol for identifying Applebus hardware addresses (3), and the new "file part" extension to TFTP. I do not know how the number for ARP will be handled. Unfortunately, the NIC has already assigned a number for ARP, but that number does not fit in 8 bits, hence cannot be used in the DDP type field.

References

[1] Apple Computer.
Applebus Link Access Protocol Specification Version 1.0.
Technical Report Applebus Developer's Handbook, Apple Computer, May, 1984.

[2] Apple Computer.
Inside Macintosh.
Documentation of Macintosh Internal Specifications provided mostly by Macintosh User Education.

[3] Clark, D.
Logging and Status Protocol.
Technical Report Network Implementation Note No. 31, MIT/LCS, 1981.
I have never seen this document, but copied the reference from [17].

[4] Apple Computer.
Datagram Delivery Protocol.
Technical Report Applebus Developer's Handbook, Apple Computer, May, 1984.

[5] Hinden, Robert and Alan Sheltzer.
The DARPA Internet Gateway.
Technical Report RFC 823, Bolt, Barenak and Newman, September, 1982.

[6] Plummer, D.
An Ethernet Address Resolution Protocol or Converting Network Protocol Addresses to 48-bit Ethernet Addresses for Transmission on Ethernet Hardware.
Technical Report RFC 826, MIT LCS, November, 1982.

[7] Postel, J.
Name Server.
Technical Report IEN 116, USC/Information Sciences Institute, August, 1979.

[8] Postel, J.
User Datagram Protocol.
Technical Report RFC 768, USC/Information Sciences Institute, August, 1980.

[9] Postel, J.
File Transfer Protocol.
Technical Report RFC 765, USC/Information Sciences Institute, June, 1980.

[10] Postel, J., editor.
Internet Protocol - DARPA Internet Program Protocol Specification.
Technical Report RFC 791, USC/Information Sciences Institute, September, 1981.

[11] Postel, J.
Internet Control Message Protocol - DARPA Internet Program Protocol Specification.
Technical Report RFC 792, USC/Information Sciences Institute, September, 1981.

[12] Postel, J., editor.
Transmission Control Protocol - DARPA Internet Program Protocol Specification.
Technical Report RFC 793, USC/Information Sciences Institute, September, 1981.

[13] Postel, J. and K. Harrenstien.
Time Protocol.
Technical Report RFC 868, USC/Information Sciences Institute, May, 1983.

[14] Postel, J. and J. Reynolds.
Telnet Protocol Specification.
Technical Report RFC 854, USC/Information Sciences Institute, May, 1983.

[15] Reynolds, J. and J. Postel.
Official Arpa-Internet Protocols.
Technical Report RFC 901, USC/Information Sciences Institute, June, 1984.

[16] Reynolds, J. and J. Postel.
Assigned Numbers.
Technical Report RFC 900, USC/Information Sciences Institute, June, 1984.

[17] Romkey, John L.
IBM PC Network Programmer's Manual.
Technical Report, MIT/LCS Computer Systems Research Group, 1983.

[18] Saltzer, J.
Network Programs using the DoD Internet Protocol for the IBM Personal Computer.
Technical Report, MIT/LCS, 1983.
I have never seen this document, but copied the reference from [17].

[19] Saltzer, J., et al.
PC/IP's User's Guide: Network Programs based on the DoD Internet Protocol for the IBM Personal Computer.
Technical Report, MIT/LCS, January, 1984.

[20] Sollins, K.
The TFTP Protocol (Revision 2).
Technical Report RFC 783, MIT/LCS, June, 1981.

[21] Withey, Katie.
Workshop User's Guide for the Lisa -- Release 3.0 Notes.
Manual distributed by Apple Computer.