Inter-Office Memorandum

To	Communication Protocols	Date	November 21, 1980
From	Ed Taft, David Boggs, Hal Murray	Location	Palo Alto
Subject	Ethernet Boot Protocols	Organization	Parc/CSL, OPD/SDD

XEROX

Filed on: [Maxc1]<Pup>EtherBoot.press

This memo describes the protocol by which Altos are boot-loaded over the Ethernet, the protocol used by Dolphins and Dorados for loading microcode, the protocol for discovering what boot files are available, the protocol for distributing the most recent version of a boot file to all boot servers, and the protocol for getting a boot server's statistics. It supersedes [Maxc]<Pup>AltoBoot.press which contains only the subset of this information that is available to the universities. The protocols described in this memo have been developed over several years, so don't be surprised if things appear a bit haphazard.

Because gateways are up 24 hours a day and are often located at places in the internet where many Ethernets come together, most gateways contain a boot server. However it is important to understand that boot servers and gateways are logically two very different things which are only physically co-located for convenience. There are gateways which aren't boot servers (e.g. Maxc1), and boot servers which aren't gateways (e.g. Peek).

Breath of Life

A Boot server periodically (every 5 seconds or so) sends a BreathOfLife packet on each directly-connected Ethernet. This is not a Pup: it is a raw Ethernet packet with the Ethernet destination address set to a special value. The remainder of the packet is an Alto Ethernet boot loader program.

When an Alto is booted with the BS key depressed, the boot microcode enables the Ethernet receiver to accept packets directed to host 377B and copies them into memory beginning at location 1. When a packet of type 602B is received without error, the Alto then begins executing instructions at location 3.

The current Alto Ethernet boot loader is contained in <AltoSource>EtherBoot.asm.

Mayday

An Alto which wants to be boot-loaded broadcasts a Pup of type BootFileRequest to Miscellaneous Services sockets of all hosts on the directly-connected Ethernet. The low-order 16 bits of the Pup ID is the number of the boot file desired. The Pup source port is the one to which the Alto wants the boot file sent.

Since the BootFileRequest Pup may be lost, it should be periodically retransmitted up to some maximum number of retries if no response is received at first. EtherBoot retransmits about once a

second for about 30 seconds. The reason for giving up after a while is that perhaps it is getting no answer because its Ethernet interface is broken and is polluting the net whenever it transmits.

The standard boot loader, when started at its normal address (3), reads one of the Alto keyboard words to determine the desired boot file number. All keys up (except BS) corresponds to boot file number 0. One-bits in this word are selected by depressing combinations of the following keys, listed most-significant bit first:

3 2 W Q S A 9 I X O L , "] BLANK-MIDDLE BLANK-TOP

The Dolphin microcode uses an alternate starting address (5) to get to the NetExec after the microcode has been loaded from the Ethernet. This requires that the desired boot file number (10B for the NetExec) be in AC0.

EFTP

The actual transfer of a boot file is accomplished using the Pup EFTP protocol. A boot server receiving a request for a boot file it is willing to supply simply attempts to EFTP that file to the port from which the BootFileRequest Pup arose.

Since several boot servers may respond to a single request, a server should be prepared for the EFTP transmit attempt to fail. When the Ethernet boot loader receives the first EFTPData Pup in response to its BootFileRequest, it locks on to that source and ignores Pups from everywhere else. Due to space limitations (254 words), it is unable to respond to other EFTP transmissions with EFTPAbort Pups, as specified by protocol.

There are two timeouts of interest here: the *abort timeout*, within which an ack must be received or the transfer is aborted, and the *retransmission timeout*, after which if an ack has not arrived the current data block is sent again. Ideally the retransmission timeout should be adaptive: about 2 times the average response time, exponentially aged over the last 8 samples. In any case it should be such as to retransmit a few times before the abort timeout takes. The abort timeout should be a function of the available bandwidth of the path between the sender and receiver. The table below lists recommended values.

	First Block		Subsequent Blocks	
	Abort	Retrans	Abort	Retrans
Fast net	500	100	5000	1000
Slow net	10000	2500	10000	2500

(I need to double check these. /HGM May-80)

The timeouts for a slow net are suitable down to about 2400 bits/sec. The retransmission timeouts listed are for EFTP implementations which do not use an adaptive algorithm; the initial adaptive retransmission timeout may have to be reduced from its default value (typically 1 second) for the first block on a fast net. A reasonable simplification is to assume that all nets except Ethernets are slow. Even on an unloaded 9600 bits/sec line it takes several minutes to send a full core image boot file. Boot servers should be able to boot an Alto over an Ethernet while simultaneously updating a boot server at the end of a slow phone line.

Boot File Names and Numbers

String names and 16 bit numbers are both used to refer to boot files. Servers deal mostly in boot file numbers: requests to send a boot file refer to it by number; servers compare the creation dates of files with identical names and numbers when distributing new versions. The NetExec sorts its directory by name, keeping the number, date, and server host address as auxiliary information.

Boot file numbers less than 100000B have a uniform meaning throughout the network, are updated automatically and are assigned by administrative fiat. The remaining numbers are available for local

use and do not propagate. [Maxc]<BootFiles>BootDirectory.txt is the master directory of registered boot files.

Most gateways have a boot file with the name <gatewayName>.boot, with number 100000B. This is intended for use by people developing new boot files. A test version of a boot file stored there won't propagate. It is also handy for experimenting and glitch chasing since there can't be any doubt about which boot server it came from.

Boot Directory Information

When the EtherBoot mechanism was first developed it was only expected to handle a small number of files -- DMT, Scavenger, FTP and a few others -- and key combinations were picked that were easy to remember and convenient for people with two hands and ten fingers. Even so, it was difficult to remember the keys and the number of files grew to the point where this scheme was getting out of hand, so the NetExec was developed. The NetExec was assigned one of the last convenient key combinations and it is now the standard way for humans to invoke other boot files.

The NetExec discovers what boot files are available by broadcasting a Pup of type BootDirRequest to Miscellaneous Services sockets on all hosts on the directly-connected Ethernet. Hosts that are boot servers respond with packets of type BootDirReply containing <boot file number, date, boot file name> tuples. A boot server with lots of boot files may fill several BootDirReply Pups. The NetExec builds a directory of <boot file name, boot file number, date, host> tuples from these responses.

Boot File Update

At present there are two programs which implement boot servers: Gateways and Peek. There are about 20 gateways in operation, and the number is growing. It takes a few minutes per gateway to update one boot file (most gateways are at the end of slow phone lines). Not all gateways are up all of the time. There are probably 50 Peek disks in the world, each with some subset of the boot files that existed when the disk was built. The owners of Peek disks are exhorted to rebuild their disks about once a month. The result of this anarchy is that old versions of boot files persisted in the internet for years.

A Boot file now includes the date on which it was built. Boot servers periodically exchange boot file directories which include these dates. When a new version of a boot file is stored onto any boot server, all other boot servers will soon discover this and automatically update their local copies. The protocol is similar to that used by name servers to update the network directory.

About once an hour and each time a new boot file arrives, each boot server broadcasts its boot file directory in a BootDirReply Pup to miscellaneous services sockets on all directly connected networks. When a boot server receives one of these, it compares the dates of its local boot files with the dates in the Pup. If the sender has a more recent version, then the receiver requests a copy using a BootFileRequest Pup. If the receiver has a more recent version, then it should send a BootDirReply to cause the sender to update. The same EFTP protocol that is used to boot an Alto is used to move new versions among boot servers.

If the date comparison is unguarded, a damaged file with a bogus date far in the future could propagate everywhere and it would be impossible to purge. To protect against this, a file which claims to have been created in the future should be treated as if it had a date of zero, thus making it elegible for update by anyone.

The automatic update mechanism has turned out to be quite robust. In fact, it is almost impossible to switch back to an old version of a boot file if a new one doesn't work correctly. If, for example, you use FTP to store it on a boot server, as soon as that server gets restarted it will notice that it has an old version and overwrite it with the (buggy) newer version from some nearby boot server. To avoid this problem, GateControl smashes the date to "now" when it sends a file with a name ending in .boot. There is no corresponding trick for microcode files or Pilot boot files. You must rerun the

program that creates the Ethernet format versions of these files.

Boot File Format

Except for two complications, the server doesn't understand the contents of a boot file. The first is simply the creation date which is needed by the automatic update mechanism. The second pertains to the format of the file. Boot servers must transform S0-format files into B-format files. See the BuildBoot documentation for the details. In summary, the the first few words of a boot file contain:

word 0: don't care
word 1: 0 (if this is not 0, your file will get reformatted)
word 2: don't care
word 3: high order half of the creation time
word 4: low order half of the creation time

In an Alto boot file, the first page is a disk boot loader. It is considered ok for boot servers to clobber the leader page by frabricating one containing the correct creation date. The idea is to avoid opening the thrash of opening the file in case contact is not established with the bootee. Note that this means that BootFrom.~ Mumble won't work if Mumble.boot has been processed this way.

Microcode boot files and Pilot boot files are *encapsulated* by adding a header page (which basically contains only the creation date) to the front of the file. This allows the normal boot file update mechanism to automatically propagate new versions of microcode and Othello around the internet with no additional software. The header page is discarded by the appropiate software.

There is no checksum in boot files, so if the version on a boot server gets smashed (in such a way that the date remains reasonable), the damaged version may propogate around along with the valid one.

Booting Microcode

When Dolphins and Dorados arrived, it became important to be able to get microcode as well as boot files from the network. The following very simple protocol is used to boot microcode.

A machine that needs microcode simply broadcasts a packet of type MicrocodeRequest. The high half of its pupID contains the version number of the boot protocol. Currently the only version supported is 1. This may be expanded when new machines impose other requirements. The low half of the pupID contains the microcode file number of the desired file.

For version 1 of the protocol, a server willing to supply the data simply sends a sequence of packets of type MicrocodeReply as fast as it can. The high half of its pupID contains the version number (1) and the low half of the pupID contains the packet sequence number. After all the data packets have been sent, the server sends an empty (0 data bytes) packet for an end marker. There are no acknowledgments. This protocol is used by Dolphins and Dorados.

> Currently, the version 1 servers send packets containing 3*n words of data. This constraint is imposed by the Rev L Dolphin EPROM microcode. I'd like to remove this restriction if I get a chance, so please don't take advantage of it unless you need to. The Rev L Dolphin EPROM also requires the second word of the source socket to be 4. /HGM May-80

This protocol depends upon the bootee being able to catch every packet of the sequence. This normally works ok for Dolphins and Dorados because their Ethernet interfaces were designed to catch back to back packets. It will not work on a *very* heavily loaded net since occasional packets will be lost after they encounter 16 collisions.

Booting Dolphin Microcode

The hard part of loading microcode into a Dolphin is deciding which microcode to load. The options are two dimensional: which microcode device drivers, and which emulator(s)? To avoid what might otherwise become unmanagable chaos, the emulator microcode is packaged into interesting combinations, and loading microcode is split into three phases.

When you poke the boot button, the *Boot microcode* is copied from an EProm into control store. It checks the processor a bit, and then tries to load the *Initial microcode* from the disk. If it runs into any problem, it tries to load Initial from the Ethernet. The normal trick for forcing a Dolphin to Etherboot is to cycle power off for a few seconds. This makes the disk appear not ready for a minute or so. In either case, the same code (except for a different starting address) is loaded directly into control store. Initial sets up memory, and if it was Etherbooted, it figures out which variant of the final microcode to load. Currently it does this by assuming that you want the Alto emulator, and looking at the clock speed on the display controller to decide if you are running on a SDD Dolphin, a CSL D0, or a TOR machine. The desired *emulator* microcode is then loaded into main memory. It is started using LoadRam. LoadRam has been standardized across all microcode variants to simplify this process.

EtherBoot is the module in the EProm and loads Initial from the Ethernet. When control is passed to EtherBoot, it first dallys for a second or so to let you read the MP. If the BootReason register contains anything other than a push button boot, the delay is lengthened to a minute or so to avoid flooding the Ethernet if the Ethernet board is sick. (If the Ethernet board causes an H4 parity error, for example, the fault handler will put a number into the MP, dally for a second, and then reboot the machine.) EtherBoot then locates the Ethernet board, and sets up the task asignments. The emulator task, sets a timer to 10 seconds, starts the input task (which will initialize itself and wait for a packet), and then pokes the output task. The output task broadcasts a request for microcode file number 0, and then goes to sleep. (It's collision processor is quite primitive.) When the first valid packet arrives, the input task will latch onto the the server's host number. Since memory has not yet been set up, the data words are stored directly into control store. Notice that the EProm must be careful to avoid overwriting itself in case the packet is bad. The problem is that the data must be stored somewhere before the status of the packet can be checked. In order to simplify things for EtherBoot, the boot server sends Pups that contain 3*n data words. Each time a packet arrives correctly, the input task resets the timer to 2 seconds. As the data arrives, a checksum is accumulated. When a jump block is encountered and the checksum is ok, Etherboot jumps to the designated location. If anything goes wrong, for example a CRC error, the input task puts a number in the MP and goes back to the beginning. In the background, the emulator task is decrementing the timer. If it expires, the emulator will retry everything up to 15 times.

EtherLoad is the module in Initial that loads the emulator microcode from the Ethernet. It is very similar to EtherBoot, in particular, it does not use the normal Ethernet microcode. EtherLoad's task is a bit simpler that EtherBoot's because it simply places the data into memory. When an empty (0 data bytes) Pup arrives, EtherLoad checks the checksum, and it if is ok, it jumps to LoadRam.

Booting Dorado Microcode

Needless to say, booting a Dorado is quite complicated. This description is quite brief. For more information, see Ed Taft's memo.

When a Dorado is powered up, a microcomputer in the baseboard goes into action. If all goes well, it manages to get Initial (which is stored as data in a prom in it's address space) into the Dorado. Initial then initializes the map and main memory, turns on the normal IO devices, and inspects the keyboard to determine which variant of the emulator microcode to load. Initial then uses the microcode booting protocol to load the desired microcode into main memory and then uses LoadRam to load it into control store. The emulator microcode will boot the software from the disk or Ethernet by testing the keyboard just like the Alto does.

Microcode Files

By convention, the names of microcode boot files end in .eb. In order to use the normal update mechanism to propogate new versions of microcode around the internet, microcode files are disguised as normal Alto format boot files. See the previous section on Boot File Formats for the details of the first few words. For simplicity, the server skips over the whole first page before it sends out the microcode, so there is plenty of spare room for other things. The version 1 server requires that the first word of the boot file contain a 1. This restriction is likely to get removed or updated to contain more information, for example, the type of the target machine. The format of the rest of the file depends, of course, on the target machine.

To avoid bolting arbitrary large constants into Proms, the boot server adds an offset to the microcode file number to translate it into a boot file number. Currently the offset is 3000B. The currently assigned microcode file numbers, the corresponding boot file numbers, and the file names are:

```
  0  3000  Initial.eb  (9 packets)
  1  3001  AltoLF.eb  (45 packets)
  2  3002  AltoCSL.eb
  3  3003  PilotLF.eb
  4  3004  PilotCSL.eb
  5  3005  AltoTOR.eb
  7  3007  PilotTOR.eb
100  3100  DoradoMesa.eb
101  3101  DoradoSmalltalk.eb
102  3102  DoradoLisp.eb
103  3103  DoradoCedar.eb
```

Pilot Boot Files and the Ether Germ

For bootstrapping and/or working on a disk that has not yet been formatted, it is very convient to be able to load Pilot boot files over the Ethernet. To support this, the Germ (Pilot boot loader) has an optional Ethernet driver. It uses the normal BootFileRequest/EFTP protocol. Pilot boot files are frequently larger than a full Alto memory. This is harmless since they won't run on an Alto anyway. By convention, the names of Pilot format boot files end in .pb and the Germ is called Pilot.eg.

MesaNetExec

The MesaNetExec is very similar to the BCPL NetExec. In addition, it contains logic to load Pilot boot files. If the desired file name ends in .pb, the MesaNetExec doesn't actually load it, but instead it uses the normal BootFileRequest/EFTP protocol to load the Germ and the appropriate microcode into main memory. Next, it moves the Germ to hyperspace and then uses the LoadRam instruction to load the control store with the new microcode. When the new microcode is started, it will start running the Germ which will finally load the specified Pilot boot file.

If your Dolphin normally runs in Alto mode, this is the way to get to Othello so that you can install new microcode on your disk. You can also use the MesaNetExec's SetVersions command to select special debugging variants of the Germ and/or microcode.

Server Statistics

Boot servers may optionally keep statistics on their activities and make them available through the net. A program requests a boot server's statistics by sending a Pup of type BootStatsRequest to the miscellaneous services socket, and the boot server responds by sending a Pup of type BootStatsReply containing the statistics. The first word of the reply is a format version number which is incremented whenever the format changes.

Pointers to other Documentation

When an Alto is hardware-booted over the Ethernet, all three of the steps (BreathOfLife, MayDay, EFTP) are executed. A software-initiated boot may be accomplished by copying the boot loader into page 0 and jumping into it, thereby starting at the "Mayday" stage with the boot file number and host as optional arguments. Further information may be found in the "EtherBoot" package documentation on [Maxc]<AltoDocs>EtherBoot.tty.

The standard Alto Ethernet boot loader can load only B-format boot files. A boot server must transform S0-format files into B-format files (by rearranging pages) before sending them. Further information may be found in the "BuildBoot" subsystem documentation on [Maxc]<AltoDocs>BuildBoot.tty.

[Ivy]<DoradoDocs>DoradoBooting.press contains detailed information about booting Dorados, including descriptions of when and how they get microcode and/or boot files from boot servers.

Details

A Breath of life packet is a raw *(non-Pup)* Ethernet packet:

Destination:	377B
Type:	602B
Contents:	A boot loader program.

The starting address of the boot loader is the the third word of the packet (first content word) which will be address 3 in Alto memory. The total packet length must not exceed 256 words.

Boot servers listen on the Miscellaneous Services socket (4) and handle some or all of the Pup types listed below.

BootFileRequest

Pup Type:	244B
Pup ID:	Low 16 bits are the boot file number desired
Pup SPort:	The port to which the boot file should be EFTPed
Pup DPort:	Miscellaneous services
Pup Contents:	None

This packet is generated in several contexts: 1) by the Ether boot loader while booting an Alto, 2) by a boot server to update a local copy of a boot file, 3) by the MesaNetExec to get copies of the Germ and the appropiate microcode, and 4) by the Germ to load a Pilot boot file.

MicrocodeRequest

Pup Type:	264B
Pup ID:	High 16 bits are the protocol version number (currently 1) Low 16 bits are the microcode file number desired
Pup SPort:	The port to which the microcode file sent
Pup DPort:	Miscellaneous services
Pup Contents:	None

This packet is generated by Dolphins and Dorados that need microcode.

MicrocodeReply

Pup Type:	265B
Pup ID:	High 16 bits are the protocol version number (currently 1)

	Low 16 bits are the packet sequence number
Pup SPort:	The port from which the microcode request was sent
Pup DPort:	Miscellaneous services
Pup Contents:	data, or empty for an end marker

A cluster of these packets is generated by boot servers in response to a packet of type MicrocodeRequest.

BootDirRequest

Pup Type:	257B
Pup DPort:	Miscellaneous services
Pup Contents:	None

This packet is generated by the NetExec to discover who the boot servers are and what files they have.

BootDirReply

Pup Type:	260B
Pup ID:	if it is in reply to a BootDirRequest, the ID should match the request.
Pup Contents:	1 or more blocks of the following format: A boot file number (the number that goes in the low 16 bits of a BootFileRequest Pup), an Alto format date (2 words), a boot file name in BCPL string format.

This packet is generated 1) in response to a BootDirRequest, 2) gratuitously broadcast every hour, and 3) in response to a BootDirReply advertising an older version of a local file.

KissOfDeath

Pup Type:	247B
Pup DPort:	Miscellaneous services
Pup SPort:	The BootFileRequest Pup is sent to SPort.host on the local Ethernet. If SPort.host is zero, it is broadcast.
Pup ID:	The low 16 bits contain the boot file number to put in the BootFileRequest Pup.
Pup Contents:	None

DMT is the only program which currently responds to KissOfDeath Pups and is used now only to run tests on the Ethernet. We have a multiprocessor with about 125 6-MIP CPUs on the second floor of Parc which is idle 16 hours a day just waiting for someone to figure out how to use it.

BootStatsRequest

Pup Type:	253B
Pup DPort:	Miscellaneous services
Pup Contents:	None

BootStatsReply

Pup Type:	254B
PupID:	same as the BootStatsRequest that triggered it.
Pup Contents:	A version number to identify the format of the following words (current version = 1). Followed by the number of boot files sent, followed by the number of boot directories sent, both in BCPL double precision format.

Revision History

October 17, 1976

First release

March 9, 1978

Boot directory protocol added.

December 31, 1978

Automatic update protocol added.

February 13, 1979

Boot server statistics protocol added.

May 15, 1980

Microcode booting added, file name changed from AltoBoot to EterBoot.

November 21, 1980

Dolphin EProm requires second word of source socket to be 4.