BOLT BERANEK AND NEWMAN INC

CONSULTING · DEVELOPMENT · RESEARCH

BBN REPORT NO. 3302
A.I. REPORT NO. 41

(12) B.S.

INTELLIGENT ON-LINE ASSISTANT AND TUTOR SYSTEM

Technical Progress Report No. 1
18 September 1975 to 31 March 1976

Sponsored by
Advanced Research Projects Agency
ARPA Order No. 3091

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by ONR under Contract No. N00014-76-0476.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: BBN Report No. 3302 AI-41

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): INTELLIGENT ON-LINE ASSISTANT AND TUTOR SYSTEM, Technical Progress Report No. 1, 18 September 1975 to 31 March 1976

5. TYPE OF REPORT & PERIOD COVERED: Tech. Prog. Rept. #1, 18 Sept. 75-31 March 76

6. PERFORMING ORG. REPORT NUMBER: BBN Report No. 3302

7. AUTHOR(s): Mario C. Grignetti, Alice K. Hartley, Catherine Hausmann, William L. Ash, Robert J. Bobrow, Alan Bell

8. CONTRACT OR GRANT NUMBER(s): N00014-76-C-0476, ARPA Order - 3491

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Bolt Beranek and Newman Inc. 50 Moulton St. Cambridge, MA 02138

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 6D30

11. CONTROLLING OFFICE NAME AND ADDRESS: ONR, Department of the Navy, Arlington, VA 22217

12. REPORT DATE: April 1976

13. NUMBER OF PAGES: 111 pp

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report): Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited. It may be released to the Clearinghouse, Department of Commerce, for sale to the general public.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Technical progress rpt. no. 1, 18 Sep 75 - 31 Mar 76.

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Artificial Intelligence, Computer Assisted Instruction, Natural Language Understanding, Syntactic Analysis, Semantic Interpretation, Semantic Networks, Question Answering, INTERLISP implementation, Writable Control Store, Memory mapping device

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report covers work performed from 18 September 1975 to 31 March 1976 under Contract No. N00014-76-C-0476. Our final objective is to develop a system to help computer-naive people deal with their Intelligent Terminal, an envisioned personal mini-computer of great sophistication and power. Our system will be capable of mediating "intelligently" between these users and the tools that will be available to them in the Intelligent

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified 060 100

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. cont'd.

Terminal: it will provide streamlined instruction on how to use these tools, and it will answer questions about those tools posed in a comfortable subset of English. The system will be written in INTERLISP.

This report describes in some detail our design and implementation plans for bringing up an INTERLISP system on a PDP-11 computer (a prototype Intelligent Terminal), and the work performed to date on an Intelligent On-Line Tutor and Assistant (INLAT) for new users of Hermes, an independently developed mail processing system that exemplifies the kind of tool that could be made available in future Intelligent Terminals.

INTELLIGENT ON-LINE ASSISTANT AND TUTOR SYSTEM

Technical Progess Report No. 1
18 September 1975 to 31 March 1976

ARPA Order No. 3091

Program Code No. 6D30

Name of Contractor:
Bolt Beranek and Newman Inc.

Effective Date of Contract:
18 September 1975

Contract Expiration Date:
31 December 1976

Amount of Contract:
$592,562.00

Contract No. N00014-76-C-0476

Principal Investigator:
Mario C. Grignetti
(617)491-1850 x394

Scientific Officer:
Gordon Goldstein

Short Title of Work:
INTELLIGENT ON-LINE ASSISTANT AND TUTOR (INLAT)

Sponsored by
Advanced Research Projects Agency
ARPA Order No. 3091

TABLE OF CONTENTS

SECTION 1. INTRODUCTION

We are interested in making it easy for computer-naive people to sit in front of a computer terminal (eventually an Intelligent Terminal) and learn how to use the computer tools available through it. We believe that a good way to achieve this goal is by means of a reactive learning environment, where tutorial services ranging from systematic teaching to occasional on-line help are made easily and immediately available to users.

The three fundamental aspects that such a learning environment must incorporate are:

1) well-organized tutorials, i.e. a set of lessons that systematically teach the how-to knowledge that is essential for useful operation of a computer tool;
2) the ability to "look" over the user's shoulder and be "aware" of what he is doing, so that when needed, the system can offer help that is specific to the task the user is engaged in.
3) the ability to perform these services in response to user requests expressed in natural language (English).

Since such reactive learning environments incorporate Artificial Intelligence techniques, and to the extent that they appear to have features that mimic human intelligence, we have coined for them the acronym INLAT ("Intelligent" oN-Line Assistant and Tutor).

The specific objectives of the present contract are to develop INLAT's for three of the tools that may conceivably be the mainstays of future Intelligent Terminals. These tools are Hermes (a computer mail processing system), the Rand editor (a multi-window, scope editor) and a News Service information retrieval system (such as the New York Times Information Bank or Stanford's News Service program).

INLAT's can be best developed in a programming environment that makes it easy to turn out large and sophisticated systems in relatively short times. It is very important, especially when dealing with potentially large populations of computer-naive users, to be able to drastically revise design concepts, approaches, and implementations. We believe that INTERLISP is the best available "milieu" in which this fast turn-over development effort can take place. Not only is it based on a language of enormous power and flexibility (LISP), the work horse of most AI work today; it is also a system that has "built in" many of the facilities (such as the Do What I Mean (DWIM) and the Programmer's Assistant) that are considered essential in an Intelligent Terminal. For these reasons, the crucial part of the present effort is to provide an INTERLISP environment on the machine that has been selected as a first prototype for the Intelligent Terminal, -- the DEC PDP-11.

The body of the Report begins with a description of our design goals and the work performed in bringing up an INTERLISP system on the PDP-11. In the remainder of the Report we describe the work performed to implement our INLAT. Although the INLAT is intended as an aid in the use of multiple tools, the bulk of our work in this area to date has been centered on providing a counselor and assistant for the Hermes message system being developed independently at BBN. Therefore, our description is limited to an INLAT-Hermes system.

SECTION 2

INTERLISP-11

Introduction and Purpose

The main objective of our Intelligent Terminal work is to make it easy for computer-naive people to learn how to work with a computer capable of performing a number of clerical functions. We believe that our objective can be achieved if we provide tutorial and on-line help facilities that rely heavily on natural language understanding techniques. These techniques have been developed for the most part using LISP programming environments of great sophistication and power, of which INTERLISP is perhaps the best known.

Although INTERLISP is a formidably powerful tool, and its advantages for AI work are widely recognized, its usage has been relegated to, and is commonly associated with powerful, large machines. In recent years, however, the technological advances in computer hardware have been so dramatic that today's "minicomputers" have almost equalled the power of such mainstays of LISP as the PDP-10, and are available at a fraction of the latter's cost. Therefore, one of our most important subgoals is to bring up a full INTERLISP system (as it now exists in the PDP-10 under TENEX) on a PDP-11 computer. This we call INTERLISP-11.

More specifically, what we want to provide is an INTERLISP environment with the following characteristics:

a) serves a single user,

b) is fully compatible with INTERLISP,

c) runs at half the speed of INTERLISP-10 when there is only one user (therefore it should appear much faster than INTERLISP-10 under "normal" TENEX load conditions),

d) has a larger address space (4 million words) than INTERLISP-10 (which is known to be too small), and

e) minimizes machine dependent code for easier exporting to another computer.

In the rest of this section we describe our plan for achieving these purposes. We begin by describing the hardware configuration that we selected and the reasons for choosing it. Next we describe the software work that is needed to accomplish our goals with the selected hardware. Finally, we describe the present status of the work and our prognosis of what will be done by when.

Hardware

The characteristics of the desired INTERLISP-11 environment impose three kinds of broad requirements on the hardware. These requirements comprise the ability for:

1) efficiently executing LISP compiled code and storing compactly LISP data structures (mainly lists),

2) dealing with a large virtual address space,

3) running the UNIX operating system and the Rand software developed to run under it, especially RITA and the Rand Editor.

In order to satisfy the third requirement we need a system centered on the PDP-11 computer; the other two requirements impose a particular selection within the PDP-11 family and the incorporation of extra hardware. Specifically, the hardware configuration that was specified for our purpose consists of a DEC PDP-11/40 processor with 128K words of core memory, a fast 512K words fixed head disk for swapping (RS04), and a slower 28M words disk for secondary swapping and filing (Telefile). In addition, we have an interface for network communications (IMP-11A), a Writable Control Store (WCS) for efficient and compact microcoding of compiled LISP code and for compact list structure, a specially designed Memory Mapping Device (MMD) to facilitate the implementation of the large address space, and a number of other peripheral devices. In what follows we will describe only the two non-standard components in the above configuration, the WCS and the MMD.

Writable Control Store -- Purpose and Description

The Writable Control Store (WCS) was designed at Carnegie Mellon University by Professor Samuel Fuller and

his group, and it is used extensively in their Hydra project. It provides a way of augmenting the capabilities of the standard 11/40 processor which is crucially important for our purposes. The most important advantage derived from using the WCS is that it allows us to design an instruction set for LISP-compiled code that is independent of the host machine instruction set and is therefore better suited to LISP idiosyncrasies. The resultant object programs are therefore more compact and run more efficiently than their PDP-10 counterparts. In terms of required memory size, we estimate that object programs written in the above set will be only 1/3 as large (in terms of number of bits) as equivalent PDP-10 compiled LISP. Alternately, they will require 3/4 as many 16-bit PDP-11 words as required on the PDP-10 (36-bit words). Finally, because of its relative machine independence, the new object code will permit substantially easier transfer to newer, more cost-effective hardware when it becomes available.

The 11/40 processor is implemented with 256 words (56 bits) of read-only microcode. It is designed so that the 11/40 options, such as the Extended Instruction Set and the Floating Point instructions, can be installed by simply plugging in additional microcode read-only memory modules. This plug-in extensibility makes it possible to design and build the WCS.*

---

*The 11/40 is the only PDP-11 able to accommodate the WCS

The WCS has the following characteristics:

1) it provides 1000 words of additional high speed microcode control store which can be used to implement critical parts of INTERLISP-11, including the object code execution "interpreter";
2) it is writable, facilitating both design and debugging, and allowing for the possibility of swapping microcode sets (such as one for the garbage collector);
3) it provides an extension of the underlying PDP-11 micro machine, enhancing its generality. The standard PDP-11 ROM has 56-bit words while the extended micro machine is controlled by 80-bit words.

While (1) and (2) are fairly clear, (3) requires somewhat more comment. Simply adding extra control store to the basic 11/40 micro machine would not provide a very useful general-purpose microprogrammed machine, since the 11/40 microcode is designed primarily for implementation of the PDP-11 instruction set and thus lacks generality. The WCS adds to it the following features

a) A general Mask-Shift unit that allows the extraction of any contiguous bit field from a 16-bit word, to be used as an input to the PDP-11 Arithmetic Logic Unit (ALU)
b) A 16 word stack that allows micro-subroutining capabilities, temporary storage, etc.
c) Quadruplication of the address space (1024 vs 256), aside from writability.

d) Extended microbranch control, i.e. N-way branching controlled by output of Mask-shifter.

The extra power provided by these facilities allowed us to design an instruction set (see the APPENDIX A at the end of this section) that is almost ideally suited for LISP compiled code.

Memory Mapping

The 4M words address space specified previously for INTERLISP-11 clearly exceeds the amount of main memory (core or RAM) one can reasonably expect will be available for Intelligent Terminal applications. This implies that a way must be found to map the INTERLISP-11 address space into a smaller real memory.

The existing mapping device for the PDP-11 (the so called Memory Management Option) doesn't help us very much since it is designed for the opposite application it maps a small virtual address space into a larger amount of real memory. This is good for multiuser applications since it allows several user processes with small virtual spaces (32k) to be resident in core simultaneously, but this is not our case.

We considered the software approach: a page table that lives in core, microprogramming for putting together a real

core address, and reloading of Memory Management Registers (MMR's) for accessing that real address. This approach is undesirable for several reasons: a) it is slow, b) the microcoded instructions that we'd need would usurp WCS memory space that could be used for other purposes, c) and it would impose an extra burden to our programming tasks. This last consideration is very important; in order to minimize reloading of MMR's, which is very expensive, we would have to be very clever in order to keep core "windowed in" as much as possible. Clearly, a solution that allows us to treat all core references equally is much preferred, since it simplifies the programming effort, reduces debugging, and speeds coding.

These reasons suggested a hardware/software trade-off: a simple hardware pager, that we call Memory Mapping Device (MMD), to map the 22-bit address space of INTERLISP-11 into a 19 bits (512K) real core address.

The MMD works as follows. The 22-bit virtual address space is visualized as consisting of 4K pages, each 1K words long. A 22-bit address specifies a page number (a page table location) with its high order 12 bits, and a location within the page with its low order 10 bits. The main component of the MMD is a 4K by 16 bits random access memory that acts as a page table, i.e. it maps a virtual page into a real memory page. The 16-bit word at the specified location in

the page table contains information on how to access the page. If the page is in core, its real core address, age, and status are specified as follows

AGE 5 bits - Clock reading at time page was last referenced (Modulo 32). AGE is jammed in from a special Age register every time the page is referenced. The Age register is incremented by software at presettable clock intervals.

WMOD 1 bit - Tells whether or not the page has been modified by a write access.

WPR 1 bit - Grants or refuses permission to write on the page.

RCP 9 bits - real core page number (most significant 9 bits of 19 bit Real core address)

If the page is not in core, AGE is set to a special value, and the Page Management module (see below) takes over.

The MMD hardware constructs a 19-bit address by concatenating RCP with the least significant 10 bits from the virtual address. An indication of the efficiency of performance we expect can be gleaned from the following example: To obtain CAR of a list (i.e. its first element)

without the MMD would take 23 microseconds under the most favorable circumstances. Of these, 14 microseconds are used for software mapping one data reference; the other memory references are assumed non-mapped (which implies that the stack is windowed in core). With the MMD, this time is cut to 10.8 microseconds, and special care to keep the stack windowed is not required.

Age Distribution Registers

The AGE field in each page table entry permits efficient demand paging by making it possible to find the least recently used pages. Such pages are good candidates for removal from memory when one needs to make room for new ones. Since the table is fairly large (4K words) a pure software search for the least recently used pages will be time consuming and inefficient; for this reason, we have included a set of age distribution registers. These registers provide a histogram of the page ages existing in the table. In fact, there are 2 sets of age distribution registers, one for pages that have been modified, the other for unmodified pages. That is, for each possible combination of AGE and WMOD there is an Age Distribution register that contains the number of pages whose table entries contain that combination. Thus, for example, to find the n oldest pages one finds the corresponding ages by examining at most 64 Age Distribution Registers, and then

uses these ages as search keys in scanning the map.

The complete set of specifications for the MMD can be found in Appendix B.

Software Implementation

The software implementation of INTERLISP is greatly facilitated by the fact that a large fraction of the system is written in LISP. The non-LISP parts of INTERLISP represent

1) the way INTERLISP is integrated with its operational environment,
2) the kernel, or irreducible parts that must be written in machine language and on which the rest of the system is built.

In the following we discuss these issues in detail.

Operating Environment

This package provides the facilities of a simple operating system, i.e.: the interfaces to specific hardware devices and the scheduling and performance of core allocation procedures. This portion of the system will be small in accord with our desire to minimize the machine dependence of the system.

Page management is the major component of the operating environment. The page manager allocates real core for the 4 million word LISP virtual address space using the memory mapping device for efficient demand paging. It uses 3 levels of storage: core, fixed head swapping disk, and bulk storage disk. The page manager also provides user mode access to the paging mechanism such as mapping file pages into address space, and get/set access permission for a page.

In addition, the operating environment supplies primitive file operations (read/write-character, open/close file); terminal echo, padding, and interrupts; interfaces to IMP 11A, auto dialer, real time clock; and scheduling of background processes. The operating environment also has the ability to run straight (non-LISP) PDP-11 processes such as UNIX user programs.

The precise method for running UNIX user programs has not yet been completely specified. An outline of our intended approach is the following. The MMD has been designed so that it is possible to dedicate any contiguous 32K segment of the virtual address space for a PDP-11 (user mode) program by appropriate setting of the relocation field in the mode register (see Appendix B). Because it is difficult, if not impossible, to demand page 11/40 code, the required pages must be resident in real memory. This poses

no particular problem - as 32K words can be swapped in approximately .3 seconds

The ootions for interfacing UNIX user programs to I/O devices are:

a) Implement a compatibility package to emulate UNIX system calls

b) Modify the UNIX system calls in key programs (RITA and RAND editor) so they will run in the INTERLISP operating environment.

c) Swap out LISP and swap in UNIX, turn the machine over to UNIX and let UNIX do the work. Of course this requires an addition to UNIX so it can "return" to LISP.

Option C is probably unsatisfactory as it does not provide any convenient communication between LISP and UNIX user programs. The choice between (a) and (b) depends heavily on whether the INTERLISP environment can be made similar to the UNIX environment without perverting the requirements of the LISP environment. We have not yet learned enough about UNIX to answer this.

LISP Kernel

The LISP Kernel includes the following modules:

1) storage allocation and garbage collection;

2) data storage and retrieval;

3) stack management;

(a) function call/return;

(b) stack allocation and deallocation (note that multiple environments make this a fairly complex process), and

(c) stack primitives - e.g. STKPOS, STKNTH;

4) Interpreter includes EVAL, APPLY, ENVEVAL, ENVAPPLY, PROG, COND, etc.;

5) other data primitives - such as strings and hash arrays;

6) Basic I/O:

(a) interfaces to the operating environment for page mapping, files, terminal, and other devices;

(b) terminal interrupt handlers, and

(c) atom hashing routine;

7) protection mechanisms for critical data such as file directories - Includes both protection against accidental destruction by user and protection against simultaneous access by asynchronous processes (e.g. user and FTP server);

8) compiled code interpreter.

The LISP kernel will be implemented in a judicious combination of machine language and microcode. Although it would be desirable to implement the entire kernel using microcode, it clearly will not fit in the available writable control store. Therefore, those portions of the kernel

---

*An exception occurs with garbage collection. There are a small number of garbage collection primitives (those that mark and chase pointers) that are executed thousand or hundreds of thousands of times during a garbage collection. In this case it is feasible to swap microcode.

which are both frequently used and simple (short), such as the compiled code interpreter, function call/return and portions of the stack allocation and deallocation code will be written in microcode. As for the other portions, while it is possible to swap microcode, the time to do the swap would generally exceed the time to accomplish the desired task with PDP-11 code instead.* What remains to be explained is how to interface smoothly the execution of straight PDP-11 code with LISP microcode, in particular how we intend to access the 4M word address space of the LISP environment while executing PDP-11 kernel code, and how to get along with only 1000 words of WCS.

While it would be possible for PDP-11 code to access the 4M word LISP address space by using Core Management windows, this would be both slow and awkward. A cleverer approach is to put to use some of the many PDP-11 opcodes (Floating Point, Extended Instruction Set) preempted by the insertion of the WCS, and redefine them as new opcodes using the WCS. All these automatically cause a transfer of control to the writable control store, where one can decode the instruction, perform the desired operation in microcode (where access to the 4M word address space is faster and simpler, see EXT register in MMD appendix), and then resume PDP-11 code execution.

The reverse operation, i.e.: to transfer control from

microcode to PDP-11 code will also be necessary. To see this, just consider that the compiled code instruction set contains 2048 miscellaneous operations (see Appendix A). While the most frequently executed ones will be microcoded, it is clear that 1K of WCS can not handle the whole set. Therefore, most of these relatively infrequent compiled code instructions will have to result in calls to routines in PDP-11 code. Another example is the set of operations involved in a return from a function. In the "spaghetti stack" (multiple environments) of INTERLISP there are two types of function returns, essentially the simple return and the return that requires an environment switch or environment copy. The simple return can be accomplished easily in microcode, while the hard return is probably both long enough and infrequent enough to be undeserving of microcode.

Both cases (the compiled code instructions and the hard return) are handled in a manner similar to a PDP-11 trap. That is, we save the state of the machine, where the state includes whether we were in the compiled code interpreter or in 11-code (executing a "new" instruction - e.g. RETURN); we perform the desired operation in 11 code; and we resume the interrupted state.

New LISP Code

As we noted before, a large portion of INTERLISP-10 is implemented in INTERLISP and may be transferred directly to INTERLISP-11 without modification. However, there is still a significant amount of additional LISP code that must be written for the PDP-11. The reasons are various -- some things are necessarily machine or implementation dependent such as the code generator for the compiler. Others are machine dependent (in INTERLISP-10) for speed, such as the arithmetic functions (SIN, COS, etc.), and READ and PRINT. Some things are provided by TENEX such as the file system.

Available Tools

Much of the required work can proceed in parallel. Those portions of the system which will be written in INTERLISP can be written and debugged using INTERLISP-10. CMU has provided an assembler and simulator for the microcode that runs on TENEX. The simulator does not capture all the idiosyncrasies of timing problems in the microcode but does facilitate initial debugging. We have a cross-net debugger for the PDP-11 that runs under TENEX and communicates with the 11 via the IMP interface. Since it doesn't suffer from the space limitations of most PDP-11 debugging aids, it is much more useful.

Present Status

Hardware

All of the hardware, except the MMD, is installed and working as of 1 May 1976. The MMD has been designed and specified. Delivery is expected in early June. Following delivery we will probably need to do some final debugging (maximum 2-3 weeks).

Software

The major component of the operating environment, the page manager, has been designed and partially written. In the LISP kernel, the compiled code instruction set is designed as well as most of the data storage formats and the stack management.

Prognosis

We expect to have an INTERLISP-11 system that is useful for testing and demonstrating some on-line help and tutorial facilities for at least one of the tools that will eventually be available under the IT. This we should be able to do by the end of the contract period. However, some components may have to be omitted due to lack of time. The garbage collector, for example, is not essential for a demonstration system since we can use LISP for a long time without filling a 4M word address space.

Schedule

To the best of our crystal ball abilities, the following appears to be a realizable schedule of INTERLISP-11 development during the remainder of the contract.

By the end of the 3rd quarter (30 June) we expect to have

1) MMD delivered, debugged and accepted
2) An operating environment complete in its essentials - possibly omitting auto dialer
3) The compiled code interpreter, stack management, and some of the data allocation, storage and retrieval.
4) The compiler done
5) Read, print, and file system designed

4th quarter (30 September)

1) Complete kernel, integrate with operating environment
2) Complete READ, PRINT and file system
3) Complete necessary LISP code (omit some arithmetic functions for example)

5th quarter INTEGRATE AND DEBUG, AND USE

SECTION 3. "INTELLIGENT" ON-LINE ASSISTANTS AND TUTORS

Introduction

We envision the learning of a computer-based tool by a computer-naive person as a multistage process. The first stage is one where a basic conceptual framework is built-up. It is here that the new user acquires the meaning and a sense of the relative importance of key concepts that are almost completely new to him. Learning at this stage can take place effectively only if the new user is "taken by the hand" and led through the maze of new knowledge by an expert that presents to him carefully selected amounts of new information and shields him from knowledge that cannot be understood at that point. This "systematic teaching" phase must be bolstered with examples, demonstrations ("let me show you how to do this") and supervised practice. The latter is a factor of fundamental importance: the new user must be posed problems that can be solved with what he should have learned so far, and his solutions and results be scrutinized. If he makes an error, the nature of that error must be explained, and some remedial material offered.

Once this basic conceptual framework has been established, learning can proceed much more independently. It is here that question answering pays off handsomely, not only because the answers are frequently complete in

themselves (they do not engender new questions) but because the no-longer-naive user knows what questions to ask.

In the rest of this section we describe our efforts in creating an INLAT for the Hermes message system. We begin by describing our work in designing tutorials for systematic teaching, we continue with a description of our work in question-answering, and we close with a description of the control structure that holds the whole system together.

Tutorials for systematic teaching

As alluded to in the Introduction to this Section, while effective teaching still depends on describing facts, actions, purposes, procedures, etc. symbolically, the most effective elements of the teaching situation are the ostensive ones, namely:

1) teaching by letting the students do things by themselves and helping them correct their mistakes.
2) teaching by way of examples
3) teaching by demonstrating actions (the tutor typing commands for the new user, for example, when a complicated new command is being introduced or when he is unable to proceed)

A very useful way to proceed is to write something like a script for a series of lessons or, in other words, a scenario for the form the INLAT adopts when in tutorial

mode. It is a way of exploring how to skillfully organize, segment, present, and sequence knowledge about the subject matter in a manner that results in easy and comfortable learning.

When a new user asks the INLAT to teach him something systematically, he is presented with an agenda-driven sequence of tutorial units. The elements of these units are:

a) delivering information
b) asking questions of the student
c) showing examples
d) demonstrating actions
e) requesting the student to perform tasks and exercises, evaluating them, and making the appropriate comments to the student
f) pausing to answer questions from the student

Elements a) and f) are always present.

The procedure is as follows. The INLAT presents exposition, embedded in which is a series of tasks. Fairly frequently, the system stops to ask whether there are any questions. If the student has no questions the exposition continues. If he has a question he types it, and when the question has been answered, he can either keep asking questions or allow the system to proceed with its tutorial exposition.

Whenever a task is proposed, the INLAT puts the new user in touch with the tool being taught. After he is done with the task, the system evaluates his performance of it. If he has done the task correctly he will be praised and the exposition will continue. If he has done the task incorrectly his mistake will be pointed out to him, his work space restored to its form before the task was initiated and he will be asked to do it again. He may ask the system to show him how to do it, or even ask the system to do it for him if he is in real trouble.

Much thought was given to how much material would be sufficient for allowing an individual to comfortably use Hermes. We decided to focus our attention on several areas including simple message processing (reading, filing, deleting, listing) simple message sending (creating a message, replying, forwarding), and several object creating tasks (classifying mail by means of message sequences, operating with different files, etc.) We felt that this material is best delivered in small lessons of two or three pages. In addition, we kept in mind that effective teaching of a new system such as Hermes, requires repetition of key material.

Methodology of building tutorial material

The methodology in building these lessons is in itself

interesting to those who would design such material. There were many iterations in going from the initial draft to the finished product. (We use the term "finished" very loosely, for we are sure to find more weak areas in the tutorial as testing continues.) We initially designed an outline of topics that we wanted to cover, and then broke them down into groups which later became lessons.

The real testing of this material came when we ran subjects using the tutorial. Our method for doing this was simple. The material was not on-line, but was handed to the student in booklet form. The booklet contained groups of text that included descriptions of tasks. At the places in the booklet labelled "TASK" the student, already sitting inside Hermes, performed the tasks on a specially designed message file. We monitored the subject through a computer link, two terminals talking to each other. When the student had questions or needed assistance he typed a semi-colon and his request. The "tutor" (a human at this point) then responded to his request. (A description of the kind of requests encountered follows.)

We were especially interested in the "bugs" exhibited in the student's conception of Hermes' commands and actions. For example, the delete command has a non-visible side effect -- it marks a message for deletion but does not remove it from the message sequence until the user types

expunge or leaves Hermes. To counter any confusion that the student might have about deleting, these effects were made explicit by telling the student exactly what happens when he deletes a message. It is very important to make explicit the hidden effects of a command, especially a powerful command like delete that can cause exasperating results if not understood completely. Another bug of this type is when to follow a filter with a colon and when not to. There is a rule that goes like this: if a filter is also a field name, follow it by a colon; otherwise there should be no colon. Although we did not explicitly state this rule in the text (as maybe we should have), we did point out when a filter must not have a colon. We are sure that hints like these greatly reduce the amount of confusion inherent in learning a new system. It is easy to get hung up, to manifest this bug, and simple enough to include a sentence or two to alleviate the problem before it becomes a point of real confusion.

Sometimes these bugs were handled by including text to warn the student of the pitfalls. Another way of dealing with the problem was to anticipate it and rearrange the tutorial. For example, the section on "panic buttons" was moved to the front of the tutorial to allay the student's fear of getting stuck in a problem situation. Forward and Reply were moved to before the Compose command in the belief that they are less complicated to learn and are really

simpler versions of Compose. The section on current objects was put last because it is information that may be more than the student wishes to deal with.

The third way of dealing with "bugs" was to interact with the Hermes project on system design. For example, the version of Hermes we were dealing with in the testing of the tutorial did not make explicit the existence of partially completed drafts, which users often forget about. Without this information, the student may enter the draft state again and begin creating what he thinks is a new message. But in fact the old draft is still there and what he is really doing is appending to each of the fields. The Hermes project has handled the problem by making explicit (telling the user) whether there is a draft sitting around that hasn't been sent.

We have described a methodology for creating tutorial material that includes adding hints to the document, reordering material to anticipate difficulties, and making suggestions for the modification of the system, Hermes, itself.

The Tutorial as a Stand-alone Document

Another version of the tutorial only slightly different from the one used in the tutoring session above is a descriptive document that stands alone, apart from the

computer delivery. It excludes tasks which the student can do and instead presents examples of the Hermes activities explicitly. It is not "learning by doing", but "learning by watching". It is a surprisingly robust document that people have found helpful, so much so that it is being put out as the user's introduction to Hermes distributed by the Hermes project. It is included in this report as Apperdix C.

Question Answering

We are convinced that a really useful on-line assistance facility for computer-naive users will have to be based on an ability to understand questions and requests posed in English. However, as is well known, Natural Language Understanding by computer is an extremely difficult problem area in Artificial Intelligence and we believe that machine comprehension of completely free and unrestricted English is still a distant goal. Therefore, our efforts in this area must be necessarily qualified and delimited if we are to produce a usable system within the period of performance of the present contract.

The main realization that we have to come to terms with is the limitedness of the range of English that we can make our INLAT comprehend, and the breadth and depth of stored knowledge we will have available to produce answers to user´s requests. Past experience with a previous INLAT-like system, NLS-SCHOLAR, showed us very clearly how inadequate even a relatively sophisticated system can be when subjected to unrestricted inputs from users unaware of the system´s limitations.

To make habitable and useful a system that inherently cannot be robust enough to withstand the onslaught of incompletely specified, vague, ambiguous, and colloquially expressed requests, we need at least: a) to teach the user

the kinds of requests the system can handle and those it cannot handle, b) to incorporate facilities for guiding users to formulate acceptable requests "on-the-fly" (for example, guidance on what kinds of continuations would be acceptable given the beginning of a question) and c) to devise more elaborate partial comprehension facilities (i.e.: the system mutters to itself "I don't know what he means by ..., but within that general area I only know these few facts. So I'll show him what I know and let him choose").

Collecting Sample Questions

The first task we undertook was to collect, classify and analyze a large number of genuine questions about Hermes. Authenticity, rather than contrivedness of these questions was ensured because we solicited them at the time Hermes was being tentatively released, and because we offered our services in trying to provide answers for them. Thus, the questions reflected genuine doubts users were having about a newly available tool.

One of the things that became apparent from the beginning was that while a large number of these requests appear answerable with state-of-the-art Natural Language Understanding techniques, others are very hard to handle . This shows that even a "restricted" domain, such as Hermes,

is fraught with difficulties, both in the complex surface structures and in the amount of implicit knowledge needed to "understand" those requests.

Some appreciation of the difficulty in machine understanding of these hard requests can be gained via a few specimens in our collection:

1) "What if I instead of typing a part of the word print followed by an escape I just typed part of the word followed by a space and the message #?"
2) "If I use the "copies" subcommand of the "reply" command, are the addresses on the cc: and bcc: lists kept on them or are they moved into the to: list?"
3) "Why don't the messages start with 1,2,3...?"
   (He has just seen an initial survey which only prints messages marked recent and unseen.)
4) "What do I do to read let's say the last message without reading every message in between?"
   (He has just learned about LF and ^ but has not been told about the Print command.)
5) "Now what happens if I want to see a preceding message. (not necessarily the one directly before but let's say 16) even if you've already bypassed the message?"
   (A complex paraphrase of the preceding request.)

One can see tnat a system able to understand these questions would have to:

a) be tolerant of dubious English constructs, as in 1;

b) be able to model the situation described in order to provide an answer by just trying it out, as in 2;

c) be able to make the correct anaphoric references ("the messages") and have an internal model of the user's knowledge to provide the pertinent explanation, as in 3, 4, and 5.

These examples also illustrate how important it is to train the user how to ask questions so that the system might have a chance of understanding them.

On the brighter side of the picture, we realized there were a large number of questions that arise naturally and for which we could successfully provide answers. These questions include those about:

1)	procedures	(How do I do something)
2)	Purposes	(What does a certain command do)
3)	Definitions	(What does something mean)
4)	Instruments	(What command performs a certain action)
5)	Differences	(What's the difference between ..)
6)	Possibility	(Can I do something) (Does a certain action have a certain effect)

Analyzing the sample questions - Vocabulary, Syntax, and Semantics.

In order to characterize the questions we could feasibly answer, we performed an analysis of the collected requests. In general, there are two problems when dealing with a corpus of utterances that map into a finite knowledge space -- a so-called "closed world." Basically, there are many ways of saying the same thing, and there are many distinct things to talk about. Let's be more specific.

The first problem is represented by the high number of utterances to which we normally attribute the same meaning. This requires understanding how to extract that single deep structure representing the meaning, from the many surface forms in which that meaning can be expressed.

The second problem is what are the distinct meanings contained in the corpus of utterances, and by extension and generalization, what are all the possible distinct meanings contained in the knowledge space. This requires designing a deep representation for those meanings, in terms of factoring them into a fewer number of "primitive" meanings. To handle efficiently a large number of distinct meanings we want to "factor" as much as possible - we want to create a class of articulated structures in which each part can be one of a small number of possibilities, and in which the interpretation of the whole is a simple function of the interpretation of the parts and their relationships. A

natural representation having most of these desired characteristics is the Semantic Network, that will be described shortly.

Our first attempt at analyzing the collected requests reflected the general approach delineated above. We classified those requests in terms of the surface main verb and the classes of nouns involved, and in terms of what single Hermes command they referred to.

Parsing.

To handle a large number of distinct surface structures having a fewer number of distinct meanings, the first step is parsing. This creates a standardized syntactic structure in which the meaningful elements of the surface structure are arranged into a smaller set of canonical positions. Thus, a first stage of the factorization process alluded to before is accomplished: some very different phrasings having equivalent meanings are reduced to parsed structures with only small localized differences.

To this end, we investigated several complex parsing systems, including the Lunar System and the General Syntactic Processor (GSP). Both of these parsers are syntax-driven and produce a parse that has much of the structure of a Case analysis. An example of the parse of the same sentence from both LUNAR and GSP is shown next.

```
(PRINT THE CURRENT MESSAGE AT THE LINEPRINTER)
***
PARSINGS:
S IMP
  NP PRO YOU
  AUX TNS PRESENT
  VP V PRINT
     NP DET THE
        ADJ CURRENT
        N MESSAGE
        NU SG
        PP PREP AT
           NP DET THE
              N LINEPRINTER
              NU SG


41**. PRINT THE CURRENT MESSAGE AT THE LPT:
***
Parse 1:
1: [LABEL = S
    MOOD = IMP
2:  SUBJ = [LABEL = NP
3:          HEAD = [LABEL = PRO
                    ROOT = YOU]]
4:  FVERB = [LABEL = V
             TENSE = PRESENT
             ROOT = WILL-MODAL]
5:  HEAD = [LABEL = V
            UNTENSED = T
            PNCODE = X3SG
            TENSE = PRESENT
            ROOT = PRINT]
6:  OBJ = [LABEL = NP
7:         DET = [LABEL = ART
                  ROOT = THE]
8:         ADJS = ([LABEL = ADJ
                    ROOT = CURRENT])
9:         HEAD = [LABEL = N
                   NUMBER = SG
                   ROOT = MESSAGE]
10:        MODS = ([LABEL = PP
11:                 PREP = [LABEL = PREP
                            ROOT = AT]
```

We can see that although GSP makes explicit what fills each case frame, Lunar is designed to allow for this also;

the output printing routines could be modified to mimic GSP's behavior. The final decision of which parser to use has been delayed to allow for the introduction of a more robust GSP parser which should be available in the very near future.

Semantic Interpretation.

We then explored what it would take to extract the meaning (semantic structure) of a request given its parse. Since the final representation was expected to be equivalent to a single Hermes command, and there seemed to be a rough relation between the main surface verb (head) and the underlying command, we investigated the collected "procedure requests" with this in mind. The first step was to break them down by the surface verb to see the kinds of Cases the verbs can take and how the procedures can differ when taking into account the contents of these Cases. This is one example:

The verb COPY

Sentence	relevant Hermes command
Copy a message to a file.	FILE
Copy a message to the lineprinter.	LIST
Copy a filter into a named filter.	COPY
Copy a message into the draft.	FORWARD
Copy a message into a field.	CTRL-B, ADD-FILE

In all but one example, the Object Case is message, but the Hermes command that is being requested is dependent on the

contents of the destination Case (file, lineprinter, filter, draft, field).

The next breakdown was the grouping of sentences by the Hermes command that is being requested. With that breakdown we could analyze all the various surface verb and Case contents that can refer to a single Hermes command. For example, the command List can have a variety of surface representations:

The Hermes command LIST

Get a listing of the message.
List the message.
Output a message to the lineprinter.
Copy a message to the lineprinter.

As we investigated this relationship further we found that there were many simple syntactic structures in plausible sentences that could not be mapped to parts of a single Hermes command. In other words, the user's natural language concepts correspond to something larger than a single command. This led to the idea of a conceptual command - a conceptually unitary action that the user desires Hermes to take. The conceptual commands fall into natural classes whose elements differ only in the objects to be affected or the details of the operations to be performed. Such classes of conceptual commands do not always map to single Hermes commands with varying parameters. In general they map to a "MACRO-command" - a

sequence of (optional) Hermes commands whose arguments can be filled in to provide a Hermes procedure for any example of the class of conceptual procedures.

As an example of such a class of conceptual commands consider the class of "display Hermes structure" commands. These commands all involve showing the user (by producing character string output on some file, including the users terminal, the line printer or a disk file) some portion of a specified collection of Hermes objects. To specify a particular display command we must specify several Cases (parameters of the class of conceptual display commands.) Some examples of Cases are:

Class - the type of object to be displayed (message sequence, switch, command, description);

Restriction - characteristics defining those objects of class which are to be displayed (e.g. filters like FROM: AIGHES, names like DELETE SWITCH);

Output template - description of a part of each object which is to be displayed, as well as formatting information (e.g. fields of message, default value or current value for a switch);

Destination: file to send output to

Given the contents of these cases, we can choose the relevant conceptual command from the class and fill in the arguments of its related MACRO-command. With the

filled-in cases we can also construct procedures that look at the values of the cases and manipulate them to achieve an understanding of the user's request.

We will now follow the interpretation of a request "Tell me the character count of messages 41 through 45". The cases we will try to fill out are:

Class
Restriction
Template
Destination

I. Class: Class is the object we are concerned with. The class is "message".

II. Restriction: Made on the class. It is "message number between 41 and 45".

III. Output Template: What part of the object is to be displayed. It is "character count".

IV. Destination: We assume it is the terminal, TTY:.

To go from these conceptual cases to arguments of a MACRO-command we need several procedures including a template maker, a sequence maker, and a display command maker. We indicate below some of the decisions to be made in each.

Template maker: Is there a template that already exists that does this? Yes, there is Ptemplate and Stemplate. Which is better? Stemplate because it contains fewer other things. Would it be better to create a new template that only does

this one thing -- give a character count? Yes.

```
>CREATE TEMPLATE TEMP
>>LINE-INSERT END CHAR-COUNT
>>DONE
>
```

Sequence maker: Creates the correct message sequence given the Class and the Restriction. In this example, 41 through 45, or 41:45.

Command maker: Should we choose the most specific or the most general command to perform this operation? The most general is Transcribe, the most specific, Print.

Tell me <msg/part> of <msg/seq> =>

Print <msg/seq> <template> <destination> =>

Print (messages) 41:45 (using filter) temp (on file) tty:

In summary, we have sketched out how we are attacking the question-answering problem. A break down and analysis of requests has begun. We now see as the next step working on a procedure for using the cases once they are filled in. In addition this next quarter we will settle on a parser and proceed to exercise it.

Semantic Network

In order to answer questions about a program like Hermes, an on-line assistant needs to have readily available

a store of factual information about the program. Given that the system is a large and complex as it is, it becomes imperative to make a judicious choice of both the nature of the information to be stored, and its representational format. In addition, any data structure which is expected to represent such potentially sophisticated information must be designed with careful attention paid to the procedures that will access it. No matter how elegantly the knowledge is represented, it is not useful unless it can be found and applied when needed.

Our first attempt has been to encode basic descriptive (static) knowledge about the internal structure of the Hermes program. Such information is the kind necessary to answer simple factual queries about Hermes commands and objects. For example, questions like:

1) "What are the arguments to the PRINT command?"
2) "What are the parts of a DRAFT message?"
3) "What commands operate on messages?"
4) "What is the difference between the SURVEY and SUMMARIZE commands?"

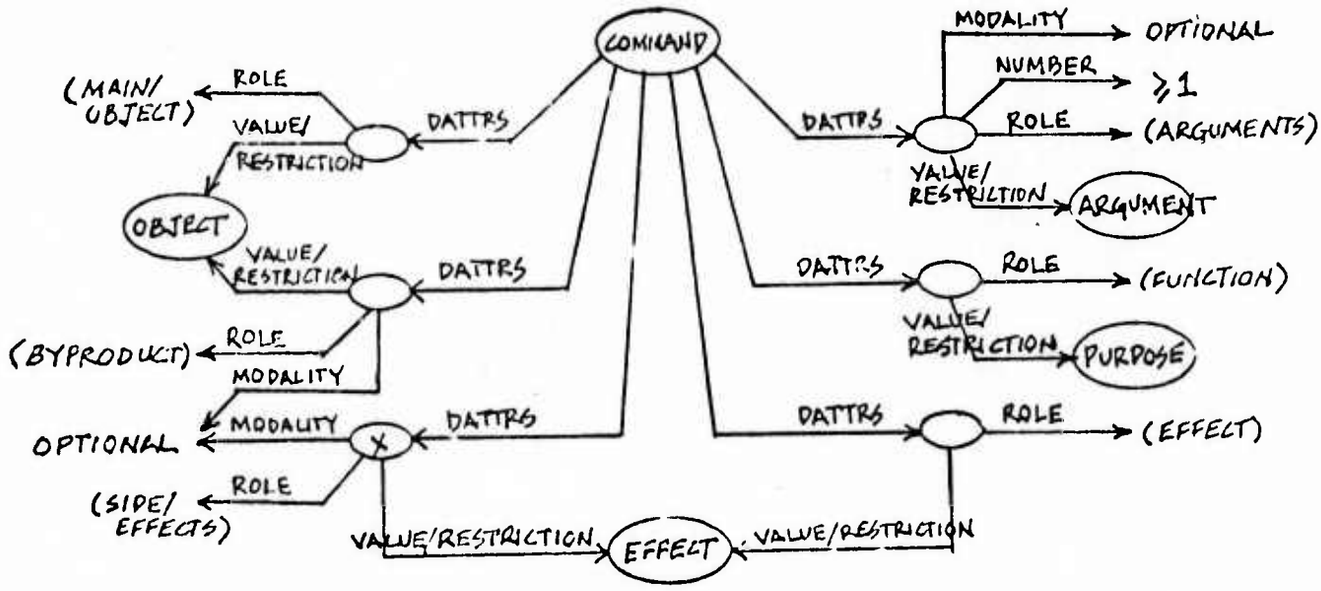
would be answered by a system whose knowledge base includes a straightforward record of the structure of Hermes commands, the parts of Hermes objects, and the objects manipulated by commands.

Since questions about such things can be phrased in many ways, and make use of many different arrangements of concepts, it makes sense to represent the information in

terms removed from those of natural language. To this end, we have chosen to represent a description of Hermes based on the semantic network formalism. The main value accrued from a network structure like the one we are using is the ability to store information about many objects in one place (usually called a "concept node"). Since one node represents an entire class of things, an assertion stored at the node is implicitly made about each individual member of the class.

We have derived a discipline for creating the Hermes Net that diverges from standard semantic net building. For our links, we use only a small set of primitively-defined link types, which explicitly reflect the basic epistemological operations of the network representation, Thus, any conceptual relation (one related to the Hermes world) is build out of primitive links, and is never implied to be primitive. While the extra "layer" of structure makes the network somewhat more difficult for a human to read, it avoids the ambiguities and type confusions invariably found when the underlying operations are glossed over.

For example, here is how we would build a node for COMMAND:



The relation DATTRS always points to a description of a part or attribute of the thing being defined. Here, we see that node A defines the "ARGUMENTS" part of a command. The MODALITY states that command-arguments are optional; if there are any, there can be more than one, as indicated by NUMBER pointing to "≥1". The ROLE link specifies the functional role of this part of a command in relation to the command as a whole. If a concept node like COMMAND is thought of as a case frame, the xxx specify the case definitions. ROLE names the particular case. Finally, VALUE/RESTRICTION points to another concept that defines the class of values legal as fillers for the case.

We have represented many of the Hermes commands and objects in this manner. We will not go into the representation in any more detail here but merely point out

that, since the underlying set of relations used in our net is fixed and well-defined, the procedures we build will be able to operate consistently. This is the main reason for breaking up the basic operations into individual links.

Notice in the figure that Hermes COMMANDS have a part called a FUNCTION. We have found that, while a network that straight forwardly reflects the structure o the program is necessary to answer simple factual questions, it is not adequate to handle even the simplest of functional questions, e.g.

1) "How do I read a message?"
2) "What is the second argument of PRINT for?"
3) "What are COMPOSE-TEMPLATES used for?"

Such questions require a functional view of the Hermes world. We have begun to represent PURPOSES in the same kind of network notation that we are using for the commands and objects. This net takes notions like CREATING, DESTROYING, CHANGING, MOVING, etc., which are the ways the user thinks of what Hermes does, and provides a structured classification so that relevant operations are easily accessible from related ones. We believe that this type of structuring will facilitate the inferences necessary to go from the natural language phrasing of a functional question to a well-defined conceptual structure for the query.

It has also become clear that "functions" do not map

one-to-one onto the commands and objects. We are studying a mechanism for interfacing the two networks, and believe that this may lead to a powerful tool for integrating different models of the same factual world.

Current Control Structure

The various modules that compose the INLAT (Natural Language Understander, Tutor, Hermes, Question Answerer, etc.) are held together by a Control Structure presided over by a monitor. Our notions in this regard are still evolving and the Monitor we shall describe next represents only our current approach. As such, it may very well change in the next few months.

The Monitor oversees all dialog between the user and the system. It is its job to read inputs from the user, analyze the input to extract contextual information, examine it for 'level of sophistication' and both syntactic and semantic error checks, record each piece of dialog on a history list (there are several history lists -- see subsequent discussion), and decide how to handle the input, i.e., which module to pass it to. Currently, the monitor attempts to parse the input as a Hermes command, and if it succeeds, it passes the command directly to Hermes; otherwise it hands it to to the Natural Language Understander to be parsed and then to the QA

(Question-Answerer) to be processed. Also, the monitor takes care of all interfacing between the user and Hermes, protecting each from the other. It sets up necessary network links, creates a copy of the user's mailbox for its own internal use, and maintains a user profile (which includes the Hermes profile and other information). In summary, the monitor is the central component that gets everything going and directs the whole operation.

The monitor also will have to 'evaluate' what the user is doing in relation to some model or notion of what the user is trying to accomplish. As a simple (and possibly unrepresentative) example of how the monitor watches over, assume the user is in an editor doing subcommands and wishes to abort a subcommand but mistakenly types ^E (taking him out of the editor). He then attempts to redo the subcommand not realizing he is at the top level. The monitor should inform the user what happened and get him back where he was in the editor.

The monitor takes care also of correcting mistakes such as typing 1*3 instead of the intended message specification 1:3, or of understanding what the user means when he types 'Print template' instead of 'Show template' or 'Send switch' instead of 'Export switch'.

Thus far, there have been two versions of the Monitor constructed. The first was a 'quick & dirty' version that

didn't do much more than establish the network couplings, do all interfacing, I/O and interrupt handling, maintain a simple profile, and delegate inputs to either NLF and QA, or to Hermes. The main problem with it was its mode of operation: having to constantly send bits of information across the network and then wait for a response before it could continue. This made real-time operation impractical due to the long delays involved. This version was abandoned over a month ago.

The current version is not yet complete, but is close to being operational. This version does not send anything across the net until it has gathered a complete Hermes command to be executed, i.e., not until it is absolutely essential -- no net delays are incurred that can be avoided. Also, the current version does a good deal more than the previous, as it is not a 'quick & dirty' monitor, but rather more along the lines of what the final one should look like.

As well as doing all of the nitty-gritty work necessary for the Hermes interface over the net, the monitor does a certain amount of syntactic error correction, maintains rather complete and useful history lists, and attempts to construct a good internal representation of user inputs. The monitor operates to a large degree by emulating Hermes. It contains a complete parser for Hermes Commands, a partial simulation of Hermes, and also a simulator for certain parts

of Tenex. (The Tenex simulation is used principally for file recognition and confirmation, which Hermes passes to Tenex to read, but also for various utility commands such as JOBSTAT, QFD, DIRECTORY, etc.)

Among the other things that it is doing, the monitor is maintaining a complete history of everything that occurs during the session. This is done on a number of different 'history lists' (not LISPX history lists). Each command line (a question is considered a 'command') begins with a line number and prompt character. This line number is then the index for everything that is recorded on the history lists up to the next command line. The things that would go on the list would include: the raw line read from the user; any modifications or interpretations or parsings of that line; any Inlat dialog with Hermes (invisible to user) needed to process the line; the final Hermes command issued; and the corresponding response, if any. In addition to these lists, there are a number of 'context' variables that are maintained. These will be the key to extracting an interpretation that can be used by the Inlat system. The idea is, as of this moment, that a chain of 'history trees' will be kept. The root of each tree is on the chain in sequential chronological order. The tree growing off of each root then represents all interactions and subtasks and

subgoals that were used to accomplish the task initiated by the root. This corresponds to the subcommand structure within Hermes itself. Although we cannot yet be too specific about these trees, they will contain all information pertinent to a given task -- the only question is what form it will have, i.e. how it will be compressed and encoded.

APPENDIX A - SPECIFICATIONS

I. INSTRUCTION SET

The LISP compiler will produce code in the instruction set described below. The code is interpreted (run) by the micro-code.

In the following description TOS means top of stack; the quantity on the top of the stack is referenced, S as a source means the quantity on the top of the stack is popped. S as a destination means the result is pushed on the stack.

Group 1

4 bit op 1 bit sub-op 3 bit sourcetyp 8 bit offset.

source types are:

STACK - temporary value in current frame extension. Offset is a positive quantity representing a negative offset from the current stack pointer.

LIT - literal of the (compiled) function now running. Offset is a positive offset from the current function literal pointer.

LOCAL - local variable in the current basic frame. Offset is positive from current basic frame pointer.

SPEC - specvar reference. An indirect reference to a value cell. In compiled functions the value cell pointers are in the literal area, and in interpreted functions the value cell pointers are in the basic frame. In either case the so-called literal pointer is the base for specvar references. The offset is positive.

SYSTEM CONSTANT - Constants that are referenced frequently, such as T and NIL, are stored in a system literal table. The offset is positive from the beginning of the system literal table.

IMMEDIATE - immediate small numbers in the range -63 to +64. The offset is the number minus 63.

op	sub-op		
ØØ	Ø	PUSH	E->S
	1	RET	return (E)
Ø1,Ø2	unused (possibly IF TRUE, RET and IF FALSE, RET)		

Group 2

4 bit op 2 bit sub-op 2 bit sourcetyp 8 bit offset

source types are:
STACK, LIT, LOCAL, SPEC.

op	sub-op		
03	0	CAR	CAR(E)->S
	1	RCAR	(return(CAR E))
	2	CDR	CDR(E)->S
	3	RCDR	etc.
04	0	CAAR	
	1	RCAAR	
	2	CADR	
	3	RCADR	
05	0	CDAR	
	1	RCDAR	
	2	CDDR	
	3	RCDDR	

Group 3

4 bit op 2 bit subop 2 bit source type 8 bit offset

op	subop		
06	0	TOS	TOS->E
	1	POP	S->E
	2	PCAR	CAR(S)->E
	3	PCDR	CDR(S)->E
07	0	SCDR	CDR(E)->E
	1	SCDDR	CDDR(E)->E
	2	ADD1	E+1->E
	3	SUB1	E-1->E

Group 4 - arith.

4 bit op 2 bit subop 2 bit source type 8 bit offset

op	subop		
10	0	ADD	TOS+E->TOS
	1	SUB	
	2	MUL	
	3	DIV	
11	0	REM	
	1	EQ	COMPARE TOS with E and POP, set indicators.
	2	>	
	3	<	

Note that the quantity on the top of the stack can be either a pointer to a number (a boxed number) or a tagged unboxed number. The result can be stored on the stack as a tagged

unboxed number.

Group 5 - type tests

4 bit op 2 bit subop 2 bit source type 8 bit offset

op	subop	
12	0	LISTP test E, set indicator
	1	ATOM
	2	LITATOM
	3	NUMBERP
13	0	FIXP
	1	STRINGP
	2	ARRAYP
	3	STACKP

Group 6 - branch

4 bit op 2 bit subop 10 bit offset

Offset is 9 bits + sign, 0 implies long branch where 16 bit offset is in the following word.

op	subop	
14	0	BRT branch if indicator true
	1	BRF branch if indicator false
	2	unused
	3	BR unconditional branch
15	0	BNIL branch if TOS=NIL and POP
	1	BNN branch if TOS NOT NIL and POP
	2	NBNIL if TOS=NIL, branch and don't pop; if TOS NOT NIL, don't branch and do pop.
	3	NBNN if TOS NOT NIL, branch and don't pop; if TOS=NIL, don't branch and do pop.
op 16	unused	

Group 7 - literal references

8 bit op 8 bit offset

op		
1700	CALL	call function, literal is #args,, fnname
1704	DCALL	call function, discard value
1710	LCALL	linked call to function
1714	DLCALL	linked call, discard value
1720	BIND	used for prog's and open lambdas that make frames

1724 TYPTST test type of TOS, arg. is type number (immediate)

1730-1734 unused

Group 8 - miscellaneous

16 bit op, 174000-177777
2048 miscellaneous operations, args. if any, are on the stack.
For example:

CONS	
MEMB	
ASSOC	
ELT	
SETA	
RPLACA	
RPLACD	
DRPLACA	(discard value)
DRPLACD	"
DPOP	(pop and discard)
NLGO	non-local GO in PROG
NLRET	return from PROG when inside an open LAMBDA
APPLY	
EVAL	
APPLY*	

II. STACK FORMAT

The stack is allocated in a fixed 64K segment limiting stack pointers to 16 bits.

Frame Extension

```
[FLGS][USE]     16 flags are easy/hard, active/inactive
SIZE or END     16
BLINK           16
ALINK           16
CLINK           16
RETURN          32


TEMPORARIES     32 bits each
```

Basic Frame

```
       BINDINGS     32 bits each


BLINK->[FLGS][CXT]  16 flgs are active/inactive
       [FRAME SIZE][#ARGS] 16
       FRAME NAME   24
       FEF or VCELLS   24
```

Holes

```
          -1        16 HOLE FLAG
          SIZE or END 16
          LAST HOLE 16
          NEXT HOLE 16
```

Frame extension is all obvious. Tems are 22 bit pointers. High order 10 bits can be used for magic markers such as:

0) POINTER
1) 22 bit integer
2) UNBOXED stack pointer
3) or anything else that can be stored in 22 bits
4) ALL ONES RESERVED (see Holes)

The basic frame overhead follows the bindings so that bindings can be stacked directly without either moving them to build the frame or having a special operation to begin a function call by skipping some stack locations. If the function is ccmpiled, the high order 10 bits of a binding

contain the relative location of the value call pointer in the FEF. If the function is interpreted, value cell pointers are in the basic frame following the framename and the high order 10 bits of a binding contain the relative locaton of the value cell pointer in the basic frame.

Stack holes are chained both frontwards and backwards, permitting quick searches for available holes and also permitting the removal of any hole from the list.

When a frame is about to be run, it is desirable to be able to check whether a stack hole immediately follows the extension. Thus a hole mark must be distinguishable from any binding, and from a frame extension header. This is accomplished by prohibiting the value -1 in the high 10 bits of a binding, and -1 as the value of USE. To remove a hole, H, from the chain,

```
(NEXT(LAST H)) <- (NEXT H)
(LAST(NEXT H)) <- (LAST H)
```

Hole merging is often useful and is accomplished as follows:

```
IF    X+(LEN X) = (NEXT X) then

   (LEN X) <- (LEN X) + (LEN(NEXT X))
   (LAST(NEXT X)) <- X
   (NEXT X) <- (NEXT(NEXT X))
```

Holes that are smaller than 4 words are not in the chain. These can only be used if required by the frame above or if they eventually get merged into adjacent holes.

*NOTE: Using an FEF contained in the compiled code means that CHANGENAME of a compiled function variable must be illegal.

Size Comparison

	PDP-11	PDP-10
Extension overhead	112 bits	180 bits
Basic Frame overhead	80	36
compiled bindings	32 bits	36
interp. bindings	48 bits	36
Typical 3 args, compiled	288	324
3 args, interpreted	336	324

III. LIST STORAGE

A CONS cell can occupy one, two or three 16 bit words. Full pointers are 22 bits. The flavors of CONS cell are:

1) CAR special, CDR special - stored in 16 bits
2) CAR special, CDR long - stored in 16 bits
3) CDR special, CAR long - stored in 32 bits
4) Both long - stored in 48 bits
5) INVIZ1 - invisible pointer - short - the cell has been RPLACA'd or RPLACD'd and the new value would not fit in a 1 word CONS. The real cons is offset from here. 0 means it is in a hash table.
6) INVIZ2 - 2 word invisible ptr

FLG	TYPE	FORMAT
000	both special	[3 bit flg][7 bit car][6 bit cdr]
001	CDR long	[3 bit flg][7 bit car][6 hi bits full cdr] [16 low bits full cdr]
010	CAR long	[3 bit flg][U][6 hi bits full car][6 bit cdr]
011	Both long	[3 bit flg][U][6 hi bits car][6 hi bits cdr] [16 low bits CAR] [16 low bits CDR]
100	INVIZ1	[3 bit flg][13 bit offset]
101	INVIZ2	[3 bit flg][UUUUUUU][6 hi bits actual addr] [16 low bits actual addr]

[U...] denotes unused bits. 7 bit CAR's are encoded as follows: bit 6 = 0 - remaining 6 bits 0 CAR is NIL, otherwise CAR is a list offset from this location by +/- 31 words. 6 bit CDR's are encoded as follows:
0 - CDR is NIL, otherwise CDR is a list offset from this location by +/- 31 words. INVIZ1 is used when the original CONS is one word long, the new cons will not fit in one word, and there is a free slot big enough within an offset of +/- 4000 words. One unique value is reserved to denote that the real cell is contained in a hash table. INVIZ2 is used when the original CONS was 2 words and the new CONS requires 3 words. Then the INVIZ2 pointer is the full 22 bit address of the actual CONS cell. Using the following data obtained by GREEN and CLARK we can compute the average number of bits required by a list cell.
29% of CAR's are lists. Of these 70% are within +/- 31.
72% of CDR's are lists. Of these 88% are within +/- 31.
25% of CDR's are NIL.
3% of CAR's are NIL.
11% of CAR's are numbers 0-15.
Thus, P(0)=.30, P(1)=.04, P(2)=.58, P(3)=.08. Assuming no INVIZ pointers the average number of bits per list cell is then 28.5 bits.

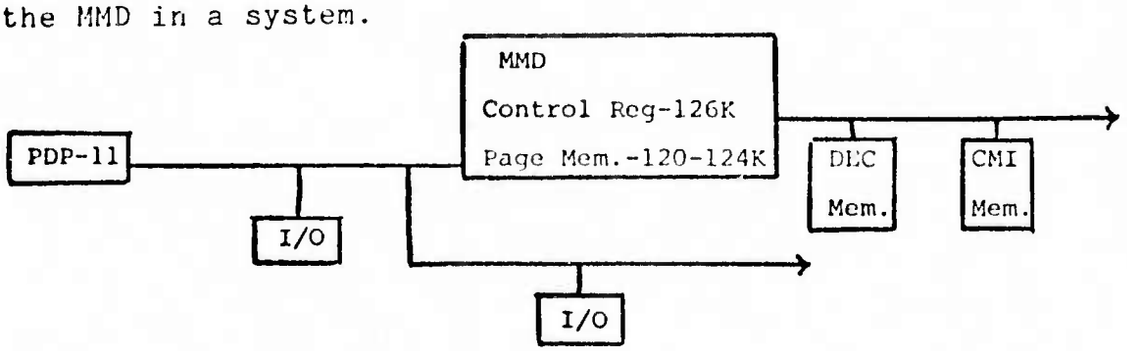
LIST ALLOCATION

To allocate list cells we will need a free list of words, one free list per page containing relative addresses within the page (10 bits). If a word on the free list has only one free word following, then a "next to last word in chunk" flag will be set in the word. Similarly the last word in a chunk will have a "last word in chunk" flag set. When allocating space for a new list cell, we will take one two or three words as needed. If the free pointer indicates that there is not enough space in the chunk, then just waste the words at the end of the chunk and step the free pointer to a chunk that is big enough. (Note that there is a bit of iteration in the above process but basically simple). As we will probably linearize lists occasionally, the small amount of waste should not matter very much. Also, given there are 2 unused CONS types, we could use one of the types to mark lonely 1 to 2 word holes. Then the holes could be used by RPLACA's to preceding cells.

APPENDIX B

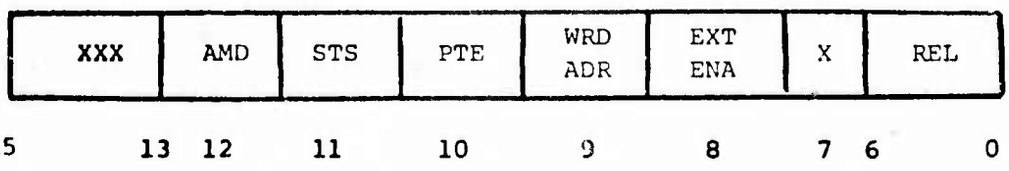
PDP-11 MEMORY MAPPING DEVICE

The PDP-11 Memory Mapping Device, to be referred to as MMD will accept Unibus memory requests from a PDP-11 processor or other bus master and generate paged memory requests on a second Unibus connected to all system memory. It shall contain a high speed 4K by 16 bit memory to hold the real core addresses of all virtual memory pages. The device will have control registers that will store the following functions: (1) relocation, (2) byte/word addressing, (3) disabling of paging and EXT registers, (4) write protection, (5) word modification detection, (6) age of pages, (7) distribution of ages, and (8) invalid page status information. The diagram below shows the location of the MMD in a system.



I Control Registers

1.1 Mode Registers (4), 764200, 764202, 764204, 764206



There are four mode registers which allow 4 different sets of parameters in using the MMD, Bits <17:16> of the PDP-11 bus address select the mode register to be used. This register is read/write. The definition of the register bits is below.

1.1.1 REL - Relocation. - This 7 bit quantity is added (or XOR'ed) to the high-order address bits for use in addressing the page table memory.

1.1.2 EXT ENA - Address Extension Enable - This bit when false causes the EXT register to be treated as if its contents were zero. When true the EXT register (combined with REL) is the high order part of the virtual address.

1.1.3 WRD ADR - Word Addressing - When true, word addressing is enabled (address is shifted left 1).

1.1.4 PTE - Page Table Enable - When true, the page table memory is used; when false, the page table memory is bypassed.

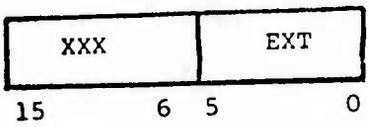
1.1.5 STS - Protect Trap Status - When true will not allow modification of information retained during trap.

1.1.6 AMD - Age Modification Disable - When true, no change occurs in the age section of the page table memory on memory references.

1.1.7 XXX - Non-existent

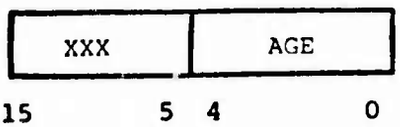
1.2 EXT Register (1), 764210

This 6 bit register is clocked from external, non-Unibus hardware and may be read or written by the processor



1.3 Age Register (1), 764212

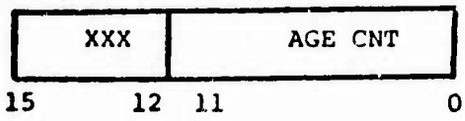
This 5 bit register can both be read and written by the processor. Its contents are entered into the page table memory during memory references unless inhibited by AMD true or PTE false.



1.4 Age Distribution Registers (64), 764000 - 764176

These 12 bit registers indicate the number of pages of

a particular age (and write modification) in the page table memory at any given time. The registers can be both read and written from the processor and are automatically updated by the MMD.



AGECNT - Age count - number of pages in the page table memory with the Write Modification Bit (05) and age bits (04:00) set to the address of this register.

XXX - Non-existent

1.5 AST(1), 764214

This 16 bit register, Address Stored on Trap, contains the low order 16 bits of the memory address requested that caused a fault. It is read-only.

1.6 DST (1), 764216

This 16 bit register, Data Stored on Trap, contains the 16 bits of data that would have been written into memory if a trap had not occurred. Its contents are only useful if the fault occurred on a write memory cycle. It is a read-only register.

1.7 PCST 764220

This 16 bit register, Program Counter Stored on trap, contains the contents of the PC at last CLKIR prior to trap. It is read-only.

1.8 MST 764222

This register, Map Status, contains the status of certain bits at the time of a trap. The cause of the trap, and the MMDENA bit. Bit 15 may be set or reset and bits 14:12 may be reset under program control. The rest of the register is read only. While any of bits 14:12 is set, trap information (AST, DST, PCST, MST) is not modified. When bits 14:12 are reset, updating of trap information resumes.

MMD ENA	ILLEG. PG.	WRITE VIOL	TIME OUT	XX	P17	P16	A17	A16	C0	C1
15	14	13	12	11 6	5	4	3	2	1	0

1.8.1 C1 - PDP-11 memory cycle control signal

1.8.2 C0 - PDP-11 memory cycle control signal.

1.8.3 A16 - 16th bit of address referenced at trap.

1.8.4 A17 - 17th bit of address referenced at trap

1.8.5 P16 - 16th bit of PC referenced at trap.

1.8.6 P17 - 17th bit of PC referenced at trap.

1.8.7 write viol - trap was caused by attempt to write on a write protected page.

1.8.8 illegal pg - trap was caused by reference to illegal page - (age field all ones)

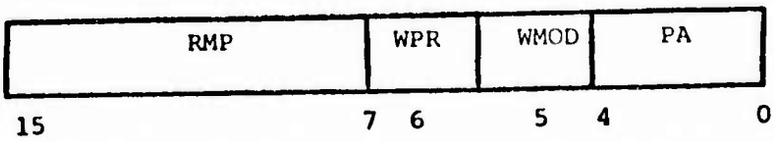
1.8.9 TIME-OUT - trap was caused by memory not responding within 5 micro secs.

1.8.10 MMDENA - The memory map device enable bit may be both read and written. When the MMDENA bit is 0, the MMD becomes completely transparent, without however clearing any of its registers. When returned to 1, operation will continue as it was before being disabled. MMDENA is reset to 0 by System Reset.

NOTE: A second input and clock line should be provided to set and reset MMDENA (to be used by possible future modifications)

II Page Table Memory (PTM)

The page table memory is a 4K by 16 bit memory which may be both read and written and is addressed between locations 740000 and 757777. Its bit assignment is given below:



2.1 RMP - Real Memory Page, The address of a 1K page in physical memory. This is sent as the nine most significant address bits to physical core if the paging function is enabled.

2.2 WPR - Write Protect, this bit when true causes a trap if any attempt to write is made.

2.3 WMOD - Write Modification, this bit when true indicates that at some time a write occurred in the page.

2.4 PA - Page Age, these 5 bits indicate the value of the age register at the last time that the page was referenced.

III Operations performed by MMD

3.1 Reading or Writing of Control Registers.

The registers detailed in Section I may be referenced at the addresses given. An attempt to write a read only register is a NO-OP. The sections of registers marked "XXX" are not only undefined but cannot be used to store and retrieve information. These registers, in general, function no differently than the registers used to command PDP-11 I/O devices.

3.2 Reading or Writing the PTM

The PTM may be used as any other memory to store and retrieve information. Standard Memory diagnostics may be run in this memory by simply setting the diagnostic to test memory between 120K to 124K. The paging mechanism must, of course, be disabled.

3.3 Initialization Sequence.

Upon power up or a system power clear signal, MMDENA is set to 0. This allows the system to operate as if the MMD were not on the system.

3.4 EXT Address Entry

The MMD will provide 6 data bit inputs for the EXT and clock or latch input. These inputs should be TTL. A set of D1P-packaged resistors will be provided for termination of these lines. The clocking of data on these lines is performed independently of any MMD function.

3.5 Paging Memory References

This function, the primary function of the MMD is described in Section IV.

IV Mapped Memory References.

Memory references are affected by most of the control registers defined in Section I and, of course, by the PTM if mapping is enabled. The series of operations used to finally arrive at a real core address is complex and for that reason has been divided into 3 sections: the calculation of the address sent to the PTM, the calculation of the Real Core Address from the output of the PTM, and other functions such as ageing which occur during memory references.

4.1 Calculation of PTM address

4.1.0 If BA <17:14> are all ones, the address is a Unibus device register. (Note: a switch or jumper should be installed to permit changing Unibus device

registers from BA <17:14> = 1111 to 0011.)

4.1.1 One of the 4 Mode Registers are selected by Bus Address 16 and 17 as follows:

BA 17	BA 16	Mode Register
0	0	764200
0	1	764202
1	0	764204
1	1	764206

NOTE: If, however, MMDENA=0, the contents of the mode registers are disabled and zero is used as the contents.

4.1.2 If EXT ENA of the selected Mode Register is true, the EXT register is concatenated with the bus address BA <15:00>. If EXTENA is false, zeros are concatenated. The result is a 22 bit address.

4.1.3 If WRD ADR of the selected Mode Register is true, the above address is intended as a word address, and is shifted left one to produce a byte address. If WRD ADR is false no shift occurs. The result is a 23 bit quantity, BBA (big bus address or bus byte address).

4.1.4 The high order 7 bits of BBA are XOR'ed with REL, the contents of the relocation field of the selected Mode Register. The result concatenated with BBA <15:00> is the RBBA (relocated big bus address). RBBA <22:11> is the PTM address.

4.2 Calculation of Real Core Address

4.2.1 The PTE bit of the selected Mode Register determines whether the page memory is used in address calculation. If PTE (Page Table) is true, the high order portion of the Real Core Address, RCA (19:11) is set to the RMP (Real Memory Page, the 9 least significant bits of PTM).

If PTE is false RCA (19:11) is set to bits <19:11> of RBBA. RBBA <22:20> are in that case ignored. Note that if PTE is false, no modifications are made to the page table, and normal page table delays are avoided.

4.2.2 RCA (10:00) are taken from BBA (10:00). This forms the 1K word, 2K byte address within a memory page.

4.3 Miscellaneous functions during Memory reference

4.3.1 Age Register

During all memory references, unless the AMD bit of

the selected Mode Register is true, or PTE is false, the contents of the age register are loaded into bits 4 through 0 of the PTM. If AMD is true, PTM (4:0) remain unchanged.

4.3.2 Write Modification.

During all write cycle-type memory references, the contents of the WMOD bit of the PTM are set to 1. During read cycles this bit is not affected. This is to be able to avoid writing pages back on to a disk if the page is unmodified.

4.3.3 Trap Status

During memory references certain information is stored in case a trap occurs. If the STS bit of the selected Mode Register is false, AST, DST, PCST, and MST are updated. If the STS bit is true, the contents of AST, DST, PCST, and MST are frozen (not updated).

4.3.4 Traps

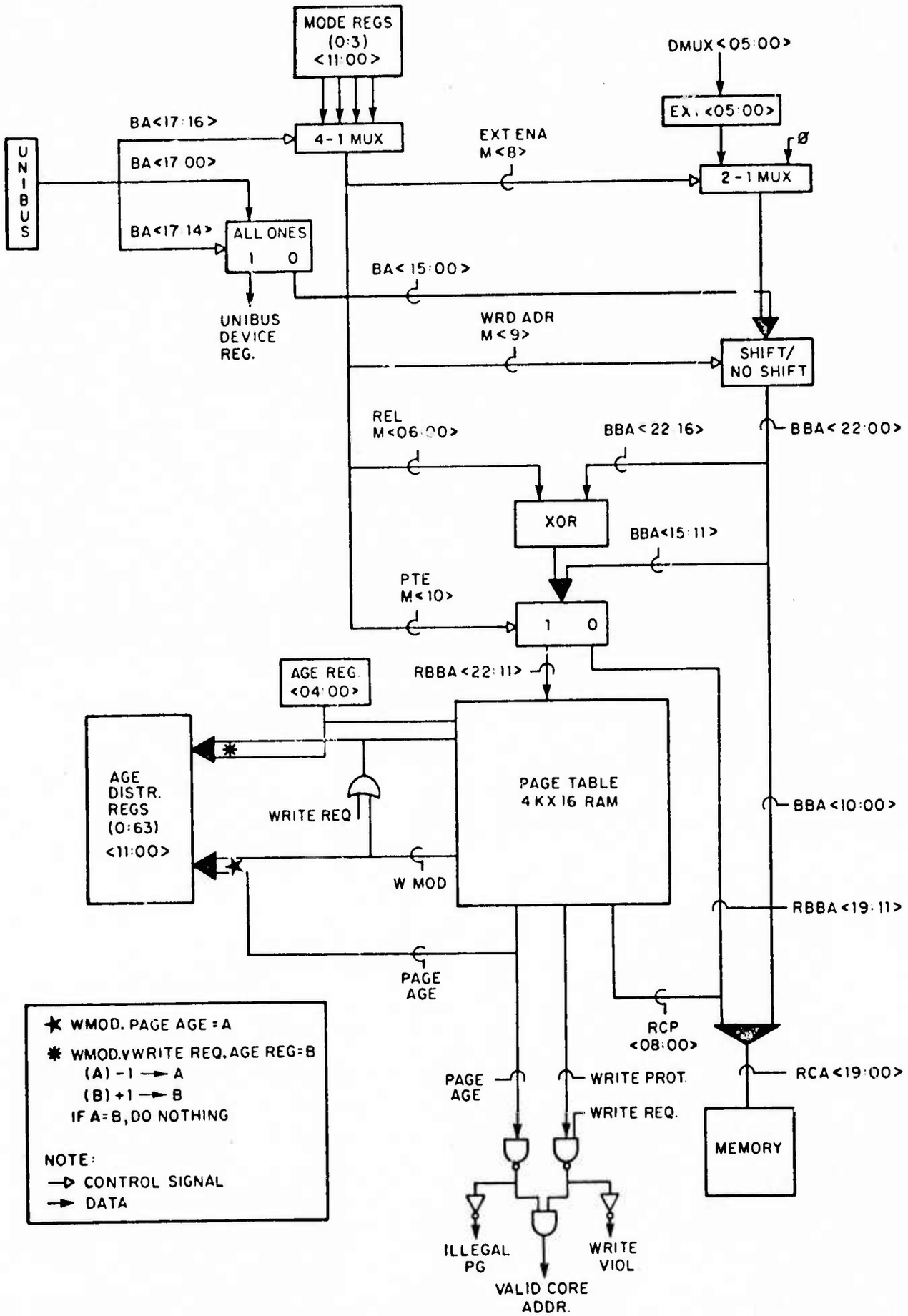
A trap occurs if (1) an attempt is made to write when the WPB of the PRM is true, or (2) the age

counter is set to all one's indicating an invalid page. A trap occurs because the Unibus signal SSYNC is not returned causing the processor to time-out. When a trap occurs, the MMDENA bit is set to 0 which protects the status information and makes the MMD transparent. Also the appropriate status bits are set (illegal page, write violation, and memory time-out), and no modification is made to the page table or age distribution registers.

4.3.5 Age Distribution Registers

These registers provide a record of how many pages there are with a particular age. This record is kept by 2 operations. When a memory reference occurs and the AMD bit is not set, the combination of the WMOD (write modification) and the PA (page age) of the PTM form an address that selects one of the ADR registers. WMOD is used as the most significant bit, bit 5; and bits 0 through 4 are taken from the PA. The contents of that location are decremented. The contents of the location specified by the present state of the write request line, or'd with WMOD, and the age register are incremented. This keeps a record of the number of pages of each age in the PTM. The age distribution

registers are also updated on direct processor writes to the page table.



MEMORY MAP BLOCK DIAGRAM

A TUTORIAL INTRODUCTION TO HERMES

Catherine L. Hausmann
Mario C. Grignetti

April 30, 1976

The Intelligent Terminal Project
BOLT BERANEK AND NEWMAN INC.
Cambridge, Massachusetts

TABLE OF CONTENTS

A TUTORIAL INTRODUCTION TO HERMES

INTRODUCTION

This guide introduces Hermes, a new system for reading and processing ARPANET messages. Hermes has features that help the user read the messages he has received, prepare messages for transmission to other users, and create and manage files of messages. This guide covers the important features that let the user accomplish these central tasks.

The material is presented in a tutorial sequence, intended to take the user in a natural progression through the various manipulations. The first four lessons cover enough to let the user accomplish basic message reading, writing and filing. The last five lessons expand on this base by introducing more sophisticated techniques in all three areas. Having progressed through the nine lessons, the user can continue to use the guide as a reference source with the aid of the detailed table of contents. The material is well laced with concrete examples. To make best use of what is presented, however, the user should also work parallel examples of his own.

Since the guide is intended to be introductory in nature, certain advanced features and techniques are omitted. For complete coverage of all Hermes details see the companion Hermes User Guide.

This guide is a product of research on advanced tutorial techniques in the BBN Intelligent Terminal Project. The main thrust of that project is to develop sophisticated, computer-based tutorial mechanisms that could be housed in a computer terminal with powerful processing capabilities of its own. The IT project has selected Hermes as a context for tutorial development because of its focus on message processing -- a general problem broadly understood by potential IT users.

This document was initially prepared to establish the framework for "live" tutorial sequences to be implemented in a computer. However, it seemed sufficiently valuable in its own right as an introduction to Hermes that we are now releasing it as a user document in the Hermes Project.

LESSON 1

Hermes is a new system for reading and processing your mail. This manual introduces a subset of Hermes that will allow you to read messages, send messages and do simple message processing. Type HERMES to enter Hermes. (Parts typed by the user are underlined. Characters like carriage return (<CR>) are shown in angle brackets.)

@hermes <CR>

```
-+ 13  329 24 Feb 76 GRIGNETTI at BBN-TENEXA Welcome
-+ 14  379 28 Feb 76 HAUSMANN at BBN-TENEXA Quotation of
-+ 15 40738 3 Mar 76 BROWN at BBN-TENEXD New Paper
>
```

Survey of messages

Upon entering Hermes an initial survey is printed out of the RECENTMESSAGES in your message file (those received since you last read your mail using Hermes). Messages read with other programs are still marked recent and are included in the initial survey because they were not read inside Hermes.

Let's look at the first line in the survey. The "-" stands for "unseen" (messages never printed out before); the "+" for "recent" (messages that came in since your last session with Hermes). This is followed by the "message

number" by which you will refer to the message. Then the character count, the date, and the author, and as much of the subject as will fit on one line.

Whenever Hermes is finished doing something for you, it prints a wedge (>) in the margin to indicate it is waiting for you to type a command.

Panic Buttons

Before telling you about Hermes commands let us make sure you understand how to correct mistakes and/or get out of sticky or unwanted situations. If you make mistakes while typing a command you can correct them with <CTRL-A>, <CTRL-W>, or <DEL>. <CTRL-A> erases the last character you typed; <CTRL-W> erases the last word; <DEL> cancels the entire line. To abort execution of a command and to return to Hermes (causing it to print the > prompt) type <CTRL-E>. To abort a long printout, type <CTRL-O>; this will cause Hermes to give you the > prompt. If you want to leave Hermes altogether, type QUIT <CR>.

Printing messages with <LF> and ^

To print the first of your recent messages, you type a linefeed (LF on your keyboard). I will do that now.

```
> <LF>
Message 13;  329 chars UNSEEN RECENT
Mail from BBN-TENEXA rcvd at 24-FEB-76 1747-EST
Date: 24 FEB 1976 1559-EST
Sender: GRIGNETTI at BBN-TENEXA
Subject: Welcome
From GRIGNETTI at BBN-TENEXA
To: Tutor
Cc: GRIGNETTI
Message-ID: <[BBN-TENEXA]24-FEB-76 15:59:44-EST.GRIGNETTI>

Welcome to your first lesson on HERMES.

Good luck!

Mario.
-------
>
```

Let's take a look at this message. At the beginning there are a few lines of "envelope" information headed by the words Date:, Sender:, Subject:, From:, To:, Cc:, and Message-ID:; these are called "fields". The Date:, Sender:, From:, To: and Message-ID: fields always appear in messages sent using Hermes. The Subject: and the Cc: fields are optional. The meaning of Subject is clear enough. The Cc: stands for Carbon Copy; people whose names appear there will receive a copy of the message.

The From: field is normally the same as the Sender: field, and Hermes sets it up that way automatically. Sometimes, however, the author of the message is not the person that sends it out (your secretary (Sender: Secretary) sends out a message for you (From: you).) In these cases the From: field can be specified by the sender.

There is another obvious field that is not preceeded explicitly by a name -- the text of the message itself is in the Text: field.

To print each subsequent message you continue typing linefeed (<LF>). To back up one message and print it again you can hit the ^ key. With these two commands you can move up and down your "message file" printing out messages in sequence. The message you just printed is called the "current message" (CMESSAGE). (There are other commands that change the cmessage that you will learn about later.) In between these one-character commands you could type other commands (that you'll learn soon), for example, to reply to the current message, file it away for future action, send a reply to someone else, etc. After doing any of these you can continue printing your messages one at a time.

Concepts covered:

Hermes
> prompt
initial survey
RECENTMESSAGES
message number
fields
  To:
  Date:
  Sender:
  Subject:
  From:
  Cc:
  Message-ID:
  Text:
<CTRL-A>
<CTRL-W>
<CTRL-O>
<CTRL-E>

<DEL>
<LF> command
^ command
CMESSAGE
message file

LESSON 2

Printing Messages with the PRINT Command

Another way of printing messages is by means of the PRINT command. You may use this command to skip anywhere in the message file and print a message. For example, to print your second message you could type PRINT followed by "2", and a <CR>:

```
> print 2 <CR>
Message 2: 298 chars
Mail from BBN-TENEXB rcvd at 19-DEC-75 1052-EST
Date: 19 DEC 1975 1019-EST
From: White
Subject: Found
To: Tutor

A sterling silver bracelet was found at the Aquarium
Thursday night near the coat rack.  Please claim it.
---Mary
-------
```

The PRINT command changes the setting of the current message (as with <LF> and ^); CMESSAGE is now set to 2. To check this, try using the ^ command and see what happens. (Remember that the ^ command prints the message preceding the current message.)

```
>^ <CR>
Message 1; 398 chars
Mail from BBN-TENEXA rcvd at 18-DEC-75 1128-EST
Date: 18 DEC 1975 1052-EST
From: GOLDMAN
Subject: A new order of widgets
To: Tutor
```

A new order of widgets has arrived and can be picked up at my office. Please only one widget to a customer.
--Frank

Abbreviating commands

You can save yourself some typing by invoking the command recognition capabilities of Hermes. After typing a few initial characters of a command, you may hit the <ESC> key. If you haven't typed enough characters to specify a command uniquely, Hermes will sound the bell to indicate that you must type more characters. Otherwise, the rest of the command will be printed out as if you had typed it entirely. In addition Hermes will print a few words (enclosed in parentheses) to cue you as to what it expects for the next part of the command. I will print message 14 now using <ESC> after PRINT to invoke command recognition.

```
>print (messages) 14 <CR>
Message 14; 379 chars UNSEEN RECENT
Mail from BBN-TENEXA rcvd at 28-FEB-76 1746-EST
Date: 28 FEB 1976 1605-EST
Sender: HAUSMANN at BBN-TENEXA
Subject: Quotation of the Week
From: HAUSMANN at BBN-TENEXA
To: Tutor
Message-ID: <[BBN-TENEXA]28-FEB-76 16:05:30-EST.HAUSMANN>

"Will there be any unforeseen complications?"

                John Seely Brown
                 26 January 1976
-------
```

Message Lists

As hinted by the cue words, you can specify a list of messages rather than a single one after PRINT and after many other Hermes commands. There are a lot of ways of specifying a message list. A message list can be a single message, like 6, or it can be an arbitrarily ordered list of messages like 7,8,3, or it can be a sequential list of messages like 3 through 8 which could be specified 3,4,5,6,7,8 or 3:8. The colon means "through". To specify a message list like 5 through the last one in the message file, type 5:last. To specify from the current message (CMESSAGE) to the last message, type .:last. The "." stands for the CMESSAGE -- it is an abbreviation.

The FILE Command

After you read a message you may wish to process it further by either filing it away or listing it at the lineprinter. Let's file CMESSAGE in a file named QUOTES.TXT. I will do this with the FILE command and command recognition.

```
>file (messages) 14 (on file) quotes.txt [Old version] <CR>
 Delete message after writing?: no
   14
```

A copy of the message is now on QUOTES.TXT and

MESSAGE.TXT. It has not been deleted from the current file.

As you saw in the initial survey, the next message (number 15) is very long. If you wanted to print it at your terminal you could use the <LF> command. Because it is so long you would probably want to abort the printout by typing <CTRL-O>.

The LIST Command

A good way to read a long message is to list it at the lineprinter. To do that you use the LIST command. I will use it now for the long message (number 15).

>list (messages) 15 <CR>

So far we have dealt mostly with manipulating mail that you received. In the next lesson we'll turn our attention to sending mail.

Concepts covered:

PRINT command
FILE command
LIST command
CMESSAGE = .
:

LESSON 3

The REPLY Command

Frequently while reading your messages one at a time, you may wish to send out a reply. An easy way to do that is by using the REPLY command. You type REPLY, followed by the number of the message you are replying to, and a <CR>. Hermes will automatically set things up for you so that all you have left to do is to type the text of your reply. When you are finished, you type a <CTRL-Z> to tell Hermes you are done with the Text: field. Before sending out the reply, Hermes will check with you so that if for any reason you are not satisfied with the message it will not be sent.

Let's reply to that long message (number 15) with the text "I'll read your paper this weekend". It is important to remember to type a carriage return after you type REPLY and the message number. (The <CR> will no longer be printed explicitly, but keep in mind that it is still there at the end of command lines.) I will avail myself of the help provided by <ESC>.

```
>reply (to message) 15
To: Tutor
Subject: (NEW PAPER)
In-Reply-To: Your message March 3, 1976
(Type text of reply, to <CTRL-Z>)
Text:
I'll read your paper this weekend. <CTRL-Z>
Format?: no
Send?: yes
Message[BBN-TENEXA]8-Apr-76 19:42:20-EST.TUTOR> sent.
```

Hermes created automatically the From:, Subject: and In-Reply-To: fields. The From: field of the message replied to becomes the To: field of the message being sent. The Subject: field is the same and the In-Reply-To: field identifies the message being replied to.

We'll see the usefulness of answering NO to "Send?" very soon.

The FORWARD Command

Sometimes you may want to send someone a copy of a message that you yourself have received. You may do so with the FORWARD command. Type FORWARD, the message number and <CR>. You will immediately be prompted for the To: field. Type the addressee and a <CR>. Then you will be asked if you want to make any comments. If you want to make comments type YES. Type your comments and end them with <CTRL-Z> as you would with the Text: field. This text will be placed at the beginning of the message. Hermes will then type "Send?" and you respond with YES or NO. I will now forward the message containing the quotation to Smith with the text beginning "For your collection".

```
>forward (messages) 14
To: Smith
Subject: A Quote
Comments (to ^Z):
For your collection. <CTRL-Z>
Format?: no
Send?: yes
```

Message <[BBN-TENEXA]8-APR-76 19:43:35-EST.TUTOR> sent.

Concepts covered:

REPLY command
FORWARD command

LESSON 4

The COMPOSE command

A more general way to send out messages is to use the COMPOSE command. COMPOSE works pretty much like SNDMSG, prompting you with the fields you need to send an ordinary message, i.e., the To:, Cc:, Subject: and Text: fields. If you don't want to send carbon copies to anyone, just type a <CR> after the Cc: prompt. Similarly if you don't wish to specify a subject just type a <CR> after the Subject: field. If you want to send the message (or copies of it) to more than one person, just type their names separated by commas, and terminate with a <CR>. You can correct your typing mistakes using <CTRL-A>, <CTRL-W> and <DEL>, the same as with Hermes commands. In addition, there are a few more facilities to help you compose your message.

If a line has gotten messy because of too many mistakes and corrections, you can see it cleaned up by typing <CTRL-R> (R for Retype). Typing <CTRL-S> will show the entire field you are typing.

To signal Hermes that you are finished typing the Text: field, type a <CTRL-Z>. Hermes will then ask you if it is OK to send the message, and you should answer YES. (Answering NO allows you to go back and change the message as described below).

I will now COMPOSE a message to Green:

```
>compose
Prompted message composition:
To:  Green
Cc:
Subject:  Hermes
Text:
Hermes is great fun to use. <CTRL-Z>
Format?: n
Message <[BBN-TENEXA]8-Apr-76 19:44:46-EST.TUTOR> sent.
```

Drafting a Message

Occasionally you will realize you made a mistake when it is too late or too inconvenient to fix it in the above described manner. It is here that answering NO to Send? comes handy. If you answer NO, Hermes will put you in a "draft state" and will indicate this by printing two wedges in the margin instead of one. Once in the draft state, you can pick out individual fields and work on them independently of the others. For example, typing To: followed by a name will add that name to the end of the To: field in your draft. You can add a new field to your message in the same way, for example, an Fcc: field. Fcc: stands for "File Carbon Copy". It provides a good way of taking a message that you have created and storing it into a file for later reference.

The SHOW Command

To see the whole contents of a field you can use the

SHOW command. To use it simply type SHOW followed by the name of the field you want to see. To see all fields, type SHOW ALL.

Editing

Most commonly, the mistakes you make will not be simple omissions at the end of a field, but will require more complex editing operations in the middle of, say, the Text: field. At this point you may call any of three editors -- TECO, NETED, or XED. To edit a field, you type the name of your favorite editor followed by the name of the field. If you omit the latter, Text: is assumed. For example, to edit Text: using TECO, simply type TECO and a <CR>. After you are done with the changes you want, type ;H<ESC> and you'll return to the DRAFT state. (Do not use EX<ESC> or ;U<ESC>.) After you are done and you are satisfied with the message, you type SEND. This will send the message. Then you type DONE to return to command level.

Let's create a message to Burton using the COMPOSE command. I will answer NO to "Send?" in order to enter the Draft state. Then I will add an Fcc: field -- filing a copy of the message in the file EDITORS.TXT.

```
>compose
Prompted message composition:
To: Burton
Cc: White
Subject: Editors
```

```
Text:
I prefer to use TECO when I edit text. <CTRL-Z>
Format?: no
Send? no
>>Fcc: (filenames) editors.txt [New file]
>>send [Confirm]
Message <[BBN-TENEXA]8-Apr-76 19:47:30-EST.TUTOR> sent.
```

Concepts covered:

COMPOSE command
SHOW command
<CTRL-R>
<CTRL-S>
draft state
Fcc: field
TECO, NETED, XED
;h

LESSON 5

Selective Printing -- Filters

The SURVEY Command

Remember that th initial survey we got when we entered Hermes showed only RECENTMESSAGES. To see a survey of the whole message file use the SURVEY command.

```
>survey
    1   398 18 Dec 75 GOLDMAN at BBN-TENEXA New Widgets
    2   298 19 Dec 75 WHITE at BBN-TENEXA Found
    3   303 25 Dec 75 AIGHES at BBN-TENEXA Letter from the
    4   111  2 Jan 76 MORTON at BBN-TENEXD Typewriter needed
    5   142 18 Jan 76 AIGHES at BBN-TENEXA Reminder
    6   546 20 Jan 76 MCCARTHY at ISID Using 2-Color Ribbons
    7   918 22 Jan 76 LUCA at BBN-TENEXA Typewriter Received
    8   489 28 Jan 76 SUSSMAN at BBN-TENEXA IEEE  Speakers
    9   570 29 Jan 76 SMITH at ISID A poll on voting choices
   10   237 31 Jan 76 SUSSMAN at BBN-TENEXA IEEE tickets
   11   287 31 Jan 76 SUSSMAN at BBN-TENEXA IEEE proceedings
   12   327  5 Feb 76 SCOT at BBN-TENEXD Quote from Abe
   13   329 24 Feb 76 GRIGNETTI at BBN-TENEXA Welcome
   14   379 28 Feb 76 HAUSMANN at BBN-TENEXA Quotation of the
   15 40738 3 Mar 76 BROWN at ISID New paper
```

Filters

Rather than taking time to survey your whole message file, you may wish to see only a certain group of messages, for example those that are from a certain person or delivered after a certain date. "Filters" are used to cut down the number of messages that you are working with. You may filter according to Sender, the date before, on, or after which they were sent, whether they are seen, unseen, deleted, or undeleted, etc.

To survey all messages from Sussman type:

SURVEY FROM: SUSSMAN

Here's what results when the above line is typed to Hermes.

```
>survey (messages) from: Sussman
     8  489 28 Jan 76 SUSSMAN IEEE  speakers
    10  237 30 Jan 76 SUSSMAN IEEE  tickets
    11  287 31 Jan 76 SUSSMAN IEEE proceedings
```

The ? Facility

At any place in any Hermes command you may type "?" to see the various options of what you may type next. This is a good way to become more familiar with Hermes' capabilities. I will now do the same survey again, but this time I'll type a ? to see all the various sequences and filters that could be used.

```
>survey (messages) ?
msg
LASTMESSAGE
CMESSAGE
ALL
PREVIOUSSEQUENCE
CSEQUENCE
Date:
From:
To:
Cc:
Bcc:
In-Reply-To:
Reference:
Keywords:
Precedence:
Message-Class:
Special-Handling:
Message-ID:
```

or SURVEY that prompt you for (messages), you can type a single message, a "." standing for CMESSAGE, a list of messages or a group of messages further specified by a filter or multiple filters.

Concepts covered:

SURVEY command
Filters
? Facility

LESSON 6

The CREATE Command

Rather than composing messages directly, using the COMPOSE command, you may wish to proceed in a more leisurely way by working first on a draft of the message you intend to send. You begin creating a draft by typing CREATE <CR>. Hermes will print two wedges. You put together the draft by typing out the fields of your choice, one at a time, with the appropriate text for each field.

Let us now go into more detail on what an addressee list looks like -- the contents of the To:, Cc:, and Bcc: fields. (Bcc: stands for "Blind Carbon Copy".) You recall that an address list may be a single name or a list of names separated by commas, like:

BROWN,SMITH,MURPHY

Each single name can also have a site associated with it, like:

BROWN@BBNA

If you find that your address list is longer than one line, end the line with a comma and you may continue the list on the next line.

Sometimes you may type in the names of people you want to send messages to only to find that you have left out one name. No need to type the whole list over again. You can add to any field by typing the field name again and what you

want to add. This will be appended to the original field.

The SAVE-FIELD Command

If you often find yourself typing the same list of addressees each time you send a message, you may wish to keep around the names on a file. Then when you get ready to type in the list of names, you may instead insert the contents of the file. To save any field on a file, use the SAVE-FIELD command:

```
>>to: Brown,Bell,Smith
>>save-field (field) to: (on File) computerpeople [New file]
```

The ADD-FILE Command

To retrieve the file just created and add it to a field, you would use the ADD-FILE Command. It will ask you for the file name and what field you want to add it to. You may often want to use the ADD-FILE command to insert the contents of a file, say a progress report or a requisition order, into the Text: field. Let's say you have a report you want to send out which you want to precede with the heading "Please send your comments". To accomplish this, enter the heading and then use ADD-FILE to bring in the main text.

```
>create
>>text: (characters to ^Z)
Please send your comments. <CTRL-Z>
```

```
>>add-file (filename) report.txt
>>
```

I could have appended more text at the end of the Text: field by typing Text: again and the material.

After you type a field, say the Text: field, you may wish to see if you have typed what you wanted. Recall that you may see a field by typing SHOW followed by the field name. To see all fields type SHOW ALL.

After seeing the Text: field you may wish to make some changes. Remember that you may call any of three editors -- TECO, NETED, or XED. To enter TECO, type TECO. Make the changes using the available TECO commands, then return to the draft state using ;H.

```
>>create
>>to: (Addressees) Hausmann
>>text: (characters to ^Z)
New phone numbers. <CTRL-Z>
Format?: n
>>add-file (filename) telephone.;1 (to part) text:
>>subject: A memo to the staff
>>show all
To: HAUSMANN
Subject: A memo to the staff

New phone numbers

New phone numbers for the office staff.
See the directory in the lirary.
>>teco: (field) text:

79 CHARS
*rrary$brary$
*;h$
Replace the Text: field?: yes
>>format
>>send [Confirm]
```

Message<[BBN-TENEXA]23-Apr-76 14:29:51-EST.TUTOR> sent.

Concepts covered:

CREATE command
SHOW command
SAVE-FIELD command
ADD-FILE Command

LESSON 7

Housekeeping and Your Message File

Once in awhile you should go through your message file and "clean it up". That is, look at the messages, decide what to do with them; either delete them, leave them, or file them away in another file. You may wish to do this every time you get new mail in Hermes. Or you may wait until your file gets unwieldy. Whichever way you choose, the method is similar.

You should begin by surveying your message file. If it is quite long, you can list the survey on the lineprinter by typing "SURVEY ALL LPT:".

>survey (messages) all (on file) lpt: [Confirm]

The DELETE Command

Let's suppose we have decided to delete all messages from Aighes. Remember filters may be used with any command that is expecting a message list. They can be put to good use here with the DELETE command:

>delete (messages) from aighes
3,5

(Note that the DELETE command only marks these messages for

deletion; it doesn't physically remove them until you leave Hermes.

I want to file the oldest messages (those received before 1-1-76) into a file called OLDMSG.TXT. I can accomplish this by the FILE command with a filter.

```
>file (messages) before 1-1-76 (on file) oldmsg.txt [Old
 version]
Delete message after writing?: y
Message 3 is marked deleted: Can't be filed.
  1,2
```

A lineprinter listing of messages whose Subject: contains IEEE will be sufficient for my records:

```
>list (messages) subject: IEEE
>delete (messages) subject: IEEE
8,10,11
```

Now to do a survey of the whole message file and see if there are any unseen messages:

```
>survey
Message 1 is marked deleted.
Message 2 is marked deleted.
Message 3 is marked deleted.
    4  111  2 Jan 76 MORTON at BBN-TENEXD Typewriter Tables
Message 5 is marked deleted.
    6  546 20 Jan 76 MCCARTHY at ISID Using 2-color ribbons
    7  918 16 Jan 76 LUCA at BBN-TENEXA Typewriter received
Message 8 is marked deleted.
    9  570 29 Jan 76 SMITH at ISID A poll on voting choices
Message 10 is marked deleted.
Message 11 is marked deleted.
   12  327  5 Feb 76 SCOT at BBN-TENEXD Quote from Abe
   13  329 24 Feb 76 GRIGNETTI at BBN-TENEXA Welcome
   14  379 28 Feb 76 HAUSMANN at BBN-TENEXA Quotation of the
```

-+ 15 40738 3 Mar 76 BROWN at 1SID New Paper

Yes, that long message has not been seen. If we wanted to look at the first few lines we could PRINT it, and use <CTRL-O> to abort the printout when we've seen enough.

Concepts covered:

DELETE command

LESSON 8

Current Objects

CSEQUENCE

In the preceding examples we have used filters to cut down on the number of messages that we are looking at. But this use of filters only applied to the message sequence for one command. Suppose you wish to refer to a particular message sequence over and over again. To do so you could make use of an object called the CSEQUENCE which stands for Current sequence.

The CONSIDER Command

When you first get into Hermes CSEQUENCE is set to ALLMESSAGES. You can reset it with the CONSIDER command. For example, suppose you want to create a CSEQUENCE that contains all message that have anything to do with quotations. You would do this by typing:

CONSIDER SUBJECT: QUOT

Note that you don't have to type the whole word that you are searching for. Now I will set CSEQUENCE to all messages with Subject: typewriter:

>consider (messages) subject: typewriter

Now watch while I type SURVEY followed by <ESC>, <ESC>.

```
>survey (messages) CSEQUENCE
   4  111  2 Jan 76 MORTON at BBN-TENEXD Typewriter needed
   7  918 22 Jan 76 LUCA at BBN-TENEXA Typewriter received
```

You can see that the SURVEY command defaults to CSEQUENCE. Many commands do.

I forgot that message 6 should be added to CSEQUENCE. It also has something to do with typewriters. I don't need to type the CONSIDER command all over again. Instead I can use the CSEQUENCE editor. To add messages to the CSEQUENCE, type ADD and the message list. To leave the editor type DONE.

```
>edit (object) csequence
>>add (messages) 6
>>show
4,7,6
>>done
>
```

Notice that when we typed SHOW, the messages were no longer arranged in ascending order by date. This is because the sequence was not put together one message at a time in order, but instead it was put together by groups and single messages all out of order. Suppose now we want to reorder the messages by date. Again we enter the CSEQUENCE editor. We will sort the messages contained by date by typing SORT followed by DATE; then DONE to leave the editor.

```
>edit (object) csequence
>>sort (by) date
>>show
4,6,7
>>done
```

Now that we have a new CSEQUENCE about typewriters what can we do with it? We can move through it with the <LF> and ^ commands handling each message one at a time. Or we can file CSEQUENCE in a file, let's say called EQUIPMENT.TXT for later use:

```
>file (messages) csequence (on file) equipment.txt [New
 file]
Delete message after writing?: y
  4,6,7
```

Now we will introduce a new concept, "CMESSAGE-FILE".

CMESSAGE-FILE; the GET Command

When you first enter Hermes, the file that automatically gets read in is named MESSAGE.TXT. This becomes what is termed the "CMESSAGE-FILE" (standing for "Current" message file). I have just created a message file called EQUIPMENT.TXT. Let's focus our attention on it now. In other words let's make it the CMESSAGE-FILE. This is done with the GET command.

```
>get (message-file) equipment.txt;1
Expunge and re-number the message-file?: y
```

MESSAGE.TXT is no longer the CMESSAGE-FILE. EQUIPMENT.TXT has taken its place. All Hermes commands you perform now will be done on that file. You can make any message file that you created with the FILE command into the CMESSAGE-FILE. This gives you a way of keeping down the size of the MESSAGE.TXT file by filing messages away into various small message files organized by subject, or author, etc. You may then access them with the GET command when you want to work with their contents.

Concepts covered:

CSEQUENCE
CONSIDER command
GET command

LESSON 9

On-line User Aids

The HELP, EXPLAIN, and DESCRIBE Commands

Before leaving you now, there are three commands for getting information about Hermes that I would like to tell you about - HELP, EXPLAIN, and DESCRIBE. The first, HELP, gives a brief introduction of the on-line aids available in Hermes. You simply type HELP. It proceeds as you wish it to by asking if you wish to see more information about specific topics, bypassing topics that you aren't currently interested in.

The EXPLAIN command permits you to explore a set of brief descriptions of the main concepts in Hermes. These descriptions are arranged in a tree-like structure with nodes higher in the tree being more general, and nodes lower being more detailed. To use this command, type EXPLAIN followed by the topic you wish to know about.

The third user aid command, DESCRIBE, provides reference information concerning various aspects of Hermes. The command is available at the top level (when prompted by a single ">"), and as a subcommand from within EXPLAIN. DESCRIBE provides a paragraph or two of reference information concerning the topic you give it. To use it, type DESCRIBE followed by the topic.

With both EXPLAIN and DESCRIBE there is a "?" feature to help you narrow down the topic you are looking for. For example, I'm looking for a category having something to do with the topic Profile. I may type EXPLAIN PROFILE and then type a "?" to see all the topics that begin with those letters. I will do that now, and then choose one of the topics:

```
>explain (topic) profile?
PROFILE
PROFILE-CONTENTS
>Explain (topic) profile-contents
PROFILE-CONTENTS
The profile contains three sort of information:
     a) Switch settings: there are certain aspects of HERMES
which are tailorable to behave in one of a few (often 3)
different ways.  Each switch controls the behavior of one of
these aspects.
     b) Templates: these are you choices for defaults, and
you own personal templates.
     c) Filters: these are your choices for defaults, and
you own personal filters.
1.  For something on switches, type DESCRIBE SWITCHES.
2.  For something on templates, type DESCRIBE TEMPLATES.
3.  Filters
>>
```

At this point I may choose one of these three things to do, or I may use DESCRIBE or EXPLAIN again for a different topic. Type DONE at the >> prompt to leave the EXPLAIN command.

You have read about many of the important features of Hermes. These include reading messages, preparing messages for transmission, and creating and managing files of messages. Hermes is a powerful and large system and only a

subset has been covered in this introductory document. You can expand your knowledge about Hermes by making further use of HELP, EXPLAIN, DESCRIBE, the ? feature, and <ESC>.

Concepts covered:

HELP command
EXPLAIN command
DESCRIBE command