User Contributed Software Supplemental Manual

August, 1983

Computer Systems Research Group
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

User Contributed Software

The following is a list of acknowledgements to the people and organizations who contributed software for 4.2BSD.

APL

Authors: John D. Bruner
Lawrence Livermore Laboratory
P.O. Box 808, L-276
Livermore, CA 94550
(415) 422-0758

Prof. Anthony P. Reeves
Cornell University, Phillips Hall
Ithaca, NY 14853
(607) 256-4296

This implementation of APL is a descendent of a program originally written by Ken Thompson at Bell Labs sometime before UNIX† version 6. It went to Yale for a while and then came to Purdue (Electrical Engineering) via a Chicago distribution in 1976. It was extensively modified at Purdue by Anthony P. Reeves, Jim Besemer, and John Bruner. The editor "apled" which APL uses to edit functions was developed from the V6 "ed" editor by Craig Strickland.

Improvements which were added at Purdue/EE include quad I/O functions, a state indicator of sorts, statement labels, and a number of primitive functions. The current version runs on both large V7 PDP-11's (those which are capable of running separated I/D programs) and VAXes. (Workspaces are not directly interchangeable but can be converted).

Bib

Authors: Timothy A. Budd
Gary M. Levin

Address: Department of Computer Science
University of Arizona
Tucson, Arizona 85721
(602) 621-6613

Bib is a program for collecting and formatting reference lists in documents. It is a preprocessor to the nroff/troff typesetting systems, much like the tbl [.tbl.] and eqn [.eqn.] systems. Bib takes two inputs: a document to be formatted and a library of references. Imprecise citations in the source document are replaced by more conventional citation strings, the appropriate references are

† UNIX is a trademark of Bell Laboratories.

selected from the reference file, and commands are generated to format both citation and the referenced item in the bibliography.

Courier

Author:	Eric C. Cooper
Address:	Computer Science Division, EECS University of California Berkeley, CA 94720
Net address:	cooper@berkeley (ARPA) ucbvax!cooper (UUCP)

This is the Courier remote procedure call protocol for Berkeley UNIX (version 4.2). Courier is both a protocol and a specification language. The protocol specifies how remote procedures are invoked and how parameters and results of various data types are transmitted. The specification language, somewhat reminiscent of Mesa, provides a simple way of defining the remote interfaces of distributed programs.

CPM

Author:	Helge Skrivervik bergensveien 8, N - Oslo 9, Norway
Net address:	helge@berkeley helge@nta-vax (norway)

Cpm lets UNIX users read and write standard cp/m 8" floppy disks and provides a cp/m like user interface for manipulating cp/m files.

Dsh (distributed shell)

Author:	Dave Presotto
Address:	University of California Computer Science Division, EECS Berkeley, CA 94720.
Net address:	presotto@berkeley (ARPA) ucbvax!presotto (UUCP)

Dsh works in two phases; bidding and execution. The bidding is performed by starting a dbid program on all the requested machines (using rsh). The dbid's send back bids using the ipc. The dsh command then picks some subset of the bidding machines and runs the requested command on them, once again using rsh.

Hyperchannel driver

Author: Steve Glaser

Address: Tektronix M/S 61-215
P.O. Box 1000
Wilsonville, OR 97070
(503) 685-2562

Net Address: steveg.tektronix@rand-relay (ARPA)
teklabs!steveg

This is release 2.1 of the hyperchannel driver and related programs.

It is being actively used on one machine at Tek (soon to bring up a few more machines). It has been used successfully to communicate with a CDC Cyber, vax/unix under unet, and PDP-11's under unix/unet. It is working on a 11/780. It has not been tested on a 11/750 or 11/730 although there should not be any difficulties.

✓ ICON

Author: Ralph E. Griswold
Department of Computer Science
The University of Arizona
Tucson, AZ 85721

Net address: icon-project.arizona@rand-relay

Icon is a high-level, general-purpose programming language that is well-suited for nonnumerical applications. Icon supports several data types, including variable-length strings, lists with flexible access methods, and tables with associative lookup. Storage management is automatic. Icon has a goal-directed evaluation mechanism that allows concise solutions of many programming tasks to be formulated easily. Its string scanning facility is comparable to pattern matching in SNOBOL4, but Icon allows all language operations to be used in the analysis and synthesis of strings. With its high-level facilities and emphasis on string and list processing, Icon fills a gap in the UNIX language hierarchy.

The design and implementation of Icon was supported, in part, by the National Science Foundation under grants MCS75-01397, MCS79-03890, and MCS81-01916.

Learn scripts for the vi editor

Author: Pavel Curtis

Net address: {decvax,vax135,allegra,harpo,...}!cornell!pavel
Pavel.Cornell@Udel-Relay

✓ MH mail system

Authors:	Bruce Borden Stockton Gaines Norman Shapiro
Help from:	Phyllis Kantar Robert Anderson David Crocker

The user command interface to MH is the UNIX "shell" (the standard UNIX command interpreter). Each separable component of message handling, such as message composition or message display, is a separate command. Each program is driven from and updates a private user environment, which is stored as a file between program invocations. This private environment also contains information to "custom tailor" MH to the individual's tastes. MH stores each message as a separate file under UNIX and it utilizes the tree-structured UNIX file system to organize groups of files within separate directories or "folders." All of the UNIX facilities for dealing with files and directories, such as renaming, copying, deleting, cataloging, off-line printing, etc., are applicable to messages and directories of messages (folders). Thus, important capabilities needed in a message system are available in MH without the need (often seen in other message systems) for code that duplicates the facilities of the supporting operating system. It also allows users familiar with the shell to use MH with minimal effort.

Notesfiles

Authors: Ray Essick

Department of Computer Science
222 Digital Computer Laboratory
University of Illinois at Urbana-Champaign
1304 West Springfield Ave.
Urbana, IL 61801
uiucdcs!essick (UUCP)
essick.uiuc@rand-relay (ARPA)

Rob Kolstad

Parsec Scientific Computer Corporation
Richardson, TX
parsec!kolstad (UUCP)

Notesfiles support computer managed discussion forums. Discussions can have many different purposes and scopes: the notesfile system has been designed to be flexible enough to handle differing requirements.

Each notesfile discusses a single topic. The depth of discussion within a notesfile is ideally held constant.. While some users may require a general discussion of personal workstations, a different group may desire detailed discussions about the I/O bus structure of the WICAT 68000 (a particular workstation). These discussions might well be separated into two different notesfiles.

Each notesfile contains a list of logically independent notes (called base notes). A note is a block of text with a comment or question intended to be seen by members of the notesfile community. The note display shows the text, its creation time, its title, the notesfile's title, the author's name (some notesfiles allow anonymous notes), the number of "responses", and optionally a "director message". Each base note can have a number of "responses": replies, retorts, further comments, criticism, or related questions concerning the base note. Thus, a notesfile contains an ordered list of ordered lists. This arrangement has historically been more convenient than other proposals (e.g., trees were studied on the PLATO (trademark of CDC) system).

The concept of a notesfile was originally implemented at the University of Illinois, Urbana-Champaign, on the PLATO system (trademark of Control Data Corporation). The UNIX notesfile system includes these ideas with adaptations and enhancements made possible by the UNIX environment.

The UNIX notesfile system provides users with the abilities to read notes and responses, write notes and responses, forward note text to other users (via mail) or other notesfiles, save note text in their own files, and sequence through a set of notesfiles seeing just new text. Each notesfile has a set of "directors" who manage the notesfile: they delete old notes, compress the file when needed, grant and restrict access to the notesfile, and set different notesfile parameters (e.g., title, "director message", policy note, whether notes' authors can be anonymous). Some notesfiles contain correspondence from other computers. Like the UNIX "USENET", notes and responses are exchanged (often over phone lines) with remote machines. The notesfile system provides automatic exchange and updating of notes in an arbitrarily connected network.

RCS Revision Control System

Author:	Walter F. Tichy
Address:	Department of Computer Sciences Purdue University West Lafayette, IN 47907
Net address:	wft@purdue(ARPA)

The Revision Control System (RCS) manages multiple revisions of text files. RCS automates the storing, retrieval, logging, identification, and merging of revisions. RCS is useful for text that is revised frequently, for example programs, documentation, graphics, papers, form letters, etc.

Sccstorcs

Author:	Kenneth L. Greer 2065 Alma St. Palo Alto, CA 94301
Net address:	kg.HP-Labs@Rand-Relay

Sccstorcs builds an RCS Revision Control System file from an SCCS Source Code Control System file. The deltas and comments for each delta are preserved and installed into the new RCS file in order. Also preserved are the user access list and descriptive text, if any, from the SCCS file.

SPMS - Software Project Management System

Author: Peter J. Nicklin

Address: University of California
Computer Systems Research Group
Computer Science Division, EECS
Berkeley, California 94720.

Net address: nicklin@berkeley (ARPA)
ucbvax!nicklin (UUCP)

The Software Project Management System (SPMS) is a system for the management of medium- to large-scale software systems. SPMS provides, within the UNIX environment, a number of commands which can greatly simplify many tasks associated with program development and maintenance. SPMS does not attempt to duplicate existing UNIX program development tools such as *make* or *SCCS*, but instead provides a way of coordinating these tools.

SPMS can be fitted to existing software systems. It retains the full capabilities of the UNIX environment with unrestricted access to UNIX tools. As a result, software packages developed using SPMS do not depend on the system for their survival and can be ported to versions of UNIX that do not support SPMS.

The design and implementation of SPMS was supported, in part, by DARPA under grant (DARPA) N00039-C-0235.

USENET ("readnews") Version B Software

Author: Matt Glickman

Address: Computer Systems Research Group,
Computer Science Division, EECS
University of California,
Berkeley, California 94720

Net address: glickman@berkeley(ARPA)
ucbvax!glickman(UUCP)

The USENET is a network of UNIX machines that exchange news articles on a frightening variety of topics known as newsgroups. To join the network, the user must find a machine already on the network willing to forward articles. News can be exchanged over UUCP, Arpanet, ethernet, or practically any network that allows file transfer and/or mail. The two main programs are *readnews*, the article reading program and *inews*, the article insertion program.

Miscellaneous tools

Author:	John Kunze
Address:	P.O. Box 4086 Berkeley, CA 94704
Net address:	jak%monet@berkeley (ARPA)

The programs here are intended to be general programmer's tools.

jot - print sequential or random data
lam - laminate files
rs - reshape a data array

Enhancements are planned for the rs command.

An Overview of the Icon Programming Language*

Ralph E. Griswold

Department of Computer Science
The University of Arizona, Tucson, AZ 85721

February 27, 1983

1. Introduction

Icon is a high-level programming language with extensive facilities for processing strings and lists. Icon has several novel features, including expressions that may produce sequences of results, goal-directed evaluation that automatically searches for a successful result, and string scanning that allows operations on strings to be formulated at a high conceptual level.

Icon resembles SNOBOL4 [1] in its emphasis on high-level string processing and a design philosophy that allows ease of programming and short, concise programs. Like SNOBOL4, storage allocation and garbage collection are automatic in Icon, and there are few restrictions on the sizes of objects. Strings, lists, and other structures are created during program execution and their size does not need to be known when a program is written. Values are converted to expected types automatically; for example, numeral strings read in as input can be used in numerical computations without explicit conversion. Whereas SNOBOL4 has a pattern-matching facility that is separate from the rest of the language, string scanning is integrated with the rest of the language facilities in Icon. Unlike SNOBOL4, Icon has an expression-based syntax with reserved words; in appearance, Icon programs resemble those of several other conventional programming languages.

Examples of the kinds of problems for which Icon is well suited are:

- text analysis, editing, and reformatting
- document preparation
- symbolic mathematics
- text generation
- program parsing and translation
- data laundry
- graph manipulation

Icon is implemented in C [2] and runs under UNIX† on the PDP-11, VAX-11, and Onyx C8002 computers. Implementations for other computers and operating systems are presently underway. An earlier version of Icon [3] is available on several large-scale computers, including the CRAY-1, DEC-10, IBM 360/370, PRIME 450/550/650, DG MV8000, and CDC Cyber/6000.

*This work was supported by the National Science Foundation under Grant MCS81-01916.
†UNIX is a trademark of Bell Laboratories.

A brief description of some of the representative features of Icon is given in the following sections. This description is not rigorous and does not include many features of Icon. See [4] for a complete description.

2. Strings

Strings of characters may be arbitrarily long, limited only by the architecture of the computer on which Icon is implemented. A string may be specified literally by enclosing it in double quotation marks, as in

```
greeting := "Hello world"
```

which assigns an 11-character string to **greeting**, and

```
address := ""
```

which assigns the zero-length *empty* string to **address**. The number of characters in a string **s**, its size, is given by ***s**. For example, ***greeting** is 11 and ***address** is 0.

Icon uses the ASCII character set, extended to 256 characters. There are escape conventions, similar to those of C, for representing characters that cannot be keyboarded.

Strings also can be read in and written out, as in

```
line := read()
```

and

```
write(line)
```

Strings can be constructed by concatenation, as in

```
element := "(" || read() || ")"
```

If the concatenation of a number of strings is to be written out, the **write** function can be used with several arguments to avoid actual concatenation:

```
write("(",read(),")")
```

Substrings can be formed by subscripting strings with range specifications that indicate, by position, the desired range of characters. For example,

```
middle := line[10:20]
```

assigns to **middle** the string of characters of **line** between positions 10 and 20. Similarly,

```
write(line[2])
```

writes the second character of **line**. The value 0 is used to refer to the position after the last character of a string. Thus

```
write(line[2:0])
```

writes the substring of **line** from the second character to the end, thus omitting the first character.

An assignment can be made to the substring of string-valued variable to change its value. For example,

```
line[2] := "..."
```

replaces the second character of **line** by three dots. Note that the size of **line** changes automatically.

There are many functions for analyzing strings. An example is

```
find(s1, s2)
```

which produces the position in **s2** at which **s1** occurs as a substring. For example, if the value of **greeting** is as given earlier,

```
find("or", greeting)
```

produces the value 8. See Section 4.2 for the handling of situations in which **s1** does not occur in **s2**, or in which it occurs at several different positions.

3. Character Sets

While strings are sequences of characters, *csets* are sets of characters in which membership rather than order is significant. Csets are represented literally using single enclosing quotation marks, as in

```
vowels := 'aeiouAEIOU'
```

Two useful built-in csets are **&lcase** and **&ucase**, which consist of the lowercase and uppercase letters, respectively. Set operations are provided for csets. For example,

```
letters := &lcase ++ &ucase
```

forms the cset union of the lowercase and uppercase letters and assigns the resulting cset to **letters**, while

```
consonants := letters -- 'aeiouAEIOU'
```

forms the cset difference of the letters and the vowels and assigns the resulting cset to **consonants**.

Csets are useful in situations in which any one of a number of characters is significant. An example is the string analysis function

```
upto(c, s)
```

which produces the position **s** at which any character in **c** occurs. For example,

```
upto(vowels, greeting)
```

produces 2. Another string analysis function that uses csets is

```
many(c, s)
```

which produces the position in **s** after an initial substring consisting only of characters that occur in **s**. An example of the use of **many** is in locating words. Suppose, for example, that a word is defined to consist of a string of letters. The expression

```
write(line[1:many(letters, line)])
```

writes a word at the beginning of **line**. Note the use of the position returned by a string analysis function to specify the end of a substring.

4. Expression Evaluation

4.1. Conditional Expressions

In Icon there are *conditional expressions* that may *succeed* and produce a result, or may *fail* and not produce any result. An example is the comparison operation

```
i > j
```

which succeeds (and produces the value of **j**) provided that the value of **i** is greater than the value of **j**, but fails otherwise.

The success or failure of conditional operations is used instead of Boolean values to drive control structures in Icon. An example is

```
if i > j then k := i else k := j
```

which assigns the value of **i** to **k** if the value of **i** is greater than the value of **j**, but assigns the value of **j** to **k** otherwise.

The usefulness of the concepts of success and failure is illustrated by **find(s1, s2)**, which fails if **s1** does not occur as a substring of **s2**. Thus

```
if i := find("or",line) then write(i)
```

writes the position at which **or** occurs in **line**, if it occurs, but does not write a value if it does not occur.

Many expressions in Icon are conditional. An example is **read()**, which produces the next line from the input file, but fails when the end of the file is reached. The following expression is typical of programming in Icon and illustrates the integration of conditional expressions and conventional control structures:

```
while line := read() do
   write(line)
```

This expression copies the input file to the output file.

If an argument of a function fails, the function is not called, and the function call fails as well. This "inheritance" of failure allows the concise formulation of many programming tasks. Omitting the optional **do** clause in **while-do**, the previous expression can be rewritten as

```
while write(read())
```

4.2. Generators

In some situations, an expression may be capable of producing more than one result. Consider

```
sentence := "Store it in the neighboring harbor"
find("or", sentence)
```

Here **or** occurs in **sentence** at positions 3, 23, and 33. Most programming languages treat this situation by selecting one of the positions, such as the first, as the result of the expression. In Icon, such an expression is a *generator* and is capable of producing all three positions.

The results that a generator produces depend on context. In a situation where only one result is needed, the first is produced, as in

```
i := find("or", sentence)
```

which assigns the value 3 to **i**.

If the result produced by a generator does not lead to the success of an enclosing expression, however, the generator is *resumed* to produce another value. An example is

```
if (i := find("or", sentence)) > 5 then write(i)
```

Here the first result produced by the generator, 3, is assigned to **i**, but this value is not greater than 5 and the comparison operation fails. At this point, the generator is resumed and produces the second position, 23, which is greater than 5.

The comparison operation then succeeds and the value 23 is written. Because of the inheritance of failure and the fact that comparison operations return the value of their right argument, this expression can be written in the following more compact form:

```
write(5 < find("or", sentence))
```

Goal-directed evaluation is inherent in the expression evaluation mechanism of Icon and can be used in arbitrarily complicated situations. For example,

```
find("or", sentence1) = find("and", sentence2)
```

succeeds if **or** occurs in **sentence1** at the same position as **and** occurs in **sentence2**.

A generator can be resumed repeatedly to produce all its results by using the **every-do** control structure. An example is

```
every i := find("or", sentence)
   do write(i)
```

which writes all the positions at which **or** occurs in **sentence**. For the example above, these are 3, 23, and 33.

Generation is inherited like failure, and this expression can be written more concisely by omitting the optional **do** clause:

```
every write(find("or", sentence))
```

There are several built-in generators in Icon. One of the most frequently used of these is

```
i to j
```

which generates the integers from **i** to **j**. This generator can be combined with **every-do** to formulate the traditional **for**-style control structure:

```
every k := i to j do
   f(k)
```

Note that this expression can be written more compactly as

```
every f(i to j)
```

There are a number of other control structures related to generation. One is *alternation*,

$expr_1$ | $expr_2$

which generates the results of $expr_1$ followed by the results of $expr_2$. Thus

```
every write(find("or", sentence1) | write("or", sentence2)
```

writes the positions of **or** in **sentence1** followed by the positions of **or** in **sentence2**. Again, this sentence can be written more compactly by using alternation in the second argument of **find**:

```
every write("or", sentence1 | sentence2)
```

Another use of alternation is illustrated by

```
(i | j | k) = (0 | 1)
```

which succeeds if any of **i**, **j**, or **k** has the value 0 or 1.

5. String Scanning

The string analysis and synthesis operations described in Sections 2 and 3 work best for relatively simple operations on strings. For complicated operations, the bookkeeping involved in keeping track of positions in strings becomes burdensome and error prone. In such cases, Icon has a string scanning facility that is analogous in many respects to pattern matching in SNOBOL4. In string scanning, positions are managed automatically and attention is focused on a current position in a string as it is examined by a sequence of operations.

The string scanning operation has the form

s ? *expr*

where **s** is the *subject* string to be examined and *expr* is an expression that performs the examination. A position in the subject, which starts at 1, is the focus of examination.

Matching functions change this position. One matching function, **move(i)**, moves the position by **i** and produces the substring of the subject between the previous and new positions. If the position cannot be moved by the specified amount (because the subject is not long enough), **move(i)** fails. A simple example is

```
line ? while write(move(2))
```

which writes successive two-character substrings of **line**, stopping when there are no more characters.

Another matching function is **tab(i)**, which sets the position in the subject to **i** and also returns the substring of the subject between the previous and new positions. For example,

```
line ? if tab(10) then write(tab(0))
```

first sets the position in the subject to 10 and then to the end of the subject, writing **line[10:0]**. Note that no value is written if the subject is not long enough.

String analysis functions such as **find** can be used in string scanning. In this context, the string that they operate on is not specified and is taken to be the subject. For example,

```
line ? while write(tab(find("or")))
   do move(2)
```

writes all the substrings of **line** prior to occurrences of **or**. Note that **find** produces a position, which is then used by **tab** to change the position and produce the desired substring. The **move(2)** skips the **or** that is found.

Another example of the use of string analysis functions in scanning is

```
line ? while tab(upto(letters)) do
   write(tab(many(letters)))
```

which writes all the words in **line**.

As illustrated in the examples above, any expression may occur in the scanning expression. Unlike SNOBOL4, in which the operations that are allowed in pattern matching are limited and idiosyncratic, string scanning is completely integrated with the rest of the operation repertoire of Icon.

6. Structures

6.1. Lists

While strings are sequences of characters, lists in Icon are sequences of values of arbitrary types. Lists are created by enclosing the lists of values in brackets. An example is

```
car1 := ["buick","skylark",1978,2450]
```

in which the list **car1** has four values, two of which are strings and two of which are integers. Note that the values in a list need not all be of the same type. In fact, any kind of value can occur in a list —even another list, as in

```
inventory := [car1,car2,car3,car4]
```

Lists also can be created by

```
a := list(i,x)
```

which creates a list of **i** values, each of which has the value **x**.

The values in a list can be referenced by position much like the characters in a string. Thus

```
car1[4] := 2400
```

changes the last value in **car1** to 2400. A reference that is out of the range of the list fails. For example,

```
write(car1[5])
```

fails.

The values in a list **a** are generated by **!a**. Thus

```
every write(!a)
```

writes all the values in **a**.

Lists can be manipulated like stacks and queues. The function **push(a,x)** adds the value of **x** to the left end of the list **a**, automatically increasing the size of **a** by one. Similarly, **pop(a)** removes the leftmost value from **a**, automatically decreasing the size of **a** by one, and produces the removed value.

A list value in Icon is a pointer (reference) to a structure. Assignment of a structure in Icon does not copy the structure itself but only the pointer to it. Thus the result of

```
demo := car1
```

causes **demo** and **car1** to reference the same list. Graphs with loops can be constructed in this way. For example,

```
node1 := ["a"]
node2 := [node1,"b"]
push(node1,node2)
```

constructs a structure that can be pictured as follows:

```
node1  .->a--.
       |     |
       |     |
node2  '--b<-'
```

6.2. Tables

Icon has a table data type similar to that of SNOBOL4. Tables essentially are sets of pairs of values, an *entry value* and a corresponding *assigned value*. The entry and assigned values may be of any type, and the assigned value for any entry value can be looked up automatically. Thus tables provide a form of associative access in contrast with the positional access to values in lists.

A table is created by an expression such as

```
symbols := table(x)
```

which assigns to **symbols** a table with the default assigned value **x**. Subsequently, **symbols** can be referenced by any entry value, such as

```
symbols["there"] := 1
```

which assigns the value 1 to the **there**th entry in symbols.

Tables grow automatically as new entry values are added. For example, the following program segment produces a table containing a count of the words that appear in the input file:

```
words := table(0)
while line := read() do
   line ? tab(upto(letters)) do
      words[tab(many(letters))] +:= 1
```

Here the default assigned value for each word is 0, as given in **table(0)**, and **+:=** is an augmented assignment operation that increments the assigned values by one. There are augmented assignment operations for all binary operators.

Tables can be converted to lists, so that their entry and assigned values can be accessed by position. This is done by **sort(t)**, which produces a list of two-element lists from **t**, where each two-element list consists of an entry value and its corresponding assigned value. For example,

```
wordlist := sort(words)
every pair := !wordlist do
   write(pair[1]," : ",pair[2])
```

writes the words and their counts from **words**.

7. Procedures

An Icon program consists of a sequence of procedure declarations. An example of a procedure declaration is

```
procedure max(i,j)
   if i > j then return i else return j
end
```

where the name of the procedure is **max** and its formal parameters are **i** and **j**.

The **return** expressions return the value of **i** or **j**, whichever is larger.

Procedures are called like built-in functions. Thus

```
k := max(*s1, *s2)
```

assigns to **k** the size of the longer of the strings **s1** and **s2**.

A procedure also may suspend instead of returning. In this case, a result is produced as in the case of a return, but the procedure can be resumed to produce other results. An example is the following procedure that generates the words in the input file.

```
procedure genword()
   local line, letters, words
   letters := &lcase ++ &ucase
   while line := read() do
      line ? while tab(upto(letters)) do {
         word := tab(many(letters))
         suspend word
         }
end
```

The braces enclose a compound expression.

Such a generator is used in the same way that a built-in generator is used. For example

```
every word := genword() do
   if find("or", word) then write word
```

writes only those words that contain the substring **or**.

8. An Example

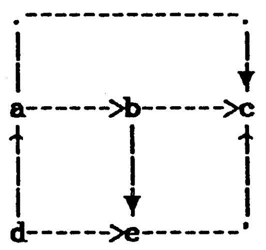
The following program sorts graphs topologically.

```
procedure main()
   local sorted, nodes, arcs, roots
   while nodes := read() do {                    # get next node list
      arcs := read()                             # get arc list
      sorted := ""                               # sorted nodes
                                                 # nodes without predecessor
      while *(roots := nodes -- snodes(arcs)) > 0 do {
         sorted ||:= roots                       # add to sorted nodes
         nodes --:= roots                        # delete these nodes
         arcs := delarcs(arcs, roots)            # delete their arcs
         }
      if *arcs = 0 then write(sorted)            # successfully sorted
      else write("graph has cycle")              # cycle if node remains
   }
end
```

```
procedure snodes(arcs)
   local nodes
   nodes := ""
   arcs ? while move(1) do {   # predecessor
      move(2)      # skip "->"
      nodes ||:= move(1)   # successor
      move(1)      # skip ";"
      }
   return nodes
end

procedure delarcs(arcs, roots)
   local newarcs, node
   newarcs := ""
   arcs ? while node := move(1) do {   # get predecessor node
      if many(roots, node) then move(4)     # delete arc from root node
      else newarcs ||:= node || move(4)  # else keep arc
      }
   return newarcs
end
```

Graph nodes are represented by single characters with a list of the nodes on one input line followed by a list of arcs. For example, the graph



is given as

```
abcde
a->b;a->c;b->c;b->e;d->a;d->e;e->c;
```

for which the output is

```
dabec
```

The nodes are represented by csets and automatic type conversion is used to convert strings to csets and vice versa. Note the use of augmented assignment operations for concatenation and in the computation of cset differences.

Acknowledgement

Icon was designed by the the author in collaboration with Dave Hanson, Tim Korb, Cary Coutant, and Steve Wampler. The current implementation is largely the work of Cary Coutant and Steve Wampler with recent contributions by Bill Mitchell. Dave Hanson and Bill Mitchell also made several helpful suggestions on the presentation of material in this paper.

References

1. Griswold, Ralph E., Poage, James F., and Polonsky, Ivan P. *The SNOBOL4 Programming Language*, second edition. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1971.
2. Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1978.
3. Griswold, Ralph E. *Differences Between Versions 2 and 5 of Icon.* Technical Report TR 83-5, Department of Computer Science, The University of Arizona. 1983.
4. Griswold, Ralph E. and Griswold, Madge T. *The Icon Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1983.

Release 5g of Icon

Release 5g of Icon is an implementation of Icon that runs on both PDP-11s and VAXs under the UNIX* operating system. This document is a brief summary of this release.

Changes

- System configuration is now performed by a shell script rather than having the installer manually edit a number of files.
- Interpretable files are made to appear to be executable on both the VAX and the PDP-11.
- The **&output** and **&errout** output streams are now line buffered by default. This provides a substantial performance improvement for programs that generate a large amount of terminal output.
- There are several language extensions to the Icon language that can be optionally included in a system.
- Several performance enhancements have been made to various components of the system. A number of minor bugs have been fixed.

Known Bugs

This list ennumerates all known bugs in Release 5g Icon. If you find a bug that is not in this list, please contact us.

- The translator does not detect arithmetic overflow in conversion of numeric literals. Very large numeric literals may have incorrect values.
- Integer overflow on multiplication and exponentiation are not detected during execution. This may occur during type conversion.
- Line numbers may be wrong in diagnostic messages related to lines with continued quoted literals.
- Program malfunction may occur if **display()** is used in a co-expression.
- In some cases, trace messages may show the return of subscripted values, such as **&null**[2], that would be erroneous if they were dereferenced.
- File names are truncated to 14 characters by UNIX. If such a truncation deletes part of the terminating **.icn** of a file that is input to the translator, mysterious diagnostic messages may occur during linking.
- On PDP-11s, list blocks can contain no more than 4090 elements. List blocks are created when the **list()** function is called, when literal lists are specified, and when the **sort()** function converts a table into a list. It should be noted that it is possible for a list to grow to beyond 4090 elements; the limitation is only upon the size of the list when it is created.

*UNIX is a trademark of Bell Laboratories.

- There is a bug in the 4.1bsd **fopen()** routine that under certain conditions returns a ~~**FILE** pointer that is out of range when one tries to open~~ too many files. On systems where this bug is present, it may manifest itself in the form of runtime Error 304 when one tries to open too many files. (On 4.1bsd systems this limit is usually 20 files.)
- If one has an expression like **x := create** ... in a loop, and **x** is not a global variable, the unreferenceable expression stacks generated by each successive **create** operation are not garbage collected. This problem can be circumvented by making **x** a global variable or by assigning a value to **x** before the **create** operation, e.g., **x := &null; x := create** ....
- Overflow of a co-expression stack due to excessive recursion is not detected and may cause mysterious program malfunction.
- The garbage collector was designed for machines with small address spaces and as such is not well-suited for machines like the VAX. No empirical studies have been made, but it is suspected that performance of the garbage collector can be improved substantially on the VAX. In particular, if the user attempts to create a very large data object that will not fit into memory, (such as a million-element list), it takes the system an inordinately long time to determine that the object can not be allocated.

March 14, 1983

Ralph E. Griswold
William H. Mitchell

Installation and Maintenance Guide for Release 5g of Icon

This document describes how to install Release 5g of the Icon programming language. The installation procedure is simple; it requires the distribution tape to be unloaded onto the target machine, the setting of site specific constants, and the recompilation of the Icon system itself. This document also contains information that may be of use to maintainers of Icon systems.

1. System Requirements

This distribution of Version 5 Icon is targeted for VAX-11s and PDP-11s (with separate instruction and data spaces) running the UNIX* operating system. This distribution package has been tested under V7 and 4.1bsd, and no problems should be encountered when installing Icon under one of these versions of UNIX.

When the system is unloaded, it requires about 1200 kilobytes of disk space. During recompilation, a total of 3000 kilobytes is required. Once the system has been built, it is possible to delete the source code. Such a configuration requires 900 kilobytes of disk space. These figures vary slightly depending upon the logical organization of a particular disk.

2. Installation Procedure

2.1. Unloading the Distribution Tape

The system is distributed as a *tar* archive on magnetic tape. The *tar* hierarchy is rooted at the directory **v5g**. Mount the distribution tape and do a *cd* to the directory that you wish to hold the system hierarchy.

The precise *tar* command to unload the distribution tape depends on the local environment. If you are on a VAX and the distribution tape density is 1600 bpi, the following command should extract the contents of the tape:

```
tar x
```

On a PDP-11 with a 1600 bpi distribution tape, use:

```
tar xf /dev/rmt0
```

For example, if you want the system to reside at **/usr/icon/v5g** on a VAX, you might type:

```
cd /usr
mkdir icon
cd icon
tar x
```

2.2. System Configuration

Note: File names used in the following sections are usually relative to the root directory for the Icon hierarchy. For example, if the Icon system is unloaded as described above, the root directory is **/usr/icon/v5g**, and the file name

int/bin/Makefile refers to **/usr/icon/v5g/int/bin/Makefile**.

The installer must perform a site-specific configuration of the Icon system. This configuration is accomplished the shell script **int/bin/icon-setup**. **icon-setup** accepts a number of parameters and modifies several source files to produce a ready-to-compile Icon system tailored as specified by the parameters.

icon-setup has the following synopsis:

icon-setup
  −vax −pdp11 −host *string*
  [**−hz** *rate*] [**−nofp**] [**−ext**] [**−directex**]
  [**−ibin** *library directory for the Icon interpreter*]
  [**−cbin** *library directory for the Icon compiler*]
  [**−iconx** *location of the Icon interpreter*]

The parameters have the following meanings:

−vax and **−pdp11**
These are mutually exclusive options that control the selection of machine dependent portions of code. Select one as appropriate for your machine.

−host *string*
The Icon system has a keyword, **&host**, whose value should be the name of the host machine where the system is running. On some systems, notably 4.2bsd, System III, and System V, it is possible to determine the name of the system at runtime via a system call. On other systems, 4.1bsd for example, the file **/usr/include/whoami.h** contains the name of the host in a **#define** statement. If neither of these methods is available, or one wishes to pick an arbitrary string for **&host**, that can be done. If you are on 4.2bsd, specify **gethost** for *string*, this will cause the *gethostname(2)* function to be used. If you are on System III, System V, or some other system that supports the *uname*(2) system call, specify **uname** for *string*. If you are on a system with a **/usr/include/whoami.h** file that has a **#define** for **sysname**, then specify **whoami** for *string*. If none of these are available on your machine, or if you want to give **&host** some value besides that of the machine name, specify an arbitrary string (you may need quotes around it) for *string*, for example: **−host UNIX**

−hz *rate*
This parameter specifies the cycle rate of the electrical environment you are in. The rate defaults to 60 Hz and you only need to specify **−hz** if you are not in a 60 Hz electrical environment. If the rate is incorrect, the value of the **&time** keyword will be wrong.

−nofp
Specify **−nofp** if you are on a PDP-11 that does not have floating point hardware.

−directex
Specify this if you are on a 4.1bsd or later system. This option causes the use of a feature of the *exec*(2) system call to make interpretable files directly executable.

−ext
Enables a number of extensions to standard Version 5 as described in [4]. These features may be of interest to persons wishing to experiment with extensions to Icon. However, the extensions are not supported.

−ibin *directory* and **−cbin** *directory*
The directories **int/bin** and **cmp/bin** contain executable versions of the Icon translator, the Icon linker, and various libraries. The path names of these directories are built into *icont* and *iconc*, the programs that control the translation of Icon programs. By default, the fully qualified names of **int/bin** and **cmp/bin** are

used for **–ibin** and **–cbin** respectively. Specifying **–ibin** and/or **–cbin** causes *icont* and/or *iconc* to use the specified directory as its library directory. If alternate directories are specified, after the Icon system is built, copy all of the files in **int/bin** and/or **cmp/bin** into the specified directories.

–iconx *interpreter-location*
By default, the *interpreter-location* is the fully qualified name of **int/bin/iconx**. If **–ibin** is specified, then *interpreter-location* defaults to *ibin-directory*/**iconx**. If the fully qualified name of **iconx** is longer than 32 characters and you are on a VAX running 4.1bsd, consult Section 3.1 to determine what to specify for **–ibin**.

Before **icon-setup** modifies a file, it copies a generic version of the file into place and works on it. Thus, **icon-setup** can be run a number of times and only the last run has any lasting effect. **icon-setup** is like any other UNIX command and thus all of its arguments must be specified on one command line.

For example, if you are on a VAX running 4.1bsd, all you need is

```
icon-setup -vax -directex -host whoami
```

If you have a PDP-11 running V7, you don't have a hardware floating point unit, and you want **&host** to be "Unix Version 7", you should use

```
icon-setup -pdp11 -nofp -host "Unix Version 7"
```

Suppose you are on a VAX running System V, you have a 50 hz electrical environment, you wish the library directories for the interpreter and the compiler to be **/usr/lib/icon/interp** and **/usr/lib/icon/comp** respectively, and the Icon interpreter to reside at **/usr/bin/iconx**. You also wish to include the language extensions. You would need:

```
icon-setup -vax -hz 50 -ext -host uname -ibin /usr/lib/icon/interp
  -cbin /usr/lib/icon/comp -iconx /usr/bin/iconx
```

Note that the last example must be entered as one logical command line.

2.3. System Compilation

This distribution of Icon contains no executable binary files and no object files, so the system must be completely recompiled from the source. After you have run **icon-setup**, you are ready to compile the system. Issuing a *make* command in **int/bin** or **cmp/bin** completely rebuilds the interpreter or the compiler, respectively. We recommend that you build the interpreter first and then if there are no problems, proceed to the compiler.

The complete system construction process is:

```
cd v5g
int/bin/icon-setup appropriate arguments
cd int/bin
make
cd ../../cmp/bin
make
```

At this point, you should have a working Icon system. The samples directory contains programs and associated test data. If you desire to test the system, consult the file **samples/Readme**.

You may wish to place a copy **int/bin/icont** and **cmp/bin/iconc** in a public directory such as **/usr/bin** or **/usr/local** to provide general user access to Icon. **docs/icont.1** and **docs/iconc.1** are *man* pages for *icont* and *iconc*; you may wish to copy the appropriate set into **/usr/man/man1**. Once you are sure that the system is working, you can issue the command **make clean-all** in **int/bin** and **cmp/bin** to remove all non-source files from the source directories.

3. Special Installation Considerations

3.1. Direct Execution of Interpretable Files

When an Icon program is processed by the interpreter translator and linker using the *icont* command, the result is a file containing opcodes and data in a format that the Icon interpreter understands. Rather than having the user "execute" this interpretable file by running the Icon interpreter with the file as an argument, the Icon system uses one of two methods to make the interpretable files appear to be directly executable.

In 4.1bsd (and later 4.nbsd) systems, a feature of *exec*(2) system call can be used to enable the interpretable file produced by the linker to appear to be directly executable. When *exec* is called with a file to execute, it examines the first two characters of the file. If the first two characters are #!, *exec* assumes that the next argument on the line is the name of a program for which the file is to serve as input. The program then is executed with the named file (the file that is being "executed") as its argument. Thus, instead of having to run the Icon interpreter with the interpretable file as input, the interpretable file can be executed directly.

An alternative method is used on systems whose *exec*(2) system call doesn't have this feature. An executable file is prepended to the data used by the interpreter. The executable portion of the file will merely run the Icon interpreter with the file itself and any supplied arguments as the arguments for the interpreter.

If **–directex** is specified for **icon-setup**, the former method will be used, otherwise, the latter method will be used. The first method is preferable in that the interpretable files are smaller and they start executing more quickly.

There is a potential complication in using the first method. The 4.1bsd *exec*(2) system call imposes a length limitation of 32 characters on the name of the program to be run. If the name exceeds 32 characters, execution of the interpretable file fails. For example, suppose the Icon interpreter (**iconx**) on a system is located at **/usr/csc/local/icon/v5g/int/bin/iconx**. This path name is longer than 32 characters, and is thus unsuitable for inclusion in interpretable files. One way to solve the problem is to link **/usr/csc/local/icon/v5g/int/bin/iconx** to **/usr/local/iconx**, and have interpretable files reference **/usr/local/iconx**. You may need to employ a similar solution on your system. Two things need to be done. First, find a filesystem location where a copy of **int/bin/iconx** can be referenced with a fully qualified path name that is no more than 32 characters long. Second, when you configure the system using **icon-setup**, specify the new location of **iconx** using the **–iconx** option. For example:

> **icon-setup** *other arguments* **–iconx /usr/local/iconx**

Be sure to place a copy of **iconx** in the specified location after the system is completely built. It is also possible to get around this problem by not specifying **–directex** and having Icon prepend the executable header on interpretable files.

3.2. PDP-11 Systems without Separate I/D Spaces

This version is designed to run on PDP-11s with separate instruction and data spaces. Some persons have decreased the sizes of various internal structures and have brought up Icon on machines without separate I/D spaces, but only small programs can be run. These adaptations to non-I/D machines were done with an earlier version of Icon and we do not have details of the changes required.

4. Assorted Information

The following sections contain information that is not needed during the installation process, but may be helpful in understanding the organization of the system.

4.1. System Layout

The Icon system has four top-level directories branching off from **v5g**:

docs	System documentation and man pages.
samples	Sample programs and associated data.
book	Source code for procedures appearing in [1].
int	Source code hierarchy for the Icon interpreter.
cmp	Source code hierarchy for the Icon compiler.

4.2. System Components

The following directories contain various components of the Icon system and branch off from both **cmp** and **int**. (The directory **iconx** only appears in **int**.)

tran	Soure code for the Icon translator.
link	Source code for the Icon linker.
h	Header files that are included in other source files.
functions	Source code for the Icon builtin functions.
lib	Source code for runtime support routines that are directly callable from an Icon program.
operators	Source code for the Icon operators.
rt	Source code for the Icon runtime system.
iconx	Source code for the Icon interpreter proper.
bin	Libraries and executable versions of various system components. The source for programs that control Icon translation also resides here.

4.3. Source File Linking

The bulk of the source code for the Icon compiler and interpreter is very similar and in most cases, analogous files are linked between the cmp and int hierarchies. For example, **int/functions/write.c** is linked to **cmp/functions/write.c**. In some cases, preprocessor control statements are used to select portions of code in a particular file depending upon whether the compiler or interpreter is being compiled. Occasionally, files that are linked may not be in analogous directories, for example, **int/iconx/start.s** and **int/iconx/init.c** are linked to **cmp/rt/start.s** and **cmp/rt/init.c** respectively. The only directories where some analogous source files are not linked are **int/link** and **cmp/link**. In addition, **Makefiles** are linked when possible.

Because the same source is used to produce versions of Icon on both VAXs and PDP-11s, the portions of the code that are machine dependent are bracketed with preprocessor control statements to select sections appropriate for each machine. This is primarily used for assembler source files, but also is used in a few cases for C source files. In some cases where differences are extreme (as for assembler files), the section of code for the VAX appears in its entireity and is followed by the code for the PDP-11 in its entireity.

If you decide to make source modifications, check link counts to be sure that you are not modifying more files than you think are, or less files than you think you are.

4.4. Recompilation of System Components

Execution of the *make* command while in any source code directory causes the system component in that directory to be rebuilt. Doing a *make* in **cmp/bin** or **int/bin** causes the appropriate system to be rebuilt.

The directories **iconx**, **tran**, and **link** each contain code for a single program. Doing a *make* in any of these directories causes the particular program to be remade. The resulting program may then be copied into the appropriate bin directory (**cmp/bin** or **int/bin**).

The directories **functions**, **lib**, **operators**, and **rt** each contain source code for a part of the Icon runtime system. The Icon interpreter, **iconx**, is formed by linking all the runtime subroutines together with the routines in **int/iconx**. Icon programs processed by the Icon compiler are linked together with a library (**cmp/bin/libi.a**) of the runtime system subroutines. When changes are made to the interpreter runtime system, all affected libraries must be rebuilt and then **iconx** must be rebuilt. For example, if the files **int/operators/bang.c** and **int/functions/read.c** have been modified, the following sequence of commands rebuilds the interpreter.

```
cd int/functions
make
cd ../operators
make
cd ../iconx
make
cp iconx ../bin           # copy new version of interpreter to int/bin
```

Alternatively, performing a **make** in **int/bin** has the same effect. Similarly, if the changes have been made in the compiler rather than in the interpreter, these commands rebuild the compiler:

```
cd cmp/functions
make
cd ../operators
make
cd ../bin
make lib                  # rebuild and randomize libi.a
```

If disk space is critical on your system, it is possible to delete all **v5g** directories except for **int/bin** and **cmp/bin**, and retain a functional Icon system. That configuration requires about 500 kilobytes of disk space.

4.5. PDP-11 yacc modifications for the Icon Translator

You only need to be concerned with this section if you are going to modify the Icon grammar, contained in the file **cmp/tran/ucon.g**. The version of *yacc* distributed with VAX systems is large enough to build the Icon parser [3], but it may be necessary to build a version of *yacc* with larger parameters if you are on a PDP-11. The following defined constants in the file **dextern** (in the *yacc* source directory) should be given the values listed below. Larger values are acceptable for all these constants, but are not necessary.

```
# ifdef HUGE
# define ACTSIZE 3000
# define MEMSIZE 6000
# define NSTATES 300
# define NTERMS 127
# define NPROD 200
# define NNONTERM 100
# define TEMPSIZE 1200
# define CNAMSZ 4100
# define LSETSIZE 200
# define WSETSIZE 200
# endif
```

The constant **HUGE** should be defined instead of **MEDIUM** at the end of the file **yacc/files**.

Then *yacc* should be rebuilt.

4.6. Obtaining Source Code Listings

Execution of the command

```
make Listall
```

in any of the directories except **bin** produces listings of all source files from that directory on standard output. Use the command

```
make List
```

to obtain listings of all files that have been altered since the last **make List** or **make Listall**.

Electronic Mail and Problem Reporting

A mailbox has been established to facilitate communication with us, should you have problems or questions. Use the following addresses for electronic mail:

icon-project.arizona@rand-relay	(CSNET and ARPANET)
arizona!icon-project	(Usenet and uucpnet)

We currently have uucp connections established through **gi**, **mcnc**, **ucbvax**, and **utah-cs**.

If you encounter any problems with the Icon system, or if you have any suggestions about Icon in general or the installation process in particular, please contact us. If you do not have access to electronic mail facilities, please use the Trouble Report Forms supplied in the distribution package or call Ralph Griswold at 602-626-1829 (602-621-6609 after April 8, 1983).

References

1. Ralph E. Griswold and Madge T. Griswold, *The Icon Programming Language*. Prentice-Hall Inc., Englewood Cliffs, New Jersey. 1983.

2. *Tape Directory for Release 5g of Icon*. Department of Computer Science, The University of Arizona. March 1983.

3. Coutant, Cary A. and Wampler, Stephen B. *A Tour Through the C Implementation of Icon; Version 5*. Technical Report TR 81-11a, Department of Computer Science, The University of Arizona. December 1981.

4. Ralph E. Griswold and William H. Mitchell, *Experimental Extensions to Icon*. Department of Computer Science, The University of Arizona. March 1983.

Ralph E. Griswold (**ralph.arizona@rand-relay, arizona!ralph**)
William H. Mitchell (**whm.arizona@rand-relay, arizona!whm**)

March 12, 1983

Notes on the Icon "Tour" (TR 81-11a)

A technical report, TR 81-11a, describing the implementation of Icon, is enclosed with Release 5g. This report is based on the original Version 5 implementation done for the PDP-11. While most of the information in this report is machine independent, the machine dependent sections refer to the PDP-11. A new report is in preparation and will be available at some point in the future.

One important difference related to external functions should be noted. Since the 4.1bsd C compiler does not initialize unions, procedure blocks for external functions must be declared and initialized using the **b_iproc** structure rather than the **b_proc** structure as shown in TR 81-11a. (Procedure blocks are declared after the body of the function.) The **b_iproc** structure is identical to the **b_proc** structure except that it uses the **sdescrip** structure rather than the **descrip** structure (**descrip** has a union, **sdescrip** does not) to hold the name of the procedure. The **b_iproc** structure should only be used for declaration and initialization of data blocks for functions.

March 10, 1983

Ralph E. Griswold
William H. Mitchell

Experimental Extensions to Version 5 of Icon

Ralph E. Griswold and William H. Mitchell

Department of Computer Science, The University of Arizona

March 3, 1983

A number of experimental extensions have been made to Version 5 of Icon. This version is called Version 5x for reference.

The extensions in Version 5x, which are described below, are upward-compatible ones and should not interfere with programs that run properly on Version 5.

1. PDCO Invocation Syntax

Version 5x contains the procedure invocation syntax that is used for programmer-defined control operations (see TR 82-8 and TR 82-16).

In this syntax, when braces are used in place of parentheses to enclose an argument list, the arguments are passed as a list of co-expressions. That is,

```
p{...}
```

is equivalent to

```
p([create , create , . create ])
```

There is always at least one co-expression in the list:

```
p{}
```

is equivalent to

```
p([create &null])
```

2. Invocation Via String Name

In the Version 5x interpreter (but not compiler), a string-valued expression that corresponds to the name of a procedure or operation may be used in place of the procedure or operation in an invocation expression. For example,

```
"image"(x)
```

produces the same call as

```
image(x)
```

and

```
"-"(i,j)
```

is equivalent to

```
i - j
```

In the case of operations, the number of arguments determines the operation. Thus

```
"-"(i)
```

is equivalent to

```
-i
```

Since **to-by** is an operation, despite its reserved-word syntax, it is included in this facility with the string name .... Thus

```
"..."(1,10,2)
```

is equivalent to

```
1 to 10 by 2
```

Similarly, range specifications are represented by ":", so that

```
":"(s,i,j)
```

is equivalent to

```
s[i:j]
```

Defaults are not provided for omitted or null-valued arguments in this facility. Consequently,

```
"..."(1,10)
```

results in a run-time error when it is evaluated.

The subscripting operation also is available with the string name [ ]. Thus

```
"[]"(&lcase,3)
```

produces **c**.

String names are available for all the operations in Icon, but not for control structures. Thus

```
"|"(,)
```

is erroneous.

String names for procedures are available through global identifiers. Note that the names of functions, such as **image** are global identifiers. Similarly, any procedure-valued global identifier may be used as the string name of a procedure. Thus in

```
global q

procedure main()
  q := p
  "q"("hi")
end

procedure p(s)
  write(s)
end
```

the procedure **p** is invoked via the global identifier **q**.

3. Integer Sequences

To facilitate the generation of integer sequences that have no limit, the function **seq(i,j)** has been added. This function has the result sequence {i, i+j, i+2j, }. Omitted or null values for **i** and **j** default to 1. Thus the result sequence for **seq()** is { 1, 2, 3, }.

MH

A Mail Handling System for UNIX

October, 1979

Bruce Borden

The Rand Corporation
1700 Main Street
Santa Monica, CA 90406

(213) 399-0568 x 7463

CONTENTS

PREFACE

This report describes a system for dealing with messages transmitted on a computer. Such messages might originate with other users of the same computer or might come from an outside source through a network to which the user's computer is connected. Such computer-based message systems are becoming increasingly widely used, both within and outside the Department of Defense.

The message handling system MH was developed for two reasons. One was to investigate some research ideas concerning how a message system might take advantage of the architecture of the UNIX time-sharing operating system for Digital Equipment Corporation PDP-11 and VAX computers, and the special features of UNIX's command-level interface with the user (the "shell"). The other reason was to provide a better and more adaptable base than that of conventional designs on which to build a command and control message system. The effort has succeeded in both regards, although this report mainly describes the message system itself and how it fits in with UNIX. The main research results are being described and analyzed in a forthcoming Rand report. The system is currently being used as part of a tactical command and control "laboratory," which is also being described in a separate report.

The present report should be of interest to three groups of readers. First, it is a complete reference manual for the users of MH (although users outside of Rand must take into account differences from the local Rand operating system). Second, it should be of interest to those who have a general knowledge of computer-based message systems, both in civilian and military applications. Finally, it should be of interest to those who build large subsystems that interface with users, since it illustrates a new approach to such interfaces.

The MH system was developed by the first author, using an approach suggested by the other two authors. Valuable assistance was provided by Phyllis Kantar in the later stages of the system's implementation. Several colleagues contributed to the ideas included in this system, particularly Robert Anderson and David Crocker. In addition, valuable experience in message systems, and a valuable source of ideas, was available to us in the form of a previous message system for UNIX called MS, designed at Rand by David Crocker.

This report was prepared as part of the Rand project entitled "Data Automation Research", sponsored by Project AIR FORCE.

SUMMARY

Electronic communication of text messages is becoming commonplace. Computer-based message systems—software packages that provide tools for dealing with messages—are used in many contexts. In particular, message systems are becoming increasingly important in command and control and intelligence applications.

This report describes a message handling system called MH. This system provides the user with tools to compose, send, receive, store, retrieve, forward, and reply to messages. MH has been built on the UNIX time-sharing system, a popular operating system developed for the DEC PDP-11 and VAX classes of computers.

A complete description of MH is given for users of the system. For those who do not intend to use the system, this description gives a general idea of what a message system is like. The system involves some new ideas about how large subsystems can be constructed. These design concepts and a comparison of MH with other message systems will be published in a forthcoming Rand report.

The interesting and unusual features of MH include the following: The user command interface to MH is the UNIX "shell" (the standard UNIX command interpreter). Each separable component of message handling, such as message composition or message display, is a separate command. Each program is driven from and updates a private user environment, which is stored as a file between program invocations. This private environment also contains information to "custom tailor" MH to the individual's tastes. MH stores each message as a separate file under UNIX, and it utilizes the tree-structured UNIX file system to organize groups of files within separate directories or "folders." All of the UNIX facilities for dealing with files and directories, such as renaming, copying, deleting, cataloging, off-line printing, etc., are applicable to messages and directories of messages (folders). Thus, important capabilities needed in a message system are available in MH without the need (often seen in other message systems) for code that duplicates the facilities of the supporting operating system. It also allows users familiar with the shell to use MH with minimal effort.

1. INTRODUCTION

Although people can travel cross-country in hours and can reach others by telephone in seconds, communications still depend heavily upon paper, most of which is distributed through the mails.

There are several major reasons for this continued dependence on written documents. First, a written document may be proofread and corrected prior to its distribution, giving the author complete control over his words. Thus, a written document is better than a telephone conversation in this respect. Second, a carefully written document is far less likely to be misinterpreted or poorly translated than a phone conversation. Third, a signature offers reasonable verification of authorship, which cannot be provided with media such as telegrams.

However, the need for fast, accurate, and reproducible document distribution is obvious. One solution in widespread use is the telefax. Another that is rapidly gaining popularity is electronic mail. Electronic mail is similar to telefax in that the data to be sent are digitized, transmitted via phone lines, and turned back into a document at the receiver. The advantage of electronic mail is in its compression factor. Whereas a telefax must scan a page in very fine lines and send all of the black and white information, electronic mail assigns characters fixed codes which can be transmitted as a few bits of information. Telefax presently has the advantage of being able to transmit an arbitrary page, including pictures, but electronic mail is beginning to deal with this problem. Electronic mail also integrates well with current directions in office automation, allowing documents prepared with sophisticated equipment at one site to be quickly transferred and printed at another site.

Currently, most electronic mail is intraorganizational, with mail transfer remaining within one computer. As computer networking becomes more common, however, it is becoming more feasible to communicate with anyone whose computer can be linked to your own via a network.

The pioneering efforts on general-purpose electronic mail were by organizations using the Defense Department's ARPANET.[1] The capability to send messages between computers existed before the ARPANET was developed, but it was used only in limited ways. With the advent of the ARPANET, tools began to be developed which made it convenient for individuals or organizations to distribute messages over broad geographic areas, using diverse computer facilities. The interest and activity in message systems has now reached such proportions that steps have been taken within the DoD to coordinate and unify the development of military message systems. The use of electronic mail is expected to increase dramatically in the next few years. The utility of such systems in the command and control and intelligence environments is clear, and applications in these areas will probably lead the way. As the costs for sending and handling electronic messags continue their rapid decrease, such uses can be expected to spread rapidly into other areas and, of course, will not be limited to the DoD.

A message system provides tools that help users (individuals or organizations) deal with messages in various ways. Messages must be composed, sent, received, stored, retrieved, forwarded, and replied to. Today's best interactive computer systems provide a variety of word-processing and information handling capabilities. The message handling facilities should be well integrated with the rest of the system, so as to be a graceful extension of overall system capability.

The message system described in this report, MH, provides most of the features that can be found in other message systems and also incorporates some new ones. It has been built on the UNIX time-sharing system,[2] a popular operating system for the DEC PDP-11 and VAX classes of computers. A "secure" operating system similar to UNIX is currently being developed,[3] and that system will also run MH.

This report provides a complete description of MH and thus may serve as a user's manual, although parts of the report will be of interest to non-users as well. Sections 2 and 3, the Overview and Tutorial, present the key ideas of MH and will give those not familiar with message systems an idea of what such systems are like.

MH consists of a set of commands which use some special files and conventions. Section 4 covers the information a user needs to know in addition to the commands. The final section, Sec. 5, describes each of the MH commands in detail. A summary of the commands is given in Appendix A, and Appendixes B and C describe the ARPANET conventions for messages (we expect that many users of MH will be using the ARPANET) and the formal syntax of such messages, respectively. Finally, Appendix D provides an illustration of how MH commands may be used in conjunction with other UNIX facilities.

A novel approach has been taken in the design of MH. The design concept will be reported in detail in a forthcoming Rand report, but it can be described briefly as follows. Instead of creating a large subsystem that appears as a single command to the user, (such as MS[4]) MH is a collection of separate commands which are run as separate programs. The file and directory system of UNIX are used directly. Messages are stored as individual files (datasets), and collections of them are grouped into directories. In contrast, most other message systems store messages in a complicated data structure within a monolithic file. With the MH approach, UNIX commands can be interleaved with commands invoking the functions of the message handler. Conversely, existing UNIX commands can be used in connection with messages. For example, all the usual UNIX editing, text-formatting, and printing facilities can be applied directly to individual messages. MH, therefore, consists of a relatively small amount of new code; it makes extensive use of other UNIX software to provide the capabilities found in other message systems.

2. OVERVIEW

There are three main aspects of MH: the way messages are stored (the message database), the user's profile (which directs how certain actions of the message handler take place), and the commands for dealing with messages.

Under MH, each message is stored as a separate file. A user can take any action with a message that he could with an ordinary file in UNIX. A UNIX directory in which messages are stored is called a folder. Each folder contains some standard entries to support the message-handling functions. The messages in a folder have numerical names. These folders (directories) are entries in a particular directory path, described in the user profile, through which MH can find message folders. Using the UNIX "link" facility, it is possible for one copy of a message to be "filed" in more than one folder, providing a message index facility. Also, using the UNIX tree-structured file system, it is possible to have a folder within a folder. This two-level organization provides a "selection-list" facility, with the full power of the MH commands available on selected sublists of messages.

Each user of MH has a user profile, a file in his $HOME (initial login) directory called ". mh_profile". This profile contains several pieces of information used by the MH commands: a path name to the directory that contains the message folders, information concerning which folder the user last referenced (the "current" folder), and parameters that tailor MH commands to the individual user's requirements. It also contains most of the necessary state information concerning how the user is dealing with his messages, enabling MH to be implemented as a set of individual UNIX commands, in contrast to the usual approach of a monolithic subsystem.

In MH, incoming mail is appended to the end of a file called . mail in a user's $HOME directory. The user adds the new messages to his collection of MH messages by invoking the command *inc*. *Inc* (incorporate) adds the new messages to a folder called "inbox", assigning them names which are consecutive integers starting with the next highest integer available in inbox. *Inc* also produces a *scan* summary of the messages thus incorporated.

There are four commands for examining the messages in a folder: *show*, *prev*, *next*, and *scan*. *Show* displays a message in a folder, *prev* displays the message preceding the current message, and *next* displays the message following the current message. *Scan* summarizes the messages in a folder, producing one line per message, showing who the message is from, the date, the subject, etc.

The user may move a message from one folder to another with the command *file*. Messages may be removed from a folder by means of the command *rmm*. In addition, a user may query what the current folder is and may specify that a new folder become the current folder, through the command *folder*.

A set of messages based on content may be selected by use of the command *pick*. This command searches through messages in a folder and selects those that match a given criterion. A subfolder is created within the original folder, containing links to all the messages that satisfy the selection criteria.

A message folder (or subfolder) may be removed by means of the command *rmf*.

There are five commands enabling the user to create new messages and send them: *comp*, *dist*, *forw*, *repl*, and *send*. *Comp* provides the facility for the user to compose a new message; *dist* redistributes mail to additional addressees; *forw* enables the user to forward messages; and *repl* facilitates the generation of a reply to an incoming message. If a message is not sent directly by one of these commands, it may be sent at a later time using the command *send*.

All of the elements summarized above are described in more detail in the following sections. Many of the normal facilities of UNIX provide additional capabilities for dealing with messages in various ways. For example, it is possible to print messages on the line-printer without requiring any additional code within MH. Using standard UNIX facilities, any terminal output can be redirected to a file for repeated or future viewing. In general, the flexibility and capabilities of the UNIX interface with the user are preserved as a result of the integration of MH into the UNIX structure.

3. TUTORIAL

This tutorial provides a brief introduction to the MH commands. It should be sufficient to allow the user to read his mail, do some simple manipulations of it, and create and send messages.

A message has two major pieces: the header and the body. The body consists of the text of the message (whatever you care to type in). It follows the header and is separated from it by an empty line. (When you compose a message, the form that appears on your terminal shows a line of dashes after the header. This is for convenience and is replaced by an empty line when the message is sent.) The header is composed of several components, including the subject of the message and the person to whom it is addressed. Each component starts with a name and a colon; components must not start with a blank. The text of the component may take more than one line, but each continuation line must start with a blank. Messages typically have "to:", "cc:", and "subject:" components. When composing a message, you should include the "to:" and "subject:" components; the "cc:" (for people you want to send copies to) is not necessary.

The basic MH commands are *inc*, *scan*, *show*, *next*, *prev*, *rmm*, *comp*, and *repl*. These are described below.

inc

When you get the message "You have mail", type the command *inc*. You will get a "scan listing" such as:

```
7+   7/13  Cas         revival of measurement work
8   10/ 9  Norm        NBS people and publications
9   11/26  To:norm     question <<Are there any functions
```

This shows the messages you received since the last time you executed this command (*inc* adds these new messages to your inbox folder). You can see this list again, plus a list of any other messages you have, by using the *scan* command.

scan

The scan listing shows the message number, followed by the date and the sender. (If you are the sender, the addressee in the "to:" component is displayed. You may send yourself a message by including your name among the "to:" or "cc:" addressees.) It also shows the message's subject; if the subject is short, the first part of the body of the message is included after the characters ≪.

show

This command shows the current message, that is, the first one of the new messages after an *inc*. If the message is not specified by name (number), it is generally the last message referred to by an MH command. For example,

show 5 will show message 5.

You can use the show command to copy a message or print a message.

show > x	will copy the message to file x.
show | print	will print the message, using the *print* command.
next	will show the message that follows the current message.
prev	will show the message previous to the current message.
rmm	will remove the current message.
rmm 3	will remove message 3.

comp

The *comp* command puts you in the editor to write or edit a message. Fill in or delete the "to:", "cc:", and "subject:" fields, as appropriate, and type the body of the message. Then exit normally from the editor. You will be asked "What now?". Type a carriage return to see the options. Typing **send** will cause the message to be sent; typing **quit** will cause an exit from *comp*, with the message draft saved.

If you quit without sending the message, it will be saved in a file called /usr/<name>/Mail/draft (where /usr/<name> is your $HOME directory). You can edit this file and send the message later, using the *send* command.

comp –editor prompter

This command uses a different editor and is useful for preparing "quick and dirty" messages. It prompts you for each component of the header. Type the information for that component, or type a carriage return to omit the component. After that, type the body of the message. Backspacing is the only form of editing allowed with this editor. When the body is complete, type a carriage return followed by <CTRL-D> (<OPEN> on Ann Arbor terminals). This completes the initial preparation of the message; from then on, use the same procedures as with *comp* (above).

repl
repl n

This command makes up an initial message form with a header that is appropriate for replying to an existing message. The message being answered is the current message if no message number is mentioned, or n if a number is specified. After the header is completed, you can finish the message as in *comp* (above).

This is enough information to get you going using MH. There are more commands, and the commands described here have more features. Subsequent sections explain MH in complete detail. The system is quite powerful if you want to use its sophisticated features, but the foregoing commands suffice for sending and receiving messages.

There are numerous additional capabilities you may wish to explore. For example, the *pick* command will select a subset of messages based on specified criteria such as sender or subject. Groups of messages may be designated, as described in Sec. V, under "Message Naming". The file ". mh_profile" can be used to tailor your use of the message system to your needs and preferences, as described in Sec. V, under "The User Profile". In general, you may learn additional features of the system selectively, according to your requirements, by studying the relevant sections of this manual. There is no need to learn all the details of the system at once.

4. DETAILED DESCRIPTION

This section describes the MH system in detail, including the components of the user profile, the conventions for message naming, and some of the other MH conventions. Readers who are generally familiar with computer systems will be able to follow the principal ideas, although some details may be meaningful only to those familiar with UNIX.

THE USER PROFILE

The first time an MH command is issued by a new user, the system prompts for a "path" and creates an MH "profile".

Each MH user has a profile which contains current state information for the MH package and, optionally, tailoring information for each individual program. When a folder becomes the current folder, it is recorded in the user's profile. Other profile entries control the MH path (where folders and special files are kept), folder and message protections, editor selection, and default arguments for each MH program.

The MH profile is stored in the file ". mh_profile" in the user's $HOME directory. It has the format of a message without any body. That is, each profile entry is on one line, with a keyword followed by a colon (:) followed by text particular to the keyword.
☛ *This file must not have blank lines.*
The keywords may have any combination of upper and lower case. (See Appendix B for a description of message formats.)

For the average MH user, the only profile entry of importance is "Path". Path specifies a directory in which MH folders and certain files such as "draft" are found. The argument to this keyword must be a legal UNIX path that names an existing directory. If this path is unrooted (i.e., does not begin with a /), it will be presumed to start from the user's $HOME directory. All folder and message references within MH will relate to this path unless full path names are used.

Message protection defaults to 664, and folder protection to 751. These may be changed by profile entries "Msg-Protect" and "Folder-Protect", respectively. The argument to these keywords is an octal number which is used as the UNIX file mode.[1]

When an MH program starts running, it looks through the user's profile for an entry with a keyword matching the program's name. For example, when *comp* is run, it looks for a "comp" profile entry. If one is found, the text of the profile entry is used as the default switch setting until all defaults are overridden by explicit switches passed to the program as arguments. Thus the profile entry "comp: –form standard.list" would direct *comp* to use the file "standard.list" as the message skeleton. If an explicit form switch is given to the *comp* command, it will override the switch obtained from

[1]See *chmod*(I) in the *UNIX Programmer's Manual*.[5]

the profile.

In UNIX, a program may exist under several names, either by linking or aliasing. The actual invocation name is used by an MH program when scanning for its profile defaults. Thus, each MH program may have several names by which it can be invoked, and each name may have a different set of default switches. For example, if *comp* is invoked by the name *icomp*, the profile entry "icomp" will control the default switches for this invocation of the *comp* program. This provides a powerful definitional facility for commonly used switch settings.

The default editor for editing within *comp*, *repl*, *forw*, and *dist*, is "/bin/ned".[2] A different editor may be used by specifying the profile entry "Editor: ". The argument to "Editor" is the name of an executable program or shell command file which can be found via the user's $PATH defined search path, excluding the current directory. The "Editor:" profile specification may in turn be overridden by a "—editor <editor>" profile switch associated with *comp*, *repl*, *forw*, or *dist*. Finally, an explicit editor switch specified with any of these four commands will have ultimate precedence.

During message composition, more than one editor may be used. For example, one editor (such as *prompter*) may be used initially, and a second editor may be invoked later to revise the message being composed (see the discussion of *comp* in Section 5 for details). A profile entry "<lasteditor>—next: <editor>" specifies the name of the editor to be used after a particular editor. Thus "comp: —e prompter" causes the initial text to be collected by *prompter*, and the profile entry "prompter—next: ed" names ed as the editor to be invoked for the next round of editing.

Some of the MH commands, such as *show*, can be used on message folders owned by others, if those folders are readable. However, you cannot write in someone else's folder. All the MH command actions not requiring write permission may be used with a "read-only" folder. In a writable folder, a file named "cur" is used to contain its current message name. For read-only folders, the current message name is stored in the user's profile.

Table 1 lists examples of the currently defined profile entries, typical arguments, and the programs that reference the entries.

Table 1

PROFILE COMPONENTS

Keyword and Argument	MH Programs that Use Component
Path: Mail	All
Current-Folder: inbox	Most
Editor: /bin/ed	*comp, dist, forw, repl*
Msg—Protect: 644	*inc*

[2]See Ref. 6 for a description of the NED text editor.

Folder–Protect: 711	*file, inc, pick*
<program>: default switches	All
cur–<read-onlyfolder>: 172	Most
prompter–next: ed	*comp, dist, forw, repl*

Path <u>should</u> be present. Folder is maintained automatically by many MH commands (see the "Context" sections of the individual commands in Sec. V). All other entries are optional, defaulting to the values described above.

MESSAGE NAMING

Messages may be referred to explicitly or implicitly when using MH commands. A formal syntax of message names is given in Appendix C, but the following description should be sufficient for most MH users. Some details of message naming that apply only to certain commands are included in the description of those commands.

Most of the MH commands accept arguments specifying one or more folders, and one or more messages to operate on. The use of the word "msg" as an argument to a command means that exactly one message name may be specified. A message name may be a number, such as 1, 33, or 234, or it may be one of the "reserved" message names: first, last, prev, next, and cur. (As a shorthand, a period (.) is equivalent to cur.) The meanings of these names are straightforward: "first" is the first message in the folder; "last" is the last message in the folder; "prev" is the message numerically previous to the current message; "next" is the message numerically following the current message; "cur" (or ". ") is the current message in the folder.

The default in commands that take a "msg" argument is always "cur".

The word "msgs" indicates that several messages may be specified. Such a specification consists of several message designations separated by spaces. A message designation is either a message name or a message range. A message range is a specification of the form name1–name2 or name1:n, where name1 and name2 are message names and n is an integer. The first form designates all the messages from name1 to name2 inclusive; this must be a non-empty range. The second form specifies up to n messages, starting with name1 if name1 is a number, or first, cur, or next, and ending with name1 if name1 is last or prev. This interpretation of n is overridden if n is preceded by a plus sign or a minus sign; +n always means up to n messages starting with name1, and –n always means up to n messages ending with name1. Repeated specifications of the same message have the same effect as a single specification of the message. Examples of specifications are:

```
1 5 7–11 22
first 6 8 next
first–10
last:5
```

The message name "all" is a shorthand for "first–last", indicating all of the messages in the folder.

The limit on the number of messages in an expanded message list is generally 999–the maximum number of messages in a folder. However, the *show* command and the commands '*pick –scan*' and '*pick –show*' are constrained to have argument lists that are no more than 512 characters long. (Under Version 7 UNIX this limit is 4096.)

In commands that accept "msgs" arguments, the default is either cur or all, depending on which makes more sense.

In all of the MH commands, a plus sign preceding an argument indicates a folder name. Thus, "+inbox" is the name of the user's standard inbox. If an explicit folder argument is given to an MH command, it will become the current folder (that is, the "Current-Folder:" entry in ". mh_profile" will be changed to this folder). In the case of the *file* and *pick* commands, which can have multiple output folders, a new source folder (other than the default current folder) is specified by "–src +folder".

OTHER MH CONVENTIONS

One very powerful feature of MH is that the MH commands may be issued from any current directory, and the proper path to the appropriate folder(s) will be taken from the user's profile. If the MH path is not appropriate for a specific folder or file, the automatic prepending of the MH path can be avoided by beginning a folder or file name with /. Thus any specific full path may be specified.

Arguments to the various programs may be given in any order, with the exception of a few switches whose arguments must follow immediately, such as "–src +folder" for *pick* and *file*.

Whenever an MH command prompts the user, the valid options will be listed in response to a <RETURN>. (The first of the listed options is the default if end-of-file is encountered, such as from a command file.) A valid response is any *unique* abbreviation of one of the listed options.

Standard UNIX documentation conventions are used in this report to describe MH command syntax. Arguments enclosed in brackets ([ ]) are optional; exactly one of the arguments enclosed within braces ({ }) must be specified, and all other arguments are required. The use of ellipsis dots (...) indicates zero or more repetitions of the previous item. For example, "+folder ..." would indicate that one or more "+folder" arguments is required and "[+folder ...]" indicates that 0 or more "+folder" arguments may be given.

MH departs from UNIX standards by using switches that consist of more than one character, e.g. "–header". To minimize typing, only a unique abbreviation of a switch need be typed; thus, for "–header", "–hea" is probably sufficient, depending on the other switches the command accepts. Each MH program accepts the switch "–help" (which *must* be spelled out fully) and produces a syntax description and a list of switches. In the list of switches, parentheses indicate required characters. For example, all "–help" switches will appear as "–(help)", indicating that no abbreviation is accepted.

Many MH switches have both on and off forms, such as "–format" and "–noformat". In many of the descriptions in Sec. V, only one form is defined; the other form, often used to nullify profile switch settings, is assumed to be the opposite.

MH COMMANDS

The MH package comprises 16 programs:

comp	Compose a message
dist	Redistribute a message
file	Move messages between folders
folder	Select/list status of folders
forw	Forward a message
inc	Incorporate new mail
next	Show the next message
pick	Select a set of messages by context
prev	Show the previous message
prompter	Prompting editor front end for composing messages
repl	Reply to a message
rmf	Remove a folder
rmm	Remove messages
scan	Produce a scan listing of selected messages
send	Send a previously composed message
show	Show messages

These programs are described below. The form of the descriptions conforms to the standard form for the description of UNIX commands.

NAME

comp – compose a message

SYNOPSIS

comp [–editor editor] [–form formfile] [file] [–use] [–nouse] [–help]

DESCRIPTION

Comp is used to create a new message to be mailed. If *file* is not specified, the file named "draft" in the user's MH directory will be used. *Comp* copies a message form to the file being composed and then invokes an editor on the file. The default editor is /bin/ned, which may be overridden with the '–editor' switch or with a profile entry "Editor:". (See Ref. 5 for a description of the NED text editing system.) The default message form contains the following elements:

```
To:
cc:
Subject:
----------
```

If the file named "components" exists in the user's MH directory, it will be used instead of this form. If '–form formfile' is specified, the specified formfile (from the MH directory) will be used as the skeleton. The line of dashes or a blank line must be left between the header and the body of the message for the message to be identified properly when it is sent (see *send*;). The switch '–use' directs *comp* to continue editing an already started message. That is, if a *comp* (or *dist*, *repl*, or *forw*) is terminated without sending the message, the message can be edited again via "comp –use".

If the specified file (or draft) already exists, *comp* will ask if you want to delete it before continuing. A reply of **No** will abort the *comp*, **yes** will replace the existing draft with a blank skeleton, **list** will display the draft, and **use** will use it for further composition.

Upon exiting from the editor, *comp* will ask "What now?". The valid responses are **list**, to list the draft on the terminal; **quit**, to terminate the session and preserve the draft; **quit delete**, to terminate, then delete the draft; **send**, to send the message; **send verbose**, to cause the delivery process to be monitored; **edit <editor>**, to invoke <editor> for further editing; and **edit**, to re-edit using the same editor that was used on the preceding round unless a profile entry "<lasteditor>–next: <editor>" names an alternative editor.

Files

/etc/mh/components	The message skeleton
or <mh-dir>/components	Rather than the standard skeleton
$HOME/.mh_profile	The user profile
<mh-dir>/draft	The default message file
/usr/bin/send	To send the composed message

Profile Components

Path:	To determine the user's MH directory
Editor:	To override the use of /bin/ned as the default editor
<lasteditor>—next:	To name an editor to be used after exit from <lasteditor>

Defaults

'file' defaults to draft
'—editor' defaults to /bin/ned
'—nouse'

Context

Comp does not affect either the current folder or the current message.

NAME

dist – redistribute a message to additional addresses

SYNOPSIS

dist [+folder] [msg] [–form formfile] [–editor editor] [–annotate]
[–noannotate] [–inplace] [–noinplace] [–help]

DESCRIPTION

Dist is similar to *forw*. It prepares the specified message for redistribution to addresses that (presumably) are not on the original address list. The file "distcomps" in the user's MH directory, or a standard form, or the file specified by '–form formfile' will be used as the blank components file to be prepended to the message being distributed. The standard form has the components "Distribute-to:" and "Distribute-cc:". When the message is sent, "Distribution-Date: date", "Distribution-From: name", and "Distribution-Id: id" (if '–msgid' is specified to *send;*) will be prepended to the outgoing message. Only those addresses in "Distribute-To", "Distribute-cc", and "Distribute-Bcc" will be sent. Also, a "Distribute-Fcc: folder" will be honored (see *send;*).

Send recognizes a message as a redistribution message by the existence of the field "Distribute-To:", so don't try to redistribute a message with only a "Distribute-cc:".

If the '–annotate' switch is given, each message being distributed will be annotated with the lines:

Distributed: ≪date≫
Distributed: Distribute-to: names

where each "to" list contains as many lines as required. This annotation will be done only if the message is sent directly from *dist*. If the message is not sent immediately from *dist* (i.e., if it is sent later via *send;*), "comp –use" may be used to re-edit and send the constructed message, but the annotations won't take place. The '–inplace' switch causes annotation to be done in place in order to preserve links to the annotated message.

See *comp* for a description of the '–editor' switch and for options upon exiting from the editor.

Files

/etc/mh/components	The message skeleton
or <mh-dir>/components	Rather than the standard skeleton
$HOME/.mh_profile	The user profile
<mh-dir>/draft	The default message file
/usr/bin/send	To send the composed message

Profile Components

Path:	To determine the user's MH directory
Editor:	To override the use of /bin/ned as the default editor
<lasteditor>–next:	To name an editor to be used after exit from <lasteditor>

Defaults

'+folder' defaults to the current folder
'msg' defaults to cur
'—editor' defaults to /bin/ned
'—noannotate'
'—noinplace'

Context

If a +folder is specified, it will become the current folder, and the current message will be set to the message being redistributed.

NAME

file – file message(s) in (an)other folder(s)

SYNOPSIS

file [–src +folder] [msgs] [–link] [–preserve] +folder ... [–nolink]
[–nopreserve] [–file file] [–nofile] [–help]

DESCRIPTION

File moves (*mv*(I)) or links (*ln*(I)) messages from a source folder into one or more destination folders. If you think of a message as a sheet of paper, this operation is not unlike filing the sheet of paper (or copies) in file cabinet folders. When a message is filed, it is linked into the destination folder(s) if possible, and is copied otherwise. As long as the destination folders are all on the same file system, multiple filing causes little storage overhead. This facility provides a good way to cross-file or multiply-index messages. For example, if a message is received from Jones about the ARPA Map Project, the command

```
file cur +jones +Map
```

would allow the message to be found in either of the two folders 'jones' or 'Map'.

The option '–file file' directs *file* to use the specified file as the source message to be filed, rather than a message from a folder.

If a destination folder doesn't exist, *file* will ask if you want to create one. A negative response will abort the file operation.

'–link' preserves the source folder copy of the message (i.e., it does a *ln*(I) rather than a *mv*(I)), whereas, '–nolink' deletes the "filed" messages from the source folder. Normally, when a message is filed, it is assigned the next highest number available in each of the destination folders. Use of the '–preserve' switch will override this message "renaming", but name conflicts may occur, so use this switch cautiously. (See *pick* for more details on message numbering.)

If '–link' is not specified (or '–nolink' is specified), the filed messages will be removed (unlink(II)) from the source folder.

Files

$HOME/. mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder
Folder–Protect:	To set mode when creating a new folder

Defaults

'–src +folder' defaults to the current folder
'msgs' defaults to cur
'–nolink'
'–nopreserve'
'–nofile'

Context

If '−src +folder' is given, it will become the current folder for future MH commands. If neither '−link' nor 'all' are specified, the current message in the source folder will be set to the last message specified; otherwise, the current message won't be changed.

NAME

folder – set/list current folder/message

SYNOPSIS

```
folder [+folder] [msg] [–all] [–fast] [–nofast] [–up] [–down] [–header]
       [–noheader] [–total] [–nototal] [–pack] [–nopack] [–help]

folders <equivalent to 'folder –all'>
```

DESCRIPTION

Since the MH environment is the shell, it is easy to lose track of the current folder from day to day. *Folder* will list the current folder, the number of messages in it, the range of the messages (low-high), and the current message within the folder, and will flag a selection list or extra files if they exist. An example of the output is:

```
    inbox+ has 16 messages ( 3– 22); cur= 5.
```

If a '+folder' and/or 'msg' are specified, they will become the current folder and/or message. An '–all' switch will produce a line for each folder in the user's MH directory, sorted alphabetically. These folders are preceded by the read-only folders, which occur as .mh_profile "cur–" entries. For example,

```
             Folder      # of messages ( range  ); cur msg (other files)
     /fsd/rs/m/tacc has  35 messages (  1– 35); cur= 23.
  /rnd/phyl/Mail/EP has  82 messages (  1–108); cur= 82.
                 ff has   4 messages (  1–  4); cur=  1.
             inbox+ has  16 messages (  3– 22); cur=  5.
                 mh has  76 messages (  1– 76); cur= 70.
              notes has   2 messages (  1–  2); cur=  1.
               ucom has 124 messages (  1–124); cur=  6; (select).

                TOTAL= 339 messages in   7 Folders.
```

The "+" after inbox indicates that it is the current folder. The "(select)" indicates that the folder ucom has a selection list produced by *pick*. If "others" had appeared in parentheses at the right of a line, it would indicate that there are files in the folder directory that don't belong under the MH file naming scheme.

The header is output if either an '–all' or a '–header' switch is specified; it is suppressed by '–noheader'. Also, if *folder* is invoked by a name ending with "s" (e.g., *folders*), '–all' is assumed. A '–total' switch will produce only the summary line.

If '–fast' is given, only the folder name (or names in the case of '–all') will be listed. (This is faster because the folders need not be read.)

The switches '–up' and '–down' change the folder to be the one above or below the current folder. That is, "folder –down" will set the folder to "<current–folder>/select", and if the current folder is a selection-list folder, "folder –up" will set the current folder to the parent of the selection-list. (See *pick* for details on selection-lists.)

The '−pack' switch will compress the message names in a folder, removing holes in message numbering.

Files

$HOME/.mh_profile	The user profile
/bin/ls	To fast-list the folders

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

Defaults

'+folder' defaults to the current folder
'msg' defaults to none
'−nofast'
'−noheader'
'−nototal'
'−nopack'

Context

If '+folder' and/or 'msg' are given, they will become the current folder and/or message.

NAME

forw – forward messages

SYNOPSIS

```
forw [+folder] [msgs] [–editor editor] [–form formfile] [–annotate]
        [–noannotate] [–inplace]  [–noinplace] [–help]
```

DESCRIPTION

Forw may be used to prepare a message containing other messages. It constructs the new message from the components file or '–form formfile' (see *comp*), with a body composed of the message(s) to be forwarded. An editor is invoked as in *comp*, and after editing is complete, the user is prompted before the message is sent.

If the '–annotate' switch is given, each message being forwarded will be annotated with the lines

```
Forwarded: ≪date≫
Forwarded: To: names
Forwarded: cc: names
```

where each "To:" and "cc:" list contains as many lines as required. This annotation will be done only if the message is sent directly from *forw*. If the message is not sent immediately from *forw*, "comp –use" may be used in a later session to re-edit and send the constructed message, but the annotations won't take place. The '–inplace' switch permits annotating a message in place in order to preserve its links.

See *comp* for a description of the '–editor' switch.

Files

/etc/mh/components	The message skeleton
or <mh-dir>/components	Rather than the standard skeleton
$HOME/. mh_profile	The user profile
<mh-dir>/draft	The default message file
/usr/bin/send	To send the composed message

Profile Components

Path:	To determine the user's MH directory
Editor:	To override the use of /bin/ned as the default editor
Current-Folder:	To find the default current folder
<lasteditor>–next:	To name an editor to be used after exit from <lasteditor>

Defaults

'+folder' defaults to the current folder
'msgs' defaults to cur
'–editor' defaults to /bin/ned
'–noannotate'
'–noinplace'

Context

If a +folder is specified, it will become the current folder, and the current message will be set to the first message being forwarded.

NAME

inc – incorporate new mail

SYNOPSIS

inc [+folder] [–audit audit-file] [–help]

DESCRIPTION

Inc incorporates mail from the user's incoming mail drop (. mail) into an MH folder. If '+folder' isn't specified, the folder named "inbox" in the user's MH directory will be used. The new messages being incorporated are assigned numbers starting with the next highest number in the folder. If the specified (or default) folder doesn't exist, the user will be queried prior to its creation. As the messages are processed, a *scan* listing of the new mail is produced.

If the user's profile contains a "Msg–Protect: nnn" entry, it will be used as the protection on the newly created messages, otherwise the MH default of 664 will be used. During all operations on messages, this initially assigned protection will be preserved for each message, so *chmod*(1) may be used to set a protection on an individual message, and its protection will be preserved thereafter.

If the switch '–audit audit-file' is specified (usually as a default switch in the profile), then *inc* will append a header line and a line per message to the end of the specified audit-file with the format:

```
≪inc≫ date
        <scan line for first message>
        <scan line for second message>
                        <etc.>
```

This is useful for keeping track of volume and source of incoming mail. Eventually, *repl*, *forw*, *comp*, and *dist* may also produce audits to this (or another) file, perhaps with "Message-Id:" information to keep an exact correspondence history. "Audit-file" will be in the user's MH directory unless a full path is specified.

Inc will incorporate even illegally formatted messages into the user's MH folder, inserting a blank line prior to the offending component and printing a comment identifying the bad message.

In all cases, the . mail file will be zeroed.

Files

$HOME/. mh_profile	The user profile
$HOME/. mail	The user's mail drop
<mh-dir>/audit-file	Audit trace file (optional)

Profile Components

Path:	To determine the user's MH directory
Folder–Protect:	For protection on new folders
Msg–Protect:	For protection on new messages

Defaults

'+folder' defaults to "inbox"

Context

The folder into which the message is being incorporated will become the current folder, and the first message incorporated will be the current message. This leaves the context ready for a *show* of the first new message.

NAME

next – show the next message

SYNOPSIS

next [+folder] [–switches for *l*] [–help]

DESCRIPTION

Next performs a *show* on the next message in the specified (or current) folder. Like *show*, it passes any switches on to the program *l*, which is called to list the message. This command is exactly equivalent to "show next".

Files

$HOME/. mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

Defaults

Context

If a folder is specified, it will become the current folder, and the message that is shown (i.e., the next message in sequence) will become the current message.

NAME

pick – select messages by content

SYNOPSIS

```
pick ⎧ –cc          ⎫ [–src +folder] [msgs] [–help] [–scan] [–noscan]
     ⎪ –date        ⎪   [–show] [–noshow] [–nofile] [–nokeep]
     ⎪ –from        ⎪
     ⎨ –search      ⎬ pattern
     ⎪ –subject     ⎪
     ⎪ –to          ⎪ [–file [–preserve] [–link] +folder ... [–nopreserve] [–nolin
     ⎩ ––component  ⎭ [–keep [–stay] [–nostay] [+folder ...] ]
```

typically:

```
pick –from jones –scan
pick –to holloway
pick –subject ned –scan –keep
```

DESCRIPTION

Pick searches messages within a folder for the specified contents, then performs several operations on the selected messages.

A modified *grep*(I) is used to perform the searching, so the full regular expression (see *ed*(I)) facility is available within 'pattern'. With '–search', pattern is used directly, and with the others, the grep pattern constructed is:

"component:. *pattern"

This means that the pattern specified for a '–search' will be found everywhere in the message, including the header and the body, while the other search requests are limited to the single specified component. The expression '––component pattern' is a shorthand for specifying '–search "component:. *pattern" '; it is used to pick a component not in the set [cc date from subject to]. An example is "pick ––reply–to pooh –show".

Searching is performed on a per-line basis. Within the header of the message, each component is treated as one long line, but in the body, each line is separate. Lower-case letters in the search pattern will match either lower or upper case in the message, while upper case will match only upper case.

Once the search has been performed, the selected messages are scanned (see *scan*) if the '–scan' switch is given, and then they are shown (see *show*) if the '–show' switch is given. After these two operations, the file operations (if requested) are performed.

The '–file' switch operates exactly like the *file* command, with the same meaning for the '–preserve' and '–link' switches.

The '–keep' switch is similar to '–file', but it produces a folder that is a subfolder of the folder being searched and defines it as the current folder (unless the '–stay' flag is used). This subfolder contains the messages which matched the search criteria. All of the MH commands may be used with the sub-folder as the current folder. This gives the user considerable power in dealing with subsets of messages in a folder.

The messages in a folder produced by '−keep' will always have the same numbers as they have in the source folder (i.e., the '−preserve' switch is automatic). This way, the message numbers are consistent with the folder from which the messages were selected. Messages are not removed from the source folder (i.e., the '−link' switch is assumed). If a '+folder' is not specified, the standard name "select" will be used. (This is the meaning of "(select)" when it appears in the output of the *folder* command.) If '+folder' arguments are given to '−keep', they will be used rather than "select" for the names of the subfolders. This allows for several subfolders to be maintained concurrently.

When a '−keep' is performed, the subfolder becomes the current folder. This can be overridden by use of the '−stay' switch.

Here's an example:

```
 1 % folder +inbox
 2        inbox+ has  16 messages (  3– 22); cur=  3.
 3 % pick −from dcrocker
 4 6 hits.
 5 [+inbox/select now current]
 6 % folder
 7  inbox/select+ has   6 messages (  3– 16); cur=  3.
 8 % scan
 9   3+  6/20  Dcrocker       Re: ned file update issue...
10   6   6/23  Dcrocker       removal of files from /tm...
11   8   6/27  Dcrocker       Problems with the new ned...
12  13   6/28  dcrocker       newest nned  <<I would ap...
13  15   7/ 5  Dcrocker       nned  <<Last week I asked...
14  16   7/ 5  dcrocker       message id format  <<I re...
15 % show all | print
16   [produce a full listing of this set of messages on the line printer.]
17 % folder −up
18         inbox+ has  16 messages (  3– 22); cur=  3; (select).
19 % folder −down
20  inbox/select+ has   6 messages (  3– 16); cur=  3.
21 % rmf
22 [+inbox now current]
23 % folder
24         inbox+ has  16 messages (  3– 22); cur=  3.
```

This is a rather lengthy example, but it shows the power of the MH package. In item 1, the current folder is set to inbox. In 3, all of the messages from dcrocker are found in inbox and linked into the folder "inbox/select". (Since no action switch is specified, '−keep' is assumed.) Items 6 and 7 show that this subfolder is now the current folder. Items 8 through 14 are a *scan* of the selected messages (note that they are all from dcrocker and are all in upper and lower case). Item 15 lists all of the messages to the high-speed printer. Item 17 directs *folder* to set the current folder to the parent of the selection-list folder, which is now current. Item 18 shows that this has been done. Item 19 resets the current folder to the selection list, and 21 removes the selection-list folder and resets the current folder to the parent folder, as shown in 22 and 23.

Files

$HOME/. mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory
Folder–Protect:	For protection on new folders
Current-Folder:	To find the default current folder

Defaults

'–src +folder' defaults to current
'msgs' defaults to all
'–keep +select' is the default if no '–scan', '–show', or '–file' is specified

Context

If a '–src +folder' is specified, it will become the current folder, unless a '–keep' with 0 or 1 folder arguments makes the selection-list subfolder the current folder. Each selection-list folder will have its current message set to the first of the messages linked into it unless the selection list already existed, in which case the current message won't be changed.

NAME

prev – show the previous message

SYNOPSIS

prev [+folder] [–switches for *l*] [–help]

DESCRIPTION

Prev performs a *show* on the previous message in the specified (or current) folder. Like *show*, it passes any switches on to the program *l*, which is called to list the message. This command is exactly equivalent to "show prev".

Files

$HOME/. mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

Defaults

Context

If a folder is specified, it will become current, and the message that is shown (i.e., the previous message in sequence) will become the current message.

NAME

prompter – prompting editor front end

SYNOPSIS

This program is not called directly but takes the place of an editor and acts as an editor front end.

prompter [–erase chr] [–kill chr] [–help]

DESCRIPTION

Prompter is an editor which allows rapid composition of messages. It is particularly useful to network and low-speed (less than 2400 baud) users of MH. It is an MH program in that it can have its own profile entry with switches, but it can't be invoked directly as all other MH commands can; it is an editor in that it is invoked by an "–editor prompter" switch or by the profile entry "Editor: prompter", but functionally it is merely a text-collector and not a true editor.

Prompter expects to be called from *comp*, *repl*, *dist*, or *forw*, with a draft file as an argument. For example, "comp –editor prompter" will call *prompter* with the file "draft" already set up with blank components. For each blank component it finds in the draft, it prompts the user and accepts a response. A <RETURN> will cause the whole component to be left out. A "\" preceding a <RETURN> will continue the response on the next line, allowing for multiline components.

Any component that is non-blank will be copied and echoed to the terminal.

The start of the message body is prompted by a line of dashes. If the body is non-blank, the prompt is "--------Enter additional text". Message-body typing is terminated with a <CTRL-D> (or <OPEN>). Control is returned to the calling program, where the user is asked "What now?". See *comp* for the valid options.

The line editing characters for kill and erase may be specified by the user via the arguments "–kill chr" and "–erase chr", where chr may be a character; or "\nnn", where nnn is the octal value for the character. (Again, these may come from the default switches specified in the user's profile.)

A <DEL> during message-body typing is equivalent to <CTRL-D> for compatibility with NED. A <DEL> during component typing will abort the command that invoked *prompter*.

Files

None

Profile Components

prompter-next:	To name the editor to be used on exit from *prompter*

Defaults

Context

None

NAME

repl − reply to a message

SYNOPSIS

```
repl [+folder] [msg] [−editor editor] [−inplace] [−annotate] [−help]
        [−noinplace] [−noannotate]
```

DESCRIPTION

Repl aids a user in producing a reply to an existing message. In its simplest form (with no arguments), it will set up a message-form skeleton in reply to the current message in the current folder, invoke the editor, and send the composed message if so directed. The composed message is constructed as follows:

```
To: <Reply-To> or <From>
cc: <cc>, <To>
Subject: Re: <Subject>
In-reply-to: Your message of <Date>
             <Message-Id>
```

where field names enclosed in angle brackets (< >) indicate the contents of the named field from the message to which the reply is being made. Once the skeleton is constructed, an editor is invoked (as in *comp*, *dist*, and *forw*). While in the editor, the message being replied to is available through a link named "@". In NED, this means the replied-to message may be "used" with "use @", or put in a window by "window @".

As in *comp*, *dist*, and *forw*, the user will be queried before the message is sent. If '−annotate' is specified, the replied-to message will be annotated with the single line

Replied: ≪Date≫.

The command "comp −use" may be used to pick up interrupted editing, as in *dist* and *forw*; the '−inplace' switch annotates the message in place, so that all folders with links to it will see the annotation.

Files

$HOME/.mh_profile	The user profile
<mh-dir>/draft	The constructed message file
/usr/bin/send	To send the composed message

Profile Components

Path:	To determine the user's MH directory
Editor:	To override the use of /bin/ned as the default editor
Current-Folder:	To find the default current folder

Defaults

'+folder' defaults to current
'msgs' defaults to cur
'−editor' defaults to /bin/ned
'−noannotate'
'−noinplace'

Context

If a '+folder' is specified, it will become the current folder, and the current message will be set to the replied-to message.

NAME

rmf – remove folder

SYNOPSIS

rmf [+folder] [–help]

DESCRIPTION

Rmf removes all of the files (messages) within the specified (or default) folder, and then removes the directory (folder). If there are any files within the folder which are not a part of MH, they will *not* be removed, and an error will be produced. If the folder is given explicitly or the current folder is a subfolder (i.e., a selection list from *pick*), it will be removed without confirmation. If no argument is specified and the current folder is not a selection-list folder, the user will be asked for confirmation.

Rmf irreversibly deletes messages that don't have other links, so use it with caution.

If the folder being removed is a subfolder, the parent folder will become the new current folder, and *rmf* will produce a message telling the user this has happened. This provides an easy mechanism for selecting a set of messages, operating on the list, then removing the list and returning to the current folder from which the list was extracted. (See the example under *pick*.)

The files that *rmf* will delete are cur, any file beginning with a comma, and files with purely numeric names. All others will produce error messages.

Rmf of a read-only folder will delete the "cur–" entry from the profile without affecting the folder itself.

Files

$HOME/.mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

Defaults

'+folder' defaults to current, usually with confirmation

Context

Rmf will set the current folder to the parent folder if a subfolder is removed; or if the current folder is removed, it will make "inbox" current. Otherwise, it doesn't change the current folder or message.

\ME
rmm – remove messages

NOPSIS
rmm [+folder] [msgs] [–help]

ESCRIPTION

Rmm removes the specified messages by renaming the message files with preceding commas. (This is the Rand-UNIX backup file convention.)

The current message is not changed by *rmm*, so a *next* will advance to the next message in the folder as expected.

iles

$HOME/. mh_profile	The user profile

'rofile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

)efaults
'+folder' defaults to current
'msgs' defaults to cur

Context
If a folder is given, it will become current.

NAME

scan – produce a one-line-per-message scan listing

SYNOPSIS

scan [+folder] [msgs] [–ff] [–header] [–help] [–noff] [–noheader]

DESCRIPTION

Scan produces a one-line-per-message listing of the specified messages. Each *scan* line contains the message number (name), the date, the "From" field, the "Subject" field, and, if room allows, some of the body of the message. For example:

```
#     Date     From          Subject   [≪Body]
15+   7/ 5     Dcrocker      nned  ≪Last week I asked some of
16–   7/ 5     dcrocker      message id format  ≪I recommend
18    7/ 6     Obrien        Re: Exit status from mkdir
19    7/ 7     Obrien        "scan" listing format in MH
```

The '+' on message 15 indicates that it is the current message. The '–' on message 16 indicates that it has been replied to, as indicated by a "Replied:" component produced by an '–annotate' switch to the *repl* command.

If there is sufficient room left on the *scan* line after the subject, the line will be filled with text from the body, preceded by ≪. *Scan* actually reads each of the specified messages and parses them to extract the desired fields. During parsing, appropriate error messages will be produced if there are format errors in any of the messages.

The '–header' switch produces a header line prior to the *scan* listing, and the '–ff' switch will cause a form feed to be output at the end of the *scan* listing. See Appendix D.

Files

$HOME/. mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

Defaults

Defaults:
'+folder' defaults to current
'msgs' defaults to all
'–noff'
'–noheader'

Context

If a folder is given, it will become current. The current message is unaffected.

NAME

send – send a message

SYNOPSIS

send [file] [–draft] [–verbose] [–format] [–msgid] [–help] [–noverbose]
[–noformat] [–nomsgid]

DESCRIPTION

Send will cause the specified file (default <mh-dir>/draft) to be delivered to each of the addresses in the "To:", "cc:", and "Bcc:" fields of the message. If '–verbose' is specified, *send;* will monitor the delivery of local and net mail. *Send* with no argument will query whether the draft is the intended file, whereas '–draft' will suppress this question. Once the message has been mailed (or queued) successfully, the file will be renamed with a leading comma, which allows it to be retreived until the next draft message is sent. If there are errors in the formatting of the message, *send;* will abort with a (hopefully) helpful error message.

If a "Bcc:" field is encountered, its addresses will be used for delivery, but the "Bcc:" field itself will be deleted from all copies of the outgoing message.

Prior to sending the message, the fields "From: user", and "Date: now" will be prepended to the message. If '–msgid' is specified, then a "Message-Id:" field will also be added to the message. If the message already contains a "From:" field, then a "Sender: user" field will be added instead. (An already existing "Sender:" field will be deleted from the message.)

If the user doesn't specify '–noformat', each of the entries in the "To:" and "cc:" fields will be replaced with "standard" format entries. This standard format is designed to be usable by all of the message handlers on the various systems around the ARPANET.

If an "Fcc: folder" is encountered, the message will be copied to the specified folder in the format in which it will appear to any receivers of the message. That is, it will have the prepended fields and field reformatting.

If a "Distribute-To:" field is encountered, the message is handled as a redistribution message (see *dist* for details), with "Distribution-Date: now" and "Distribution-From: user" added.

Files

$HOME/.mh_profile	The user profile

Profile Components

Path:	To determine the user's MH directory

Defaults

'file' defaults to draft
'–noverbose'
'–format'
'–nomsgid'

Context

Send has no effect on the current message or folder.

NAME

show – show (list) messages

SYNOPSIS

show [+folder] [msgs] [–pr] [–nopr] [–draft] [–help] [*l* or *pr* switches]

DESCRIPTION

Show lists each of the specified messages to the standard output (typically, the terminal). The messages are listed exactly as they are, with no reformatting. A program called *l* is invoked to do the listing, and any switches not recognized by *show* are passed along to *l*.

If no "msgs" are specified, the current message is used. If more than one message is specified, *l* will prompt for a <return> prior to listing each message.

l will list each message, a page at a time. When the end of page is reached, *l* will ring the bell and wait for a <RETURN> or <CTRL-D>. If a <return> is entered, *l* will clear the screen before listing the next page, whereas <CTRL-D> will not. The switches to *l* are '–p#' to indicate the page length in lines, and '–w#' to indicate the width of the page in characters.

If the standard output is not a terminal, no queries are made, and each file is listed with a one-line header and two lines of separation.

If '–pr' is specified, then *pr*(I) will be invoked rather than *l*, and the switches (other than '–draft') will be passed along. "Show –draft" will list the file <mh-dir>/draft if it exists.

Files

$HOME/. mh_profile	The user profile
/bin/l	Screen-at-a-time list program
/bin/pr	*pr*(I)

Profile Components

Path:	To determine the user's MH directory
Current-Folder:	To find the default current folder

Defaults

'+folder' defaults to current
'msgs' defaults to cur
'–nopr'

Context

If a folder is given, it will become the current message. The last message listed will become the current message.

Appendix A
COMMAND SUMMARY[3]

```
comp [−editor editor] [−form formfile] [file] [−use] [−nouse] [−help]

dist [+folder] [msg] [−form formfile] [−editor editor] [−annotate] [−noannotate]
        [−inplace] [−noinplace] [−help]

file [−src +folder] [msgs] [−link] [−preserve] +folder ...  [−nolink] [−nopreserve]
        [−file file] [−nofile] [−help]

folder [+folder] [msg] [−all] [−fast] [−nofast] [−up] [−down] [−header] [−noheader]
        [−total] [−nototal] [−pack] [−nopack] [−help]

forw [+folder] [msgs] [−editor editor] [−form formfile] [−annotate] [−noannotate]
        [−inplace]  [−noinplace] [−help]

inc [+folder] [−audit audit-file] [−help]

next [+folder] [−switches for l] [−help]

pick ⎧ −cc               ⎫ [−src +folder] [msgs] [−help] [−scan] [−noscan]
     ⎪ −date             ⎪  [−show] [−noshow] [−nofile] [−nokeep]
     ⎪ −from             ⎪
     ⎨ −search           ⎬ pattern
     ⎪ −subject          ⎪
     ⎪ −to               ⎪ [−file [−preserve] [−link] +folder ... [−nopreserve] [−nolink] ]
     ⎩ −−component       ⎭ [−keep [−stay] [−nostay] [+folder ...] ]

prev [+folder] [−switches for l] [−help]

prompter [−erase chr] [−kill chr] [−help]

repl [+folder] [msg] [−editor editor] [−inplace] [−annotate] [−help] [−noinplace]
[−noannotate]

rmf [+folder] [−help]

rmm [+folder] [msgs] [−help]

scan [+folder] [msgs] [−ff] [−header] [−help] [−noff] [−noheader]

send [file] [−draft] [−verbose] [−format] [−msgid] [−help]  [−noverbose] [−noformat]
[−nomsgid]

show [+folder] [msgs] [−pr] [−nopr] [−draft] [−help] [l or pr switches]
```

[3]All commands accept a −help switch.

Appendix B
MESSAGE FORMAT

This section paraphrases the format of ARPANET text messages given in Ref. 6.

ASSUMPTIONS

(1) Messages are expected to consist of lines of text. Graphics and binary data are not handled.

(2) No data compression is accepted. All text is clear ASCII 7-bit data.

LAYOUT

A general "memo" framework is used. A message consists of a block of information in a rigid format, followed by general text with no specified format. The rigidly formatted first part of a message is called the header, and the free-format portion is called the body. The header must always exist, but the body is optional.

THE HEADER

Each header item can be viewed as a single logical line of ASCII characters. If the text of a header item extends across several real lines, the continuation lines are indicated by leading spaces or tabs.

Each header item is called a component and is composed of a keyword or name, along with associated text. The keyword begins at the left margin, may contain spaces or tabs, may not exceed 63 characters, and is terminated by a colon (:). Certain components (as identified by their keywords) must follow rigidly defined formats in their text portions.

The text for most formatted components (e.g., "Date:" and "Message-Id:") is produced automatically. The only ones entered by the user are address fields such as "To:", "cc:", etc. ARPA addresses are assigned mailbox names and host computer specifications. The rough format is "mailbox at host", such as "Borden at Rand-Unix". Multiple addresses are separated by commas. A missing host is assumed to be the local host.

THE BODY

A blank line signals that all following text up to the end of the file is the body. (A blank line is defined as a pair of <end-of-line> characters with *no* characters in between.) No formatting is expected or enforced within the body.

Within MH, a line consisting of dashes is accepted as the header delimiter. This is a cosmetic feature applying only to locally composed mail.

Appendix C

MESSAGE NAME BNF

```
msgs            :=    msgspec            |
                      msgs msgspec

msgspec         :=    msg                |
                      msg-range          |
                      msg-sequence

msg             :=    msg-name           |
                      <number>

msg-name        :=    "first"            |
                      "last"             |
                      "cur"              |
                      "."                |
                      "next"             |
                      "prev"

msg-range       :=    msg"-"msg          |
                      "all"

msg-sequence    :=    msg":"signed-number

signed-number   :=    "+"<number>        |
                      "—"<number>        |
                      <number>
```

Where <number> is a decimal number in the range 1 to 999.

Msg-range specifies all of the messages in the given range and must not be empty.

Msg-sequence specifies up to <number> of messages, beginning with "msg" (in the case of first, cur, next, or <number>), or ending with "msg" (in the case of prev or last). +<number> forces "starting with msg", and −<number> forces "ending with number". In all cases, "msg" must exist.

Appendix D
EXAMPLE OF SHELL COMMANDS

UNIX commands may be mixed with MH commands to obtain additional functions. These may be prepared as files (known as shell command files or shell scripts). The following example is a useful function that illustrate the possibilities. Other functions, such as copying, deleting, renaming, etc., can be achieved in a similar fashion.

HARDCOPY

The command:

```
(scan −ff −header; show all −pr −f) | print
```

produces a scan listing of the current folder, followed by a form feed, followed by a formatted listing of all messages in the folder, one per page. Omitting "−pr −f" will cause the messages to be concatenated, separated by a one-line header and two blank lines.

You can create variations on this theme, using *pick*.

REFERENCES

1. Crocker, D. H., J. J. Vittal, K. T. Pogran, and D. A. Henderson, Jr., "Standard for the Format of ARPA Network Test Messages," *Arpanet Request for Comments*, No. 733, Network Information Center 41952, Augmentation Research Center, Stanford Research Institute, November 1977.

2. Thompson, K., and D. M. Ritchie, "The UNIX Time-sharing System," *Communications of the ACM*, Vol. 17, July 1974, pp. 365-375.

3. McCauley, E. J., and P. J. Drongowski, "KSOS—The Design of a Secure Operating System," *AFIPS Conference Proceedings*, National Computer Conference, Vol. 48, 1979, pp. 345-353.

4. Crocker, David H., *Framework and Functions of the "MS" Personal Message System*, The Rand Corporation, R-2134-ARPA, December 1977.

5. Thompson, K., and D. M. Ritchie, *UNIX Programmer's Manual*, 6th ed., Western Electric Company, May 1975 (available only to UNIX licensees).

6. Bilofsky, Walter, *The CRT Text Editor NED—Introduction and Reference Manual*, The Rand Corporation, R-2176-ARPA, December 1977.