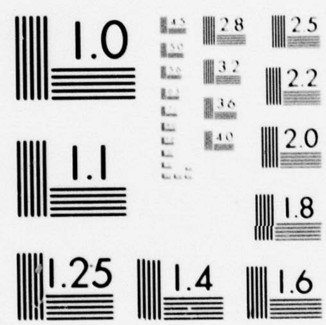
AD-A066 994 WHARTON SCHOOL PHILADELPHIA PA DEPT OF DECISION SCIENCES F/G 9/2
Q A COMMUNICATIONS QUERY LANGUAGE FOR SEED.(U)
MAY 78 J HAYWARD, R SANGAL, O P BUNEMAN N00014-75-C-0462
UNCLASSIFIED 78-05-02 NL

| OF |

AD
A066994

END
DATE
FILMED
6-79
DDC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

arton

rtment of Decision Sciences

LEVEL

(12)

University of
Pennsylvania
Philadelphia PA 19104

Q - A Communications Query
Language for SEED

Jonathan Hayward
Rajeev Sangal
Peter Buneman

78-05-02

Department of Decision Sciences
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

May 1978

Research supported in part by the Office of Naval Research
under Contract N00014-75-C-0462.

Introduction

With a few notable exceptions, query languages have been designed to enable people to communicate with database systems. The advent of computer networking has made increasingly important the task of designing languages with which another program may talk to a database system. The DATACOMPUTER [1] supports a query language which, while it may be used directly, was designed to be generated by other programs. The DATACOMPUTER maintains a quasi-relational database system with no direct linking between records. Q is an attempt to do the same thing for a network database: specifically SEED, which is a CODASYL like system developed at the Wharton School. In designing such a language there are two main goals: first the language should be as terse and as powerful as possible in order to reduce the message traffic in both directions when a query is sent and answered; second to design a good message passing protocol so that synchronization between programs is possible. In the next section these goals are described more fully together with more details of the operating environment for which Q was designed.

The problem of program-to-program communication

Generally, computer systems have concentrated on having one or two languages (such as FORTRAN and COBOL) which are standard on on a given system. The standardization has led to a number of support packages written in FORTRAN or COBOL which can be loaded only with other FORTRAN or COBOL programs. SEED [2] is such a system. As understanding of programming languages has continued, one finds that special purpose languages have been developed that can be used for for production (as BLISS) or research (as POP10). However, support programs written in FORTRAN or COBOL cannot generally be loaded with languages such as POP10 or LISP.

Development of network communications has worsened the situation. Until network communications became more important, the concept of machine independence was important to allow transfer of programs from one system to another more easily. FORTRAN and COBOL were the standard languages for machine independence. Even after network communications became important, one of the main uses was to transfer programs from one machine to another (FTP on the ARPANET for example) and machine independence was still important. However, now, computer networking is starting to emphasize the segmentation of program systems into various "tools" that are available at the sites on a network. The possibility of using many tools on different hosts means that a program cannot be loaded into one contiguous section of memory. Consequently, the concept of program independence is not as important; linking the independent operation of separate tasks becomes the main goal.

Both of these reasons lead to a concept in programming that is not fully understood: that of breaking apart a large task into smaller asynchronous components which synchronize activity by sending messages between themselves.

We have been faced with several research projects at the University of Pennsylvania which require a database to behave as a separate asynchronous component of a larger system. DBLOOK of the SEED database system has been used to accomplish asynchronous operation in the past. Several problems become apparent with DBLOOK when it is used as an asynchronous task serving another task. DBLOOK is fairly intelligent, and to a person using DBLOOK, the results are satisfying. DBLOOK carries on an "implied" conversation. It lets the user figure out what it is reporting and requesting. For humans, the brevity of the output and input is an excellent feature, since it cuts out the information the user already understands. Programs which use DBLOOK do not have the same intelligence as humans, and have a much harder time carrying on the conversation. For instance, when DBLOOK displays a record, it is not explicitly clear where all the fields begin and end.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 78-05-02	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Q - A Communications Query Language for SEED	5. TYPE OF REPORT & PERIOD COVERED: Technical Report, Apr 78 - Mar 79
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Jonathan Hayward, Rajeev Sangal, O. Peter Buneman	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0462
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Decision Sciences, University of Pennsylvania, Philadelphia, PA 19104	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: Task NR049-272
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research	12. REPORT DATE: May 1978
	13. NUMBER OF PAGES: 22
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

(14) (6) (10) (15) (11) May 78 (12) 25p.

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

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

Distribution unlimited

18. SUPPLEMENTARY NOTES

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

query languages, computer networking, datacomputer, network database, communication query system

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

The advent of computer networking has made increasingly important the task of designing languages with which another program may talk to a database system. The DATACOMPUTER supports a query language which was designed to be generated by other programs. It maintains a quasi-relational database system with no direct linking between records. Q is an attempt to do the same thing for a network database: specifically SEED, which is a CODASYL like system developed at the Wharton School.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

408957

DBLOOK also has some limitations on its capability which make some queries difficult to perform. DBLOOK cannot give back values which are the result of computations on fields in the database. In addition, the CODASYL DML functions are not very appropriate once one has decided to access a database through a separate task. The DML definition was based on the ability to access a global area containing all the records easily (the UWA).

It is the intention of this project to try to overcome some of these problems by:

1 - designing a query language which is concise, allows complicated queries to be processed simply.

2 - designing a control structure for executing the query which allows simple synchronization between the communicating tasks.

3 - reporting output in formats which contain all the information for a program to easily ascertain what the output means.

Of course all of these criteria are quite vague. Number 1 is especially vague, since that is the object of any query language. In considering what other criteria we might apply, we decided to adopt the following:

4 - the language should allow any query to be processed which does not require storage that increases more than linearly with the size of the query.

5 - the language should not allow any explicit control

structures, such as do loops or conditional branching, yet should be able to selectively process portions of the database.

Statement 4 effectively rules out any processing which would require sorting or merging.

Statement 5 eliminates the need for any functions such as "find first" or "find next". In the limited scope of a query language, it would be burdensome to require an explicit "find" for every record, since a "find" is generally necessary. In addition, we arbitrarily decided to limit ourselves to exploring the database by defined set relationships.

Query Language

We look at the database as a hierarchy by starting at one particular point in the database. Then, the particular fields that one wishes to access can be specified. Items called computed fields can be defined that are computed on the basis of other fields. Computed fields can be given a name for later reference, or used to restrict further processing. The functions that are allowed in computed fields are PLUS, MINUS, MULTIPLY, DIVIDE, EQUAL, GT, LT, GE, LE, AND, OR, NOT, and INT. Two more functions are provided which "reduce" portions of the tree. They are SUM and COUNT.

The query language will be explained with reference to the following database structure:

```
      STUDENT                          CLASSES
      --------------                   -------------------
      :studname:...:                   :classname:.......:
      --------------                   -------------------
             \                               /
              \                             /
               \                           /  ROSTER
                \  TAKES                  /
                 \                       /
                  \  ENROLLMENT         /
                   ----------------
                   :grade:........:
                   ----------------
```

The language is designed around the concept of streams. We use the term "stream" to denote a generator for a sequence of objects (records, field values, other streams). The term is used in preference to "set", which denotes a specific data structure in the database and "list" which denotes a specific in-core database. A stream is effectively a procedure for generating a specified sequence of objects. See Burge [3] for a detailed explanation of this concept. One creates a stream by openning a set of parentheses preceded by a set or record name. For instance:

-STUDENT(...)

creates a stream of students. Operators can be applied to a stream to define elements of the stream:

-STUDENT(NAME:STUDNAME)

or to create a stream of streams:

```
-STUDENT(!TAKES( ... ))
```

In the former case, a stream of student names is created. In the latter case, a stream of enrollments for student is created. When defining a stream of streams, a "!" is used to indicate that the stream owns a set of items represented by the inner stream. A "^" indicates that the outer stream is owned by one item in the inner stream. "!" and "^" allow traversal of the Bachman diagram representing the schema. "-" is used to indicate that the stream that is being defined is simply a set of records, and is not related to any other streams. ("-" can only appear at the outside of an expression).

Items in the schema are referenced by placing the item name in the parentheses. If a name followed by a ":" precedes the item then the name is a user defined name for the item. In the example above. "NAME" is the user defined name for the item "STUDNAME" in the schema.

Once some items have been defined, they can be printed with a "$P". For example:

```
-STUDENT(NAME:STUDNAME, !TAKES(GRADE, ^ROSTER(CLASSNAME,
            $P NAME, $P CLASSNAME, $P GRADE)))
```

will print out the names of students, and the classes they are taking, and the grades they have in the classes.

Suppose we would like to know how many classes the students are taking. Then we could say:

```
-STUDENT(NAME:STUDNAME, !TAKES(GRADE) NUM:COUNT GRADE,
```

```
$P NAME, $P GRADE)
```

The function count produce a count of the number of items in the stream given as an argument. Suppose we wish to get the average grade of all the students:

```
-STUDENT(NAME:STUDNAME, !TAKES(GRADE) S1:SUM GRADE,
  N1:COUNT GRADE, AVE:DIV S1 N1, $P NAME, $P AVE)
```

The function divide will divide S1 by N1 to produce the average grade. But, a problem could occur with the query above if a student is not taking any courses. A division by zero would occur. To eliminate certain portions of a stream, a restriction can be introduced:

```
-STUDENT(NAME:STUDNAME, !TAKES(GRADE) N1:COUNT GRADE)
 $R GT N1 0 (S1:COUNT GRADE, $P NAME, AVE:DIV S1 N1,
       $P AVE)
```

The function after the "$R" is used to restrict any further processing of streams that contain no grade records. "$R" might also be used to look at the record of a particular student:

```
-STUDENT(NAME:STUDNAME)$R EQUAL NAME 'MARTIN MEYERSON'
      • (!TAKES(GRADE, ^ROSTER(CLASSNAME),
            $P CLASSNAME, $P GRADE))
```

The control structure for executing queries is very straightforward. One has the option of entering a query, opening the database, processing a query, and aborting execution. If an error should occur, the system waits for a specific response to resynchronize itself with the controlling task. The system also informs the controlling task when the processing of a request has started and stopped.

The responses given back are straightforward. They fall into 4 categories:

1 - Errors

2 - Data

3 - Synchronization

4 - Resynchronization request

System Operation

To start the query system, one simply types "R Q" at the monitor level. When the system is ready, it will type "READY". At this point, commands can be entered. Legal commands are DBOPEN, PROGRA, RUN, VERIFY, DBCLOS, and EXIT.

DBOPEN will open a database. The name of the database must follow the DBOPEN command as the 7th through 12th characters on the line. The privacy key must start in the 14th character position.

PROGRA will allow the lines following it to be entered as a query. Syntax is checked as the query is entered. To end the entry of the query, an "#" is typed as the first character on a line. To abort, an "@" is typed as the first character on a line. If a filename is specified after the PROGRA, then the file is used as the source of input for the query.

RUN will process the query. First the query is checked against the schema to make sure that all the items and sets are correctly defined. Then the database is accessed to process the query.

VERIFY will check a query against the schema.

DBCLOS closes the database.

EXIT stops the execution of the system.

The system responds to commands with the following keywords in the first 6 character positions of a line: START, DATA, DONE, SCHERR, RUNERR, SYSERR, CMDERR, CLRACK, ABOK, ENTER, SYNERR, FILE.

START indicates that the processing of the request has started.

DATA indicates that the rest of the line contains output from a print request in the query

ENTER indicates that the query system is waiting for a line of the query to be typed. ENTER will appear if the query is not input from a file.

FILE will appear if a query is input from a file. The remainder of the line contains a line of data as read from the file.

SYNERR indicates that an error in the syntax of the query exists.

DONE indicates that the processing of the query is complete.

SCHERR indicates that the query is in conflict with the schema.

RUNERR indicates that a run error has occurred.

SYSERR indicates that an error has been detected in the system's operation

CMDERR indicates that the processing of a command is incomplete because of an error.

Any SCHERR will automatically cause a CMDERR at the end of the VERIFY process. After a CMDERR, SYSERR, RUNERR, or SYNERR the word "CLEAR" must sent back to the query system to indicate acknowledgement of the error. The query system acknowledges with "CLRACK".

To abort processing of a "RUN", or "PROGRA", an "@" can be typed. The system will respond with "ABOK" when it recognizes the abort request. (The system only checks for abort before printing a DATA statement).

Further Developments

The idea that record selectors could be generalized to work over streams (or streams of streams) of records and that the usual boolean operators and arithmetic operators could be similarly extended originally led us to believe that we could develop an "APL for databases". The semantics of the language have taken us some way towards this goal; however the syntax is still lacking. One of the main problems is that one needs to be able to define new record types and their selectors in the middle of a query. For example, in the student - course database described earlier there is a query in which one constructs a stream of triples: (student number of courses taken, total grade). We have no very good method of labelling this stream for future use in the query. A second difficulty is the standard problem in applicative programming: that of giving the same argument to two different procedures without using an assignment. The latter problem can be solved by the use of combinators [4] or by the syntax suggested by Friedman and Wise [5], but we know of no practical language which exploits these.

The other omission of Q is that there is no provision for performing updates.There are some straightforward methods of specifying an update and these should be added. To give full power to Q, one also needs to implement the operators which take the union, join etc. of streams. Such instructions may be computationally expensive and it is not clear that the database should be charged with performing them.

In spite of these drawbacks it is gratifying to see that Q is being used for allowing a LISP program to access a database. The programmers, who have to learn the language, develop the ability to construct monstrous and opaque "one-liners" and in a rather limited sense, our ambition to develop an APL for databases has been fulfilled.

References

1. DATACOMPUTER Usérs manual. Computer Corporation of America. 1976.

2. Gerritsen, R. et al. Seed Reference Manual. Decision Sciences Working Paper, University of Pennsylvania, 1977

3. Burge, W.H. Recursive Programming Techniques, Adison Wesley, 1974

4. Hindley, J.R., Lercher, B., Seldin, J.P. Introduction to Combinatory Logic, Cambridge, CUP 1972

5. Friedman, D.P. and Wise, D. "Cons should not evaluate its arguments",Technical report 24, Dept of Computer Science, Indiana University (1974).

Appendix A - Sample Execution

The following example is taken from the CTEC data base describing a naval scenario. The portion of the database which we are exploring involves ships and radar, and the confluency between them.

```
            R18SHIP08 ship record      radar record
            ----------------------    --------------
            :I18NAME08 shipname   :   :I37NAME12     :
            ----------------------    : radar name   :
                                      --------------
                   /       \                /
                  /         \              /
      S1878      /           \ S1822      / S3722
                /             \          /
  R78SPNAM08/ shipnames        \        /radar ship confluency
  --------------------        ----------------------------------
  :I78NAME08 shipname :       :I22SPRDR08 number of radar on   :
  --------------------        :      ship                      :
                              ----------------------------------
```

```
.RU Q

READY
DBOPEN  ONRSUB CTEC
START   OF PROCESSING
DONE    QUERY RUNTIME:  0.013 SEED RUNTIME:  1.624
PROGRA  Q0.DAT
START   OF PROCESSING
FILE    -R78SPNAM08(NAME:I78NAME08)$R EQUAL NAME 'CHICAGO'
FILE    (S1878(SHIPNAME:I18NAME08,!S1822(NUMBER:I22SPRDR08,
FILE    S3722(RADARNAME:I37NAME12, $P SHIPNAME, $P RADARNAME,
FILE    $P NUMBER))))
DONE    QUERY RUNTIME:  0.875 SEED RUNTIME:  0.000
RUN
START   OF PROCESSING
DATA    SHIPNAME                        =CHICAGO
DATA    RADARNAME                       =SPS-10
DATA    NUMBER                          =                1
DATA    SHIPNAME                        =CHICAGO
DATA    RADARNAME                       =SPS-30
DATA    NUMBER                          =                1
DATA    SHIPNAME                        =CHICAGO
DATA    RADARNAME                       =SPS-43
DATA    NUMBER                          =                1
```

```
DATA    SHIPNAME                          =CHICAGO
DATA    RADARNAME                         =SPS-48
DATA    NUMBER                            =              1
DATA    SHIPNAME                          =CHICAGO
DATA    RADARNAME                         =SPS-52
DATA    NUMBER                            =              1
DONE    QUERY RUNTIME:  1.878 SEED RUNTIME:  0.998
PROGRA  Q4.DAT
START   OF PROCESSING
FILE    -R18SHIP08(SHIPNAME:I18NAME08,!S1822(N:I22SPRDR08),
FILE    TOTALRAD:SUM N,NTYPE:COUNT N)$R GE NTYPE 2
FILE    ($P SHIPNAME, $P NTYPE, $P TOTALRAD,
FILE    !S1822(NUMBER:I22SPRDR08,S3722
FILE    (RADARNAME:I37NAME12,$P RADARNAME,
FILE    $P NUMBER)))
DONE    QUERY RUNTIME:  0.936 SEED RUNTIME:  0.000
RUN
START   OF PROCESSING
DATA    SHIPNAME                          =DOWNES
DATA    NTYPE                             =              2
DATA    TOTALRAD                          =              2
DATA    RADARNAME                         =SPS-40
DATA    NUMBER                            =              1
DATA    RADARNAME                         =SPS-10
DATA    NUMBER                            =              1
DATA    SHIPNAME                          =TRUETT
DATA    NTYPE                             =              2
DATA    TOTALRAD                          =              2
DATA    RADARNAME                         =SPS-10
DATA    NUMBER                            =              1
DATA    RADARNAME                         =SPS-40
DATA    NUMBER                            =              1
DATA    SHIPNAME                          =BOWEN
DATA    NTYPE                             =              2
DATA    TOTALRAD                          =              2
DATA    RADARNAME                         =SPS-10
DATA    NUMBER                            =              1
DATA    RADARNAME                         =SPS-40
DATA    NUMBER                            =              1
@
ABOK    ABORT RECOGNIZED
EXIT

END OF EXECUTION
CPU TIME: 11.24 ELAPSED TIME: 4:25.83
EXIT
```

Appendix B - Query Language Syntax

The query language uses the following syntax. "[" and "]" indicate optional clauses. "(" and ")" indicate mandatory clauses.

<stmt>::=(("^" or "!")<setname> or
    "-"<recordname>)"("<los>")"[<restmt>]

<restmt>::="$R"<bool exp>["("<los>")"[<restmt>]]

<los>::=<ss>[,<ss>]*

<ss>::=<item1> or <name>":"<item2> or <name>":"<expr> or
    <stmt> or "$P"<arg>

<expr>::=<unaryexp> or <binaryexp>

<unaryexp>::=<unaryop>" "<arg>

<binaryexp>::=<binaryop>" "<arg>" "<arg>

<arg>::=<integerliteral> or <realliteral> or <stringliteral>
    or <name> or <item1>

<item1>::=<item>

<item2>::=<item>

<item>::= item name from schema

<setname>::=setname from schema

<record>::= record name from schema

<name>::= a user defined naME

<unaryop>::="INT" or "NOT" or "SUM" or "COUNT"

<binaryop>::="PLUS" or "MINUS" or "MULTIPLY" or "DIVIDE" or

This is subject to revision: if problems are encountered, please contact Rajeev Sangal.

"EQUAL" or "GT" or "LT" or "GE" or "LE" or "AND" or "OR"

Appendix C - Q Internal Documentation

Q is based on the concept of recursive fortran by having a central subroutine which calls a function for you, to which you "RETURN" with a particular parameter if you want to execute a call. All functions have computed goto's at the beginning so that they can remember where they executed the "call". (see the beginning of q3.F4 for details). A recursive structure made life much easier since the syntax of the language is defined recursively. Furthermore, the parser produces a list structure of "statements" which can also be traversed recursively for execution.

The execution of a query is a 3 step process. The query is read in, parsed, and a list structure is produced in step 1. Then the list structure is preprocessed, to verify that all the referenced items are in the database, and have the correct relationship to one-another. Step 2 also includes allocating temporary variable space, and noting where all the relevant UWA locations are. In step 3, the query is actually processed. Every time the list structure goes another level deep, a new loop is entered which begins with a FINDAP, FINDPO, FINDO or FINDC, which ever is appropriate. From there on, the "statements" in the list structure are processed one at a time. The subroutines PROGS, VARS, and RUNS do these tasks.

PROGS opens an input file, sets up global variables for allocation of symbol and statements. Then STMTS is called recursively. The parser is organized into a set of subroutines which follow the description of the syntax exactly. Subroutine GETNEXT is used to get the next key word and separator from the input file. From that, next allowable state of the parser is entered, by calling STMTS, RESTMTS, LOW, or SS. This process is continued until the entire input is parsed.

VERS calls the recursive function TRAN. TRAN translates the statements, by checking item names and set names, and their relations to one-another, by recording where all the UWA information exists in the unused portions of the statement array, by allocating temporary variable space, and copying literals into the temporary space, and by changing the structure of "reduction" functions (such as count and sum) so that they consist of sequential statements also.

RUNS calls LOOP recursively. LOOP takes as an argument the type of FIND it is to do to get records from the data base. If it is to do a FINDAP, it checks to see if it can substitute a FINDC, and does so if possible. Then, it executes the statements. If it finds a restriction that yields a value of false, then it simply goes back to the beginning of the loop for the next record out of the database.

Appendix D - Q Main Data Structures

Internal list structure format:

```
        stmt(1,n)=pointer to next list element
        stmt(2,n)=statement type
                0-item definition
                2-to owner set
                3-to owned set
                4-assigned function
                5-restricted function
                6-not used
                7-start a record class
                101-header for start of record class
                103-void function from "count" or "sum"
                104-identity element for count etc
        stmt(3,n)=not used
        stmt(4,n)=pointer to user defined symbol for
                        this assignment
        stmt(5,n)=pointer to set name, item name,
                        function name
        stmt(6,n)=pointer to arg1 of fn
        stmt(7,n)=pointer to arg2 of fn
        stmt(8,n)=type of arg1  - 0 undef, 1 character
                        literal in symbol table,
                        2 pointer is integer
                        literal, 3 pointer is floating
                        point literal, 4 pointer is symbol
```

pointer

stmt(9,n)=same as above for arg2

For type 101:

stmt(6,n)=record class index from schema

stmt(7,n)=uwa offset to record (for get)

stmt(8,n)=uwa offset to area for set

(for findpo, or findap)

stmt(9,n)=current level in structure

Symbol Table Information:

udef( )=pointer to header statement in which this label was defined

ucalc( )=0 if label is not a calc key. =index into uwa for calc key if it is a calc key

utloc( )=location in tvar for start of this variable

utlen( )=length in bytes of storage in tvar

utyp( )= 0 - character variable

1 - floating point

2 - integer

3 - double precision floating point

symbol(1-6,n)=symbol itself

fdef( )=function index if this label represents a function

narg(function index)=number of args for the function

DISTRIBUTION LIST

Department of the Navy - Office of Naval Research

Data Base Management Systems Project

Defense Documentation
Cameron Station
Alexandria, VA 22314
12 copies

Office of Naval Research
Branch Office, Chicago
536 South Clark Street
Chicago, IL 60605

Office of Naval Research
New York Area Office
715 Broadway - 5th Floor
New York, N.Y. 10003

Dr. A.L. Slafkosky
Scientific Advisor (RD-1)
Commandant of the Marine Corps
Washington D.C. 20380

Office of Naval Research
Code 458
Arlington, VA 22217

Office of Naval Research
Information Systems Program
Code 437
Arlington, VA 22217
2 copies

Office of Naval Research
Branch Office, Boston
495 Summer Street
Boston, MA 02210

Office of Naval Research
Branch Office, Pasadena
1030 East Green Street
Pasadena, CA 91106

Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375
6 copies

Office of Naval Research
Code 455
Arlington, VA 2217

Naval Electronics Lab. Cener
Advanced Software Technology Div.
Code 5200
San Diego, CA 92152

Mr. E. H. Gleissner
Naval Ship Research and
Development Center

Computation and Mathematic Dept.
Bethesda, MD 20084

Mr. Kim Thompson
Technical director
Information Systems Division
OP-911G
Office of Chief Naval Operations
Washington, D.C. 20350

Prof. Omar Wing
Columbia University
in the City of New York
Dept. of Electrical Engineering
and Computer Science
New York, N.Y. 10027

Commander, Naval Sea Systems Command
Department of the Navy
Washington, D.C. 20362
ATTENTION: PMS30611

Captain Richard Martin, USN
Commanding Officer
USS Francis Marion (LPA-249)
FPO New York 09501

Captain Grace M. Hopper
NAICOM/MIS Planning Branch
OP-916D
Office of Chief of Naval Research
Washington, D.C. 20350

Bureau of Library and
Information Science Research
Rutgers - The State University
189 College Avenue
New Brunswick, N.J. 08903
ATTENTION: Dr. Henry Voos

Defense Mapping Agency
Topographic Center
ATTN: Advanced Technology Div.
Code 41300
6500 Brookes Lane
Washington, D.C. 20315