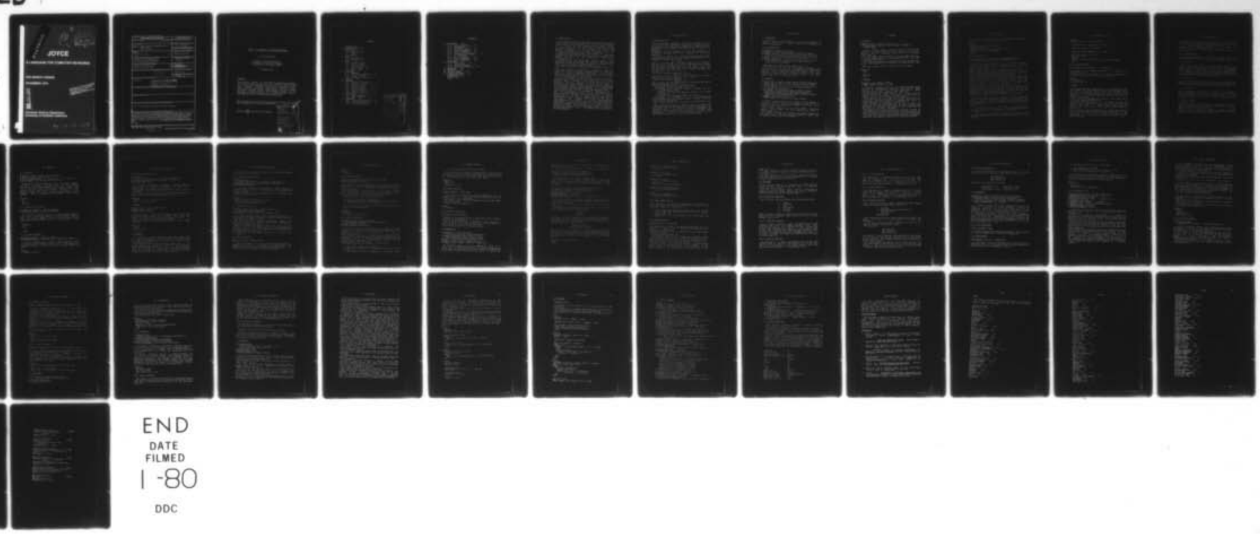
AD-A078 706 UNIVERSITY OF SOUTHERN CALIFORNIA LOS ANGELES DEPT OF--ETC F/G 9/2
JOYCE. A LANGUAGE FOR COMPUTER NETWORKS.(U)
NOV 79 P BRINCH HANSEN N00014-77-C-0714

UNCLASSIFIED NL

| OF |
AD A078706



END
DATE FILMED
1-80
DDC

JOYCE

A LANGUAGE FOR COMPUTER NETWORKS

PER BRINCH HANSEN

NOVEMBER 1979

Computer Science Department
University of Southern California

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	
2. GOVT ACCESSION NO.	
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle)	JOYCE - A LANGUAGE FOR COMPUTER NETWORKS
5. TYPE OF REPORT & PERIOD COVERED	Technical Report
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s)	Per Brinch Hansen
8. CONTRACT OR GRANT NUMBER(s)	N00014-77-C-0714
9. PERFORMING ORGANIZATION NAME AND ADDRESS	Computer Science Department University of Southern California Los Angeles, California 90007
10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	NR048-647
11. CONTROLLING OFFICE NAME AND ADDRESS	Office of Naval Research Arlington, Virginia 22217
12. REPORT DATE	November 1979
13. NUMBER OF PAGES	41
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	12 43
15. SECURITY CLASS. (of this report)	Unclassified
15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	Not applicable
16. DISTRIBUTION STATEMENT (of this Report)	Unlimited This document has been approved for public release and sale; its distribution is unlimited.
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)	
18. SUPPLEMENTARY NOTES	
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)	Joyce, Programming Language, Distributed Processes

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

This report defines an experimental programming language called Joyce which is intended for real-time applications controlled by microcomputer networks without common storage. The language includes distributed processes which communicate and synchronize themselves by means of procedure calls and guarded regions. The present version of the language is implemented on a PDP 11 single-processor system. The compiler is not available for distribution.

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)

410617

JOYCE - A LANGUAGE FOR COMPUTER NETWORKS

Per Brinch Hansen

Computer Science Department
University of Southern California
Los Angeles, California 90007

November 1979

Abstract

This report defines an experimental programming language called Joyce which is intended for real-time applications controlled by microcomputer networks without common storage. The language includes distributed processes which communicate and synchronize themselves by means of procedure calls and guarded regions. The present version of the language is implemented on a PDP 11 single-processor system. The compiler is not available for distribution.

The development of Joyce has been supported by the Office of Naval Research under contract NR048-647.

CONTENTS

CONTENTS

1. INTRODUCTION

This report defines an experimental programming language called Joyce which is intended for real-time applications controlled by microcomputer networks without common storage. It is based on the concurrent programming concept distributed processes [1] which unifies the monitor and process concepts [2, 3] and provides a structured alternative to message communication in networks.

A Joyce program consists of a fixed number of concurrent processes that are started initially and exist forever. Each process can access its own variables only. There are no common variables.

A process can call procedures defined within other processes. These procedures are executed when the other processes are waiting for some conditions to become true. This is the only form of process communication.

Processes are synchronized by means of guarded regions [4, 5].

The data types and sequential statements in Joyce are borrowed from the programming language Pascal [6]. The data types are integers, booleans, characters, arrays, records, and process types. Processes and procedures can be nested arbitrarily and activated recursively.

In a microcomputer network without common storage each processor can be dedicated to the execution of a single process. When a process is waiting for some condition to become true then its processor is also waiting until a procedure call from another process makes this condition true. Parameter passing between processes can be implemented by input/output between separate stores. It is possible that such a network will require a restricted subset of Joyce.

The present version of the language is implemented on a PDP 11 microcomputer. The language includes several machine-dependent features which are necessary to control peripherals on the PDP 11. Input/output is controlled by direct manipulation of device registers without the use of interrupts. The purpose of the single-processor implementation is to discover the algorithmic advantages and limitations of distributed processes in concurrent programs. Since the language is experimental in its present form no attempt has been made to distinguish between its abstract and machine-dependent parts. The compiler is not available for distribution.

2. SYNTAX NOTATION

The programming language consists of three parts: (1) a vocabulary of words and special characters, called symbols; (2) syntactic rules that define sequences of symbols, called sencences; and (3) semantic rules that define the meaning of sencences.

Sentences can be combined to form other sentences. A program is a sentence that is not contained in any other sentence.

A syntactic entity is a class of sencences with common properties that will be defined together. The definition of a syntactic entity has the form

S: E

where S is the name of the entity while E is a syntactic expression that defines the class of sencences that S stands for. The name S consists of one or more words. The first word begins with an upper case letter followed by lower case letters only. Any following words consist of lower case letters only.

A syntax expression has the form

T1 # T2 # ... # Tn

which stands for the union of the alternative sencences defined by the syntax terms T1, T2, ..., Tn.

Each syntax term has the form

F1 F2 ... Fn

which stands for concatenation of the sencences defined by the syntax factors F1, F2, ..., Fn.

Each syntax factor is one of the following:

(1) A symbol stands for itself (see 3.2).

(2) The name of a syntactic entity S stands for the sencences defined by S.

(3) A factor [E] stands for Empty # E (where E is a syntax expression).

(4) A factor [E]* stands for Empty # E # EE # ...

Occurrences of the symbols #, [,], * in sencences are denoted Number sign, Left bracket, Right bracket, Asterisk in the syntax expressions.

This syntax notation is a variant of the Backus-Naur form [7].

Each sencence defined by a syntactic entity is constructed by choosing one of the terms of the syntax expression and replacing each of its factors by one of the symbols (or sencences) which it stands for. If a factor includes other expressions or refers to other syntactic entities by name the sencence construction must be done recursively.

3. VOCABULARY

Each sencence in the language is a finite sequence of symbols chosen from a finite vocabulary. The symbols are character sequences.

3.1. Character set

```
# Character: Graphic # New line
# Graphic: Letter # Digit # Special character # Space
# Letter: a # b # c # d # e # f # g # h # i # j # k # l #
    m # n # o # p # q # r # s # t # u # v # w # x # y # z
# Digit: 0 # 1 # 2 # 3 # 4 # 5 # 6 # 7 # 8 # 9
# Special character: " # ' # ( # ) # + # , # - # . # / #
    : # ; # < # = # > # Number sign # Left bracket #
    Right bracket # Asterisk
```

Characters are used to form symbols (see 3.2).

The character set may be arbitrarily extended. In particular, the letters may be represented in both upper case and lower case with different fonts (roman, italic, or boldface). These different representations of the same letters are equivalent when the letter is part of a word symbol (see 3.2) or a name (see 3.3).

3.2. Symbols

```
# Symbol: Special symbol # Word symbol # Name # Numeral #
    Character constant # String constant # Symbol Comment
# Special symbol: + # - # / # = # <> # < # <= # > # >= #
    := # ( # ) # . # , # : # ; # .. # Left bracket #
    Right bracket # Asterisk
# Word symbol: and # array # begin # const # create #
    cycle # do # else # end # if # not # of # or #
    procedure # process # record # skip # space # start #
    type # val # var # when # while
# Comment: Space # New line #
    "Any sequence of characters without quotes"
```

A symbol denotes a primitive concept of the language. It is either a special symbol, a word symbol, a name (see 3.3), a numeral (see 6.1), a character constant (see 6.2), or a string constant (see 6.3).

The special symbols and the word symbols have fixed meanings. In this report the word symbols are shown in italics (see 3.1).

Any symbol followed by a sequence of comments stands for the symbol itself. Two adjacent word symbols or names must be separated by at least one comment.

3.3. Names

\# Name: Letter [Letter # Digit]*
\# Standard name: boolean # char # false # integer #
 true # write

A name denotes either a constant (see 6.4), a data type (see 5.3), a record field (see 5.4), a variable (see 8.1), a procedure (see 12), a parameter (see 12.1), or a process module (see 13).

In this report names are shown in italics when they occur in syntax expressions and in roman type when they are used in examples (see 3.1).

The standard names have predefined meanings. The meanings of all other names must be defined by declarations (see 4.1).

The word symbols (see 3.2) cannot be used as names.

Examples:

```
char
i
producer
getl
```

4. BLOCKS

\# Block: [Declaration]* Body
\# Body: begin Statement list end

A program consists of entities called blocks. Each block consists of declarations (see 4.1) that define named entities and a body that defines operations on these entities by means of a statement list (see 4.2).

Blocks may contain other blocks. If a block contains another block, the blocks are said to be nested and the latter is called an inner block of the former.

A program (see 14) is a block that is not contained in any other block. The inner blocks of a program are parts of procedures (see 12) and process modules (see 13).

The process of following the text of a block and performing the operations defined by the body is called the execution of the block. The person or device that performs the execution is called a processor.

The execution of a block creates the entities defined by the declarations and executes the body. The execution of the body ends within a finite time (unless it fails or cycles).

The execution fails if the processor detects a meaningless operation. The execution cycles if it continues forever.

The entities declared within a block disappear again when the execution of the block ends or fails.

Example:

```
const midnight = 1440 "minutes"
var due: integer
begin due:= (time + minutes) mod midnight;
  when time = due do skip end
end
```

4.1. Declarations

```
# Declaration: const Constant list # type Type list #
    var Variable list # Procedure # Process module
```

Declarations introduce names (see 3.3) to denote constants (see 6.4), data types (see 5.3), fields (see 5.4), variables (see 8.1), procedures (see 12), parameters (see 12.1), or process modules (see 13).

With the exception of the standard names (see 3.3), all names must be introduced by declarations before they are used in the program text. The standard names are considered to be predeclared at the beginning of a program.

That part of the program text in which a name can be used with a single meaning is called the scope of a name. The name is said to be valid within its scope. The scope generally extends from the declaration of the name to the end of the block in which the declaration appears. This block is known as the *origin of the name*.

A name is said to be local to its origin and global to the inner blocks that are contained in its scope. The local and global names that are valid within a block must all be different. A name can, however, be declared with different meanings in blocks that are not nested.

The general scope rules are modified in three cases:

(1) A field name of a record type is valid within the scope of the record type, but only when it is used to select field variables. The field names of a record type must all be different. The names can be redeclared outside the record type (see 5.4).

(2) A procedure name, which is local to a process module, is also valid within the scope of the process module, but only when it is used in process calls (see 11.3).

(3) The scope of a variable or procedure name does not include process modules that are inner blocks of its origin (see 13).

Examples:

```
  const midnight = 1440 "minutes"

  type identifier = array 1..12 of char

  var this: char; full: boolean

  procedure signal begin s:= s + 1 end

  process source(succ: sink)
  var next: char
  begin
    while true do produce(next); succ.put(next) end
  end
```

4.2. Statement lists

* Statement list: Statement [; Statement]*

A statement list denotes execution of a sequence of statements (see 11).

The execution of a statement list causes the statements to be executed one at a time in the order written.

Examples:

```
  consume(this)
  free:= false; r:= 1
```

5. TYPES

A program defines operations to be performed on data values which are either simple or structured. A simple value can only be operated upon as a whole. A structured value consists of a finite sequence of other values, called subvalues; it can either be operated upon as a whole or subvalue by subvalue.

Data values are grouped into classes, called types, which are either simple or structured.

A simple type determines a set of simple values. The simple types integer, boolean, and character are predefined standard types (see 5.1). Other simple types known as process types are defined by declarations of process modules (see 5.2).

A structured type determines a set of structured values. The structured types are known as record types (see 5.4) and array types (see 5.5). They are defined by means of type declarations in terms of the (previously defined) types of their subvalues.

Every constant, variable or expression is of one and only one type. The types determine the possible values which these entities may assume during program execution.

Every operator expects operands of fixed types and delivers a result of a fixed type.

The types can be determined from the program text without executing it.

5.1. Standard types

The predefined types are called standard types. A standard type determines a finite, ordered set of simple values.

5.1.1. Integers

The standard type integer is denoted by the standard name integer. The integer values are a finite set of successive whole numbers in the range -32768..32767. Non-negative integer values are denoted by decimal or octal numerals (see 6.1). Negative integer values are denoted by octal numerals or are computed by applying the sign inversion operator - to operands with positive integer values (see 9).

5.1.2. Booleans

The standard type boolean is denoted by the standard name boolean. The boolean values are the truth values denoted by the standard names false and true (where false < true).

5.1.3. Characters

The standard type character is denoted by the standard name character. The character values are the ASCII characters denoted by character constants (see 6.2). The ordering of character values is determined by their ordinal values (see 9.2).

5.2. Process types

A process type is denoted by the (previously declared) name of a process module (see 13). The values of a process type are called process references.

The creation of a process of type P assigns a process reference to a variable of type P. The process reference serves to identify the process (see 11.4).

5.3. Type declarations

Type list: Type declaration [; Type declaration]
Type declaration: Record type # Array type

A type declaration defines a new structured type which is either a record type (see 5.4) or an array type (see 5.5).

5.4. Record types

Record type: Type name = record Field list end
Field list: Field declaration [; Field declaration]*
Field declaration: Field name : Field type
Type name: Name
Field name: Name
Field type: Type name

A record type introduces a name, called a type name, to denote a set of structured values, called record values. Each record value consists of a finite sequence of subvalues known as fields. Each field is of some (previously defined) type.

The record type includes a field list consisting of a sequence of field declarations. Each field declaration introduces a name to denote the field. The field type is given by a type name.

A record value contains one field for each field name of the record type. The set of record values consists of all the possible combinations of the possible field values.

Record values are computed by means of assignments to record variables or field variables (see 8.4).

A record type cannot be used as a field type of itself.

Examples:

```
date = record
          day: integer; month: integer; year: integer
       end

attributes = record
                protected: boolean;
                address: integer
             end

datafile = record id: identifier; attr: attributes end
```

5.5. Array types

\# Array type: Type name = array Range of Element type
\# Element type: Type name

An array type introduces a type name to denote a set of structured values, called array values. Each array value consists of a finite sequence of subvalues known as elements. The elements are of the same (previously defined) type.

An element is given by its position in the array value. The positions are denoted by the successive values in a range (see 7). The position of an element is called its index value. The element type is given by another type name.

An array value contains one element for each value in the index range. The set of array values consists of all the possible combinations of of the possible element values.

Array values are computed by means of assignments to array variables or indexed variables (see 8.5).

An array type cannot be used as an element type of itself.

Examples:

```
row = array 1..100 of integer
matrix = array 1..100 of row
catalog = array 'a'..'z' of datafile
```

5.5.1. String types

An array type with n elements of type character is called a string type of length n. The length must be even.

The string values are denoted by string constants (see 6.3).

Example:

```
identifier = array 1..12 of char
```

6. CONSTANTS

\# Constant: Numeral # Character constant #
String constant # Constant name

A constant denotes a fixed value of a fixed type. It is either a numeral (see 6.1), a character constant (see 6.2), a string constant (see 6.3), or the name of a (previously defined) constant (see 6.4).

6.1. Numerals

Numeral: Decimal numeral # Octal numeral
Decimal numeral: Digit [Digit]*
Octal numeral: Number sign Octal digit [Octal digit]*
Octal digit: 0 # 1 # 2 # 3 # 4 # 5 # 6 # 7

A numeral is either decimal or octal. A decimal numeral denotes a non-negative decimal value. An octal numeral denotes an octal value. Numerals are of type integer (see 5.1.1) and have their conventional meaning. The octal numerals in the range #0 .. #077777 correspond to the decimal values 0 to 32767. The octal numerals in the range #100000 .. #177777 correspond to the decimal values -32768 to -1.

Examples:
```
0
914
#177342
```

6.2. Character constants

Character constant: ' Character symbol '
Character symbol: Graphic # @ Numeral

A character constant denotes a value of type character (see 5.1.3). A character constant 'c' denotes the character with the graphic symbol c (see 3.1). A character constant '@n' denotes the character with the ordinal value n (see 9.2).

Examples:
```
'b'
'?'
' '
'@10'
```

6.3. String constants

String constant: ' Character string '
Character string: Character symbol [Character symbol]*

A string constant with n character symbols denotes a value of a string type of length n (see 5.1.1). If the length of a character string s is odd the string is replaced by s@0 (see 6.2).

Example:
```
'syntax error@10'
```

6.4. Constant declarations

```
# Constant list:
    Constant declaration [ ; Constant declaration ]*
# Constant declaration: Constant name = Constant
# Constant name: Name
```

A constant declaration introduces a name, called a constant name, to denote a constant. The type of the constant name is the type of the constant.

A constant declaration cannot use its own constant name as a constant.

The constant names false and true are standard names that denote the values of type boolean (see 5.1.2).

Examples:

```
length = 512; nl = '010'; lf = nl
error = 'end of file.'
```

7. RANGES

```
# Range: Lower bound .. Upper bound
# Lower bound: Constant
# Upper bound: Constant
```

A range denotes a finite set of simple values from a lower bound to an upper bound, both included. The bounds are denoted by constants of the same standard type (see 6). The type of the range is the type of its bounds.

Ranges are used to define the index values of array types (see 5.5).

Examples:

```
1..100
false .. true
'a' .. 'z'
```

8. VARIABLES

A variable is an entity that may assume any of the values of a (previously defined) type. The value of a variable may be used in expressions (see 9) and may be changed by means of assignments (see 11.2).

Variables of simple and structured types (see 5) are called simple and structured variables, respectively. A structured variable consists of a set of other variables known as subvariables. It contains a subvariable for each subvalue of its type. A variable that is not a subvariable of any other variable is called a whole variable.

A processor records the values of the variables by means of a device called a store.

8.1. Variable declarations

Variable list:
  Variable declaration [; Variable declaration]*
Variable declaration: Variable name : Type name
Variable name: Name

A variable declaration introduces a name, called a variable name, to denote a whole variable of the type given by the type name.

Other variables known as parameters are introduced by parameter lists (see 12.1).

Examples:

```
maxno: integer; ok: boolean; c: char
data: matrix; directory: catalog
now: date; name: identifier
```

8.2. Variable selection

Variable: Whole variable [Type transfer]
  Subvariable [Type transfer]
Subvariable: Field variable # Indexed variable

A variable denotes either a whole variable (see 8.3) or a subvariable. The latter is either a field variable (see 8.4) or an indexed variable (see 8.5).

The execution of a variable denotation causes the processor to locate the variable in the store. This process is called variable selection.

When a variable has been selected its value can be retrieved (see 8.7) or changed (see 11.2). The variable can also be bound (see 12.1) to a parameter during the execution of a procedure call (see 11.3).

Variable selection is further explained in sections 8.3, 8.4, and 8.5.

8.3. Whole variables

Whole variable: Variable name

A whole variable is a (previously declared) variable or parameter (see 12.1). It is denoted by a variable name. The type of a whole variable is given by its declaration.

The selection of a whole variable always ends.

Examples:
  now
  directory

8.4. Field variables

\# Field variable: Record variable . Field name
\# Record variable: Variable

A variable of a record type (see 5.4) is called a record variable. It contains a subvariable corresponding to each field of its record value. The subvariables are known as field variables.

A field variable is denoted by a record variable followed by a field name. The type of a field variable is the corresponding field type given by the record type.

A field variable is selected in two steps:

(1) The record variable is selected.
(2) The field variable corresponding to the field name is selected within the record variable.

The selection of the field variable ends if the selection of the record variable ends.

Examples:
  now.day
  directory[c].attr.address

8.5. Indexed variables

\# Indexed variable: Array variable
  Left bracket Index expression Right bracket
\# Array variable: Variable
\# Index expression: Expression

A variable of an array type (see 5.5) is called an array variable. It contains a subvariable corresponding to each element of its array value. The subvariables are known as indexed variables.

An indexed variable is denoted by an array variable followed by an index expression. The index expression is an expression (see 9) of the same type as the index range of the array type. The type of an indexed variable is the element type of the array type.

An indexed variable is selected in three steps:

(1) The array variable is selected.
(2) The index expression is evaluated to obtain an index value.
(3) The indexed variable corresponding to the index value

is selected within the array variable.

The selection of an indexed variable ends if the selection of the array variable and the index expression both end with an index value within the index range. The selection fails if the index value is outside the index range.

Examples:

```
name[i + j]
data[i][j]
directory[c].id[i]
```

8.6. Type transfers

Type transfer: : Type name

A variable v of a type T1 can be made compatible with an operand (see 9) of another type T2 by using the notation v: T2, where T2 is a type name.

The types T1 and T2 must be represented by the same number of store locations.

A type transfer always ends.

Example:

```
c: integer
```

8.7. Variable retrieval

Variable value: Variable

The use of a variable within an expression (see 9) denotes the value of the variable.

During the evaluation of the expression the variable is first selected and then a copy of its value is obtained. This process is called variable retrieval.

The retrieval ends if the selection ends.

9. EXPRESSIONS

```
# Expression: Simple Expression
    [ Relational operator Simple expression ]
# Simple expression: Unary operator Term #
    Simple expression Adding operator Term
# Term: [ Term Multiplying operator ] Factor
# Factor: Simple factor [ Type transfer ]
# Simple factor: Simple operand # ( Expression )
```

An expression denotes a rule for computing a value of a fixed type. An expressibn consists of subexpressions known as operands and operations denoted by operators. Parentheses may be used to explicitly define the order in which the subexpressions are executed. The type of an expression is the type of its value.

The execution of an expression is known as its evaluation.

Operand: Simple expression # Term
  Factor # Simple factor # Simple operand
Simple operand: Constant # Variable

An operand denotes a value of a fixed type.

The evaluation of a simple operand yields the value denoted by a constant (see 6) or a variable (see 8.7). The evaluation of other kinds of operands is defined in the following.

Operator: Relational operator # Adding operator
  Multiplying operator # Unary operator
Relational operator: = # <> # < # <= # > # >=
Unary operator: - # not # val
Adding operator: + # - # or
Multiplying operator: Asterisk # / # mod # and

A unary operator denotes an operation on the value of a single operand. The other operators denote operations on the values of two operands.

The effect of executing the operators is defined in section 10.

An expression (or subexpression) may consist of an operand only (possibly enclosed in parentheses):

Operand
(Operand)

The type of such an expression is the type of the operand. The expression value is obtained by evaluating the operand.

An expression (or subexpression) may also consist of an operator with one or two operands:

Operator Operand
Operand Operator Operand

The type of such an expression is the type of the operator result (see 10). The expression value is obtained by first evaluating the operand(s) one at a time and then performing the operation denoted by the operator on the operand value(s).

Examples of simple operands:
  80
  free

Examples of simple factors:
  ... all the examples above ...
  (here = 0)

Examples of factors:
  ... all the examples above ...
  c: integer

Examples of terms:
  ... all the examples above ...
  not full
  (time + minutes) mod midnight

Examples of simple expressions:
  ... all the examples above ...
  x - y + z

Examples of expressions:
  ... all the examples above ...
  c <> em

9.1. Type compatibility

An operation can only be performed on two operands if their data types are compatible, that is if one of the following conditions is satisfied:

(1) Both types have the same standard name or are defined by the same type declaration (see 5.3) or process module (see 5.2).
(2) Both types are string types of the same length (see 5.5.1).

9.2. Type transfers

Type transfer: : Type name

An operand x of a type T1 can be made compatible with an operand of another type T2 by using the notation x: T2, where T2 is a type name.

The types T1 and T2 must be represented by the same number of store locations.

A type transfer always ends.

If x denotes a value of type integer and y denotes a value of any standard type T then x = y:integer if and only if y = x:T. The values of x and y are called corresponding values. The value of x is also called the ordinal value of y.

For any integer value x we have x:integer = x. The boolean values have the ordinal values false: integer = 0 and true: integer = 1. The ordering of the characters in the ASCII character set determines their ordinal values in the range 0..127.

The type transfer of a value x from one standard type T1 to another standard type T2 satisfies the relation x:T2 = x:integer:T2.

The type transfer of a value x from a non-standard type T1 to another type T2 yields a value of type T2 with the same storage representation as the value x.

10. OPERATORS

Each operator applies to operands of a fixed type and delivers a result of a fixed type. The type of an operator is the type of its result.

When the same operator symbol applies to operands of different types the symbol stands for several different operations determined by the operand types.

10.1. Relational operators

The relational operators denote the following relations

=	equal
<>	not equal
<	less
<=	not greater
>	greater
>=	not less

These operators generally apply to operands of any type as defined in the following. They yield a result of type boolean (see 5.1.2).

10.1.1. Standard operands

The relational operators apply to operands of the same standard type (see 5.1). These operators have their conventional meaning for operands of type integer (see 5.1.1): they yield the value *true* if the operand values satisfy the relations, and the value *false* if they do not. If x and y are operands of another standard type T then x = y means x:integer = y:integer, and similarly for the other relations (see 9.2).

10.1.2. Process operands

The operators =, <> apply to operands x and y of the same process type P (see 5.2). The relation x = y is true if x and y are references to the same process, and false otherwise. The relation x <> y means *not* (x = y).

10.1.3 Record operands

The operators =, <> apply to variables x and y of the same type T = record f1: T1; f2: T2; ...; fn: Tn end. The relation x = y means (x.f1 = y.f1) and (x.f2 = y.f2) ... and (x.fn = y.fn), where f1, f2, ..., fn denote the fields of the record type. The relation x <> y means not (x = y).

10.1.4. Array operands

The operators =, <> apply to operands x and y of the same array type T = array i1..in of Te. The relation x = y means (x[i1] = y[i1]) and (x[i2] = y[i2]) ... and (x[in] = y[in]) where i1, i2, ..., in denote the successive index values of the array type. The relation x <> y means not (x = y).

10.2. Integer operators

The integer operators apply to operands of type integer and yield a result of type integer (see 5.1.1):

+	addition
-	subtraction (or sign inversion)
*	multiplication
/	division
mod	modulus

These operators have their conventional meaning. When the symbol - is used as a unary operator it denotes sign inversion.

If the result of one of these operators is outside the range of integers the execution fails.

The operators

not	negation
or	conjunction
and	disjunction

also apply to integer operands. The operations are performed on all bits in the storage representation of the integer values (see 10.3). The resulting bits are then considered to be an integer result.

The unary operator val applies to an integer operand x and yields the integer value of the device location with the address x. The value of x must be even in the range #160000 .. #177776, otherwise the execution fails.

10.3. Boolean operators

The boolean operators apply to operands x and y of type boolean and yield a result of type boolean (see 5.1.2):

<u>not</u> negation
<u>or</u> conjunction
<u>and</u> disjunction

The results are defined as follows:

<u>not</u> false = true	<u>not</u> true = false
x <u>or</u> false = x	x <u>or</u> true = true
x <u>and</u> false = false	x <u>and</u> true = x

11. STATEMENTS

<u>Statement</u>: Simple statement # Structured statement
<u>Simple statement</u>: Skip statement # Assignment
    Procedure call # Create statement # Start statement
<u>Structured statement</u>: If statement # While statement
    When statement # Cycle statement

A statement denotes one or more operations and is either simple or structured. A simple statement denotes an elementary operation and is either a skip (see 11.1), an assignment (see 11.2), a procedure call (see 11.3), a create statement (see 11.4), or a start statement (see 11.5).

A structured statement consists of other statements, called substatements. It determines (at least partially) the order in which the substatements are to be executed. The structured statements are called if statements (see 11.7), while statements (see 11.8), when statements (see 11.9), and cycle statements (see 11.10).

11.1. Skip statements

<u>Skip statement</u>: <u>skip</u>

The symbol <u>skip</u> denotes the empty operation. The execution of a skip statement has no effect and always ends.

11.2. Assignments

<u>Assignment</u>: Variable := Expression

An assignment denotes assignment of a value given by an expression (see 9) to a variable (see 8.2). The variable and the expression must be compatible (see 9.1).

An assignment is executed in three steps:

(1) The variable is selected.
(2) The expression is evaluated to obtain a value.
(3) The value is assigned to the variable.

An assignment to a structured variable assigns a value to all of its subvariables. An assignment to a subvariable of a structured variable assigns a value to the given subvariable without changing the rest of the subvariables.

Examples:

```
x:= x - y
now.month:= 11
directory[c].id:= 'spascaltext '
```

11.3. Procedure calls

```
# Procedure call: Local call # Process call
# Process call: Process variable . Local call
# Process variable: Variable
# Local call: Procedure name [ ( Argument list ) ]
# Procedure name: Name
# Argument list: Argument [ , Argument ]*
# Argument: Value argument # Variable argument
# Value argument: Expression
# Variable argument: Variable
```

A procedure call denotes execution of the block given by a procedure name (see 12). It is either a local call or a process call.

A local call is used by a process (see 13) to execute a procedure that operates on the variables of the process. The procedure must be declared within the innermost process module that contains the call.

A process call is used by a process to execute a procedure that operates on the variables of another process. The other process is given by the value of a process variable of some type P (see 5.2). The procedure must be declared within the process module P.

The argument list denotes values and variables that may be operated upon by the procedure block. The argument list must contain one argument for each parameter name in the parameter list of the procedure (see 12.1). The order in which the arguments and the parameter names are written in the argument list and the parameter list defines a one to one correspondence between the arguments and the parameters.

Each argument is either a value argument or a variable argument.

A value argument corresponds to a value parameter. It must be an expression (see 9) that is compatible with the corresponding parameter. An argument corresponding to a value parameter of a string type (see 5.5.1) may, however, be a string constant of any length.

A variable argument corresponds to a variable parameter. It must be a variable (see 8.2) of the same type as the corresponding parameter.

The parameter types are given by the parameter list of the procedure.

The execution of a local call takes place in two steps:

(1) The arguments are evaluated one at a time in the order written. A value argument is evaluated by evaluating the given expression to obtain a value which is then assigned to the corresponding parameter. A variable argument is evaluated by selecting the given variable and binding (see 12.1) the corresponding parameter to it.

(2) The procedure block is executed.

The execution of a process call is delayed until it can be performed as an indivisible operation (see 13). It then proceeds as a local call.

The execution of a procedure call ends when the execution of the procedure block ends.

The execution of a process call fails if the given process has not been created (see 11.4).

Examples:
```
finish
read(x, y)
timer.wait(15)
chain[1].put(A[i])
```

11.4. Create statements

\# Create statement:
    create Process variable [, Process Variable]*

A create statement denotes the creation of one or more processes (see 13) and the assignment of references to these processes to a list of process variables.

The execution of a create statement creates the processes in the order in which the variables are written. If a variable v is of a process type P then the operation create v will create a process of type P and assign a reference to that process to the variable v.

The operation create v, v, ...v (where v occurs n times) creates n processes of type P and assigns a reference to the n'th process to v.

Examples:
  create consumer, producer
  create ring[i]

11.5. Start statements

\# Start statement:
  start Process start [, Process start]*
\# Process start: Process variable [(Argument list)]

A start statement denotes the starting of one or more processes given by the values of a list of process variables.

Each argument list denotes values that may be operated upon by a single process. The rules of the argument list are the same as those defined for procedure calls (see 11.3).

The execution of a start statement starts the processes in the order in which the process variables are written. Each process is started in two steps:

(1) The arguments are evaluated one at a time in the order written and their values are assigned to the corresponding process parameters.

(2) The execution of a process given by the value of a process variable begins.

When a process has been started by another process the two processes continue to operate simultaneously.

A process must be created (see 11.4) before it is started, and can only be started once. Otherwise the start operation fails.

Examples:
  start consumer, producer(consumer)
  start ring[i](ring[i - 1], ring[i + 1])

11.6. Conditional statements

\# Conditional statement list:
  Conditional statement [else Conditional statement]*
\# Conditional statement: Expression do Statement list

A conditional statement list denotes the execution of one of several conditional statements (or none of them).

Each conditional statement consists of an expression (see 9) of type boolean and a statement list (see 4.2).

The execution of a conditional statement list evaluates the expressions one at a time in the order written until one of them yields the value true or until all of them yield the value false. If the value true is obtained from an expression then the statement list that follows the expression is executed; otherwise, none of the statement lists are executed. In the former case, one of the conditional statements is said to be executed, while in the latter case all of them are said to be skipped. This ends the execution of the conditional statement list.

Examples:

```
  c <> em do read(c)

  free  do free:=false; r:= 1 else
  r > 0 do r:= r + 1

  op = 1 do create else
  op = 2 do delete else
  op = 3 do rename
```

11.7. If statements

\# If statement: if Conditional statement list end

An if statement denotes a single execution of a conditional statement list (see 11.6).

The execution of an if statement executes the conditional statement list once.

Examples:

```
  if eof do formfeed; eof:=false end

  if op = 1 do create
  else op = 2 do delete
  else op = 3 do rename end
```

11.8. While statements

\# While statement: while Conditional statement list end

A while statement denotes one or more executions of a conditional statement list (see 11.6).

The execution of a while statement executes the conditional statement list repeatedly until all the conditional statements are skipped.

If the conditional statements continue to be executed forever the execution cycles.

Examples:

```
  while c <> em do read(c) end

  while x > y do x:= x - y
  else y > x do y:= y - x end
```

11.9. When statements

When statement: when Conditional statement list end

A when statement denotes one or more executions of a conditional statement list (see 11.6).
The execution of a when statement executes the conditional statement list repeatedly until one of its conditional statements has been executed.
As long as the conditional statements are skipped the when statement is said to be blocked; otherwise it is said to be feasible.
When a process (see 13) attempts to execute a blocked statement it can only become feasible if another process or a peripheral device changes the variables of the given process by a process call or by an input/output operation (see 11.3).
If the when statement continues to be blocked the execution cycles.

Examples:

```
when r + w = 0 do w:= 1 end

when free do free:=false; r:= 1
else r > 0 do r:= r + 1 end
```

11.10. Cycle statements

Cycle statement: cycle Conditional statement list end

A cycle statement denotes the repeated execution forever of a conditional statement list (see 11.6).
The execution of a cycle statement executes the conditional statement list forever.
As long as the conditional statements are skipped the cycle statement is said to be blocked; otherwise it is said to be feasible (see 11.9).

Examples:

```
cycle full do consume(this); full:= false end

cycle here = 2 do transmit
else (here = 2) and (rest > 0) do receive end
```

12. PROCEDURES

Procedure: procedure Procedure name
[(Parameter list)] Procedure block
Procedure name: Name
Procedure block: Block

A procedure introduces a name, called a procedure name, to denote a parameter list (see 12.1) and a block (see 4) which is known as a procedure block.

The execution of a procedure is caused by a procedure call (see 11.3). It creates the entities defined by the parameter list and executes the block.

A procedure may call itself recursively.

Example:

```
procedure wait(minutes: integer)
const midnight = 1440 "minutes"
var due: integer
begin due:= (time + minutes) mod midnight;
  when time = due do skip end
end
```

12.1. Parameters

\# Parameter list:
  Parameter declaration [; Parameter declaration]*
\# Parameter declaration:
  Value parameter # Variable parameter
\# Value parameter: Variable declaration
\# Variable parameter: var Variable declaration

A parameter declaration introduces a variable (see 8.1) which is called a value parameter or a variable parameter.

A value parameter is a variable that is assigned the value of an argument (see 11.3) before the procedure block is executed.

A variable parameter denotes a variable argument (see 11.3) which is selected before the procedure block is executed. During the execution of the procedure block all operations performed on the variable stand for the same operations performed on the variable argument. The variable parameter is said to be bound to the variable argument during the given execution of the procedure block.

Examples:

```
minutes: integer
var value: char
pred: node; succ: node
```

12.2. Local variables

The scope of the names declared within a procedure extends from their declarations to the end of the procedure block. The names are therefore said to be local to the procedure (see 4.1).

Each execution of a procedure block creates a fresh instance of the parameters and local variables. This is known as a procedure instance. When the execution of the procedure block begins the values of the parameters are determined by the arguments of a procedure call. The initial values of the local variables are undefined and must be defined by assignments (see 11.2) before they are used within the procedure block.

When the execution of the procedure block ends the procedure instance disappears and the execution of the calling process (see 13) continues with the statement that follows the procedure call in the program text.

12.3. Standard procedures

The following procedure is considered to be predeclared at the beginning of each process module:

```
procedure write(address: integer; value: integer)
```

When this procedure is called it assigns an integer value to a device location with a given address. The address must be an even integer in the octal range #160000 .. #177776; otherwise, the execution fails.

13. PROCESSES

```
# Process module:
    process Process name [ ( Parameter list ) ]
    Space reservation Process block
# Process name: Name
# Process block: Block
# Space reservation: space Constant
```

A process module introduces a name, called a process name, to denote a parameter list (see 12.1) and a block (see 4) which is known as a process block.

The parameter list must only contain value parameters of simple types (see 5).

The local variables and procedures declared within a process module are called communication variables and communication procedures.

The body of the process block is called the initial statement of the process block. The initial statement and the bodies of the communication procedures are known as the main statements of the process block.

A process block can be executed simultaneously by several processors as long as they operate on different instances of the communication variables.

Each execution of a process block is called a process and the variable instances on which the process operates are called its context.

The execution of a create statement (see 11.4) creates a process with a empty context.

The execution of a start statement (see 11.5) adds a fresh instance of the communication variables to the context and initializes the parameter values. It then begins the execution of a process block.

A process can operate on its own communication variables by means of local calls of procedures declared within the process module. When a process begins the execution of a local call a fresh instance of the procedure variables is added to its context, and when the execution of the procedure ends these variables are removed from its context.

A process can operate on the communication variables of another process by means of process calls on communication procedures declared within the process module of the other process. When a process begins the execution of a process call the (already existing) instances of the given communication variables are added to the context of the process. The execution of the procedure then proceeds as a local call. When the execution of the procedure ends (and its local variables have been removed from the context) the communication variables are also removed from the context.

The above defines the dynamic change of the context of a process. When a process refers to a whole variable by its name, this selects the most recent instance of that variable in the current context of the process.

All operations on the variables of a process are indivisible in the sense that they are performed one at a time as explained below:

An indivisible operation begins when a process begins the execution of a main statement (or a feasible when or cycle statement).

An indivisible operation ends when a process ends the execution of a main statement (or reaches a blocked when or cycle statement).

If several processes attempt to operate on the same variable instances during the same interval of time the indivisible operations on these variables will be performed one at a time in unspecified order.

The first indivisible operation on the variables of a process begins when the process is started.

When this initial operation ends the original process may perform another indivisible operation (if it is now feasible), or another process may either begin a process call or continue one by performing an indivisible operation that has become feasible within a communication procedure.

This interleaving of indivisible operations on the variables of a process continues forever. If the process reaches the end of the process block other processes can still continue to operate on the communication variables by means of process calls.

A process module may create and start instances of itself recursively.

A process module that is contained within another process module cannot refer to the variables and procedures that are declared on the outer module (see 4.1).

The compiler determines the storage space required for each process under the assumption that all procedures are non-recursive. The number of additional bytes of storage locations required for the variables of recursive procedures must be defined by an integer constant following the symbol space.

Example:

```
process sink
var this: char; full: boolean

procedure put(c: char)
begin
  when not full do this:= c; full:= true end
end

begin full:= false
  cycle full do continue(this); full:= false end
end
```

Example:

```
process source(succ: sink)
var next: char
begin
  while true do produce(next); succ.put(next) end
end
```

Example:

```
process semaphore
var s: integer

procedure wait
begin when s > 0 do s:= s - 1 end end

procedure signal
begin s:= s + 1 end

begin s:= 0 end
```

14. PROGRAMS

Program: Block

A program is a block (see 4) that is not contained in any other block.

The execution of a program causes the block to be executed as a process (see 13) called the initial process.

The execution of a program never ends. It either cycles or fails.

Example:

```
  process sink ... begin ... end

  process source(succ: sink) ... begin ... end

  var consumer: sink; producer: source
  begin create consumer, producer;
    start consumer, producer(consumer)
  end.
```

Example:

```
  process node(pred: node; succ: node) ...
  begin ... end

  type nodes = array 0..9 of node

  var ring: nodes; i: integer
  begin i:= 0;
    while i <= 9 do create ring[i]; i:= i + 1 end;
    i:=0;
    while i <= 9 do
      start ring[i](ring[(i + 9) mod 10],
        ring[(i + 1) mod 10]);
      i:= i + 1
    end
  end.
```

Example:

```
  process tree(level: integer; maxlevel: integer)
  var left: tree; right: tree
  begin
    if level < maxlevel do
      create left, right;
      start left(level + 1, maxlevel),
            right(level + 1, maxlevel)
    end
  end;

  var root: tree
  begin create root; start root(1, 4) end.
```

15. SYNTAX SUMMARY

```
# Name: Letter [ Letter # Digit ]*
# Numeral: [ Number sign ] Digit [ Digit ]*
# Character symbol: Graphic # @ Numeral
# Constant: Numeral # Constant name #
    ' Character symbol [ Character symbol ]* '
# Constant declaration: Constant name = Constant
# Constant list:
    Constant declaration [ ; Constant declaration ]*
# New type: record Field list end #
    array Constant .. Constant of Type name
# Field list: Field declaration [ ; Field declaration ]*
# Field declaration: Field name : Type name
# Type declaration: Type name = New type
# Type list: Type declaration [ ; Type declaration ]*
# Variable declaration: Variable name : Type name
# Variable list:
    Variable declaration [ ; Variable declaration ]*
# Declaration: const Constant list # type Type list #
    var Variable list # Procedure # Process module
# Variable: Variable name
    [ . Field name # Left bracket Expression Right bracket ]*
    [ : Type name ]
# Simple factor: Constant # Variable # ( Expression )
# Factor: Simple factor [ : Type name ]
# Term: Factor [ Multiplying operator Factor ]
# Multiplying operator: Asterisk # / # mod # and
# Simple expression:
    [ - # not # val ] Term [ Adding operator Term ]*
# Adding operator: + # - # or
# Expression: Simple expression
    [ Relational operator Simple expression ]
# Relational operator: = # <> # < # <= # > # >=
# Statement: skip # Variable := Expression #
    Procedure call #
    create Process variable [ , Process variable ]* #
    start Process start [ , Process start ]* #
    if Conditional statement list end #
    while Conditional statement list end #
    when Conditional statement list end #
    cycle Conditional statement list end
# Procedure call: [ Process variable . ]
    Procedure name [ ( Argument list ) ]
# Argument list: Argument [ , Argument ]*
# Argument: Expression # Variable
# Process start: Process variable [ ( Argument list ) ]
# Conditional statement list:
    Conditional statement [ else Conditional statement ]*
```

```
# Conditional statement:
    Expression do Statement list
# Statement list: Statement [ ; Statement ]*
# Procedure:
    procedure Procedure name [ ( Parameter list ) ] Block
# Parameter list:
    Parameter declaration [ ; Parameter declaration ]*
# Parameter declaration: [ var ] Variable declaration
# Block: [ Declaration ]* begin Statement list end
# Process module: process Process name
    [ ( Parameter list ) ] [ space Constant ] Block
# Program: Block .
```

16. STORAGE AND SPEED

Each integer, boolean, character, or process reference value requires 2 bytes of storage.

A string value of length n requires n bytes of storage.

The storage requirement of a record or array value is the sum of the storage required for each of its subvalues (fields or elements).

The following is the evaluation times of operands and the execution time of operators and statements in microseconds (measured on an LSI-11 with MOS memory.) In the following n stands for the number of simple values in the operands, v stands for the time to select a variable, E and S stand for the time to evaluate an expression and execute a statement list respectively, while m stands for the number of times a conditional statement list is executed in a while statement.

constant c	15
whole variable v	25
field variable v.f	25+v
indexed variable v[E]	85+E+v
:=	15*n
= <> < <= > >=	15+15*n
and	25
or not	15
+ -	20
*	60
/ mod	120
val	20
write(E,E)	25+E+E
if E do S end	25+E+S
while E do S end	25+E+(25+E+S)*m
when E do S end	125+E+S
procedure call P	175
procedure call v.P	220+v

In this implementation, the processor executes one indivisible operation at a time, and then switches to another process. The switching among processes is cyclic, but the precise order of execution of processes is generally unpredictable. The time to switch from one process to another (120 microseconds) is usually small compared with the time required to execute an indivisible operation.

ACKNOWLEDGEMENT

The programming language is named after the author James Joyce. Charles Hayden wrote the compiler in Sequential Pascal and the code interpreter for the LSI-11 microcomputer. Hayden's Ph.D. thesis explains the process implementation and discusses several Joyce programs [8]. The work has been supported by the Office of Naval Research under contract NR048-647.

REFERENCES

1. Brinch Hansen, P., Distributed processes: a concurrent programming concept. Comm. ACM 21, 11 (Nov. 1978), 934-941.

2. _____, Operating system principles. Prentice-Hall, Englewood Cliffs, NJ, July 1973.

3. Hoare, C.A.R., Monitors: an operating system structuring concept. Comm. ACM 17, 10 (Oct. 1974), 549-57.

4. _____, Towards a theory of parallel programming. In Operating systems techniques, Academic Press, New York, NY, 1972.

5. Brinch Hansen, P., and Staunstrup, J., Specification and implementation of mutual exclusion. IEEE Trans. on Software Engineering 4, 5 (Sept. 1978), 365-70.

6. Wirth, N., The programming language Pascal (Revised report). ETH, Zurich, Switzerland, July 1973.

7. Naur, P. (ed.), Revised report on the algorithmic language ALGOL 60, Comm. ACM 6, 1 (Jan 1963).

8. Hayden, C. Distributed processes: experience and architectures. Computer Science Department, University of Southern California, Los Angeles, CA, 1979.

INDEX

The names of syntactic entities are shown in italics while semantic terms are shown in roman type. The entries refer to section numbers of the report.

DISTRIBUTION LIST

FOR THE TECHNICAL, ANNUAL, AND FINAL REPORTS

FOR CONTRACT N00014-77-C-0714

Defense Documentation Center Cameron Station Alexandria, VA 22314	12 copies
Office of Naval Research Information Systems Program Code 437 Arlington, VA 22217	2 copies
Office of Naval Research Code 715LD Arlington, VA 22217	6 copies
Office of Naval Research Code 200 Arlington, VA 22217	1 copy
Office of Naval Research Code 455 Arlington, VA 22217	1 copy
Office of Naval Research Code 458 Arlington, VA 22217	1 copy
Office of Naval Research Branch Office, Boston 495 Summer Street Boston, MA 02210	1 copy
Office of Naval Research Branch Office, Chicago 536 South Clark Street Chicago, Illinois 60605	1 copy
Office of Naval Research Branch Office, Pasadena 1030 East Green Street Pasadena, CA 91106	1 copy
Office of Naval Research New York Area Office 715 Broadway - 5th Floor New York, NY 10003	1 copy

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

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

Naval Ocean Systems Center — 1 copy
Advanced Software Technology Division
Code 5200
San Diego, CA 92152

Mr. E. H. Gleissner — 1 copy
Naval Ship Research & Development
Center
Computation and Mathematics Department
Bethesda, MD 20084

Captain Grace M. Hopper — 1 copy
NAICOM/MIS Planning Branch (OP-916D)
Office of Chief of Naval Operations
Washington, D. C. 20350

Mr. Kin B. Thompson — 1 copy
NAVDAC 33
Washington Navy Yard
Washington, D.C. 20374