AD-A046 372 WISCONSIN UNIV MADISON MATHEMATICS RESEARCH CENTER F/G 9/2
THE INTERVAL ARITHMETIC PACKAGE.(U)
JUN 77 J M YOHE
UNCLASSIFIED MRC-TSR-1755 DAAG29-75-C-0024
NL

| OF |
ADA
046372





END
DATE
FILMED
12-77
DDC

AD A046372

(9) MRC Technical Summary Report #1755

(6) THE INTERVAL ARITHMETIC PACKAGE

(10) J. M. Yohe

(11) BS

(14) MRC-TSR-1755

Mathematics Research Center
University of Wisconsin—Madison
610 Walnut Street
Madison, Wisconsin 53706

DDC
RECEIVED
NOV 15 1977
F.

(11) June 1977

(12) 88 p.

Received May 23, 1977

DDC FILE COPY

(15) DAAG29-75-C-0024

Approved for public release
Distribution unlimited

Sponsored by

U. S. Army Research Office
P.O. Box 12211
Research Triangle Park
North Carolina 27709

14D
221200
4B

UNIVERSITY OF WISCONSIN - MADISON
MATHEMATICS RESEARCH CENTER

THE *INTERVAL* ARITHMETIC PACKAGE

J. M. YOHE

Technical Summary Report # 1755
Change Notice 1
September, 1977

This change notice consists of two replacement sheets (four pages) -- pp. 17, 18, 63, and 64. Please remove the designated pages from your copy of the report and insert the replacement pages in the appropriate places.

This change redefines the relational operators INTSGE, INTSGT, INTSLE, and INTSLT. This redefinition was occasioned by a discussion with Prof. Karl Nickel of the University of Freiburg, Germany; Prof. Nickel felt that it was inappropriate to define relational operators which did not induce a partial ordering on the space of intervals. We have redefined the relations (3.7) so that they now do induce a partial ordering; however, by reason of computational convenience, we have kept the relations (3.6) unchanged even though they do not induce a partial ordering.

UNIVERSITY OF WISCONSIN - MADISON
MATHEMATICS RESEARCH CENTER

THE *INTERVAL* ARITHMETIC PACKAGE

J. M. YOHE

Technical Summary Report # 1755

June 1977

ABSTRACT

This report provides user documentation and technical documentation for a package of FORTRAN subroutines for performing interval arithmetic calculations. Apart from a relatively small number of primitives and constants, the package is directly transferrable to most large scale computing systems, and information for implementing the package on other systems is included. The implementation described herein is that for the UNIVAC 1110, but the package has also been implemented on CDC, DEC, Honeywell, and IBM equipment.

This package has been designed so as to be compatible with the AUGMENT precompiler, and includes interval analogs of appropriate standard FORTRAN functions and operators, as well as many operations and functions peculiar to interval arithmetic. The user who has access to AUGMENT may write programs to perform interval arithmetic calculations just as though FORTRAN recognized INTERVAL as a standard data type.

AMS(MOS) Subject Classification: 94-04

Key words: Interval Arithmetic Program Package
Portable software
User documentation
Technical documentation

Work Unit No. 8 (Computer Science)

Sponsored by the United States Army under Contract Number DAAG29-75-C-0024.

EXPLANATION

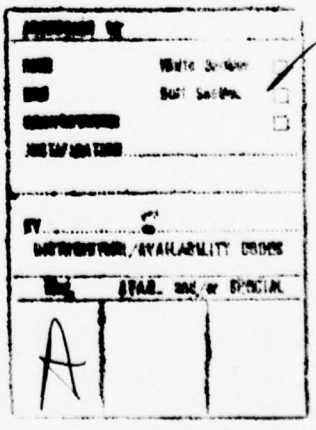
Interval arithmetic is a means of obtaining information about the accuracy of computed results by calculating with pairs of approximate real numbers rather than a single approximation. The first member of the pair is a lower bound for the true result; the second, an upper bound. Uncertainty in data can be taken into account by the use of an appropriate interval; for example, if we are given the value 2.05, good to three significant digits, we would use the interval [2.045, 2.055] instead of the approximation to 2.05. The package itself takes roundoff error into consideration; when a new interval is computed, the left endpoint is always rounded down, and the right endpoint is rounded up. Thus, at each stage of the computation, the interval result contains the true result.

Interval arithmetic can be used in any situation where rigorous error bounds on the results of a computation are required. For example, if interval arithmetic is used to calculate the stresses on a critical component of a mechanical device, the upper bound(s) of the resulting interval(s) would provide worst-case information about the stresses resulting from the given loading(s); any computational instability and roundoff error would be taken into account.

The purpose of this paper is to provide documentation for the INTERVAL arithmetic package--a collection of FORTRAN subroutines for performing interval arithmetic calculations.

TABLE OF CONTENTS

1. Introduction 1
2. Using the INTERVAL arithmetic package 2
   2.1 Using the package with the AUGMENT precompiler 2
   Declaring INTERVAL variables 2
   Assigning constant values to INTERVAL variables 2
   Reading INTERVAL variables -- free format 3
   Reading INTERVAL variables -- formatted 5
   Computing with INTERVAL variables 6
   Writing INTERVAL variables 9
   Errors 10
   Producing an object program 10
   2.2 Using the package without benefit of AUGMENT 11
   2.3 Using BPA variables in the source program 13
3. Theoretical basis 14
4. Design and implementation of the package 23
5. Adapting the package to other hardware 43
6. Conclusion and Disclaimer 55
   References 56
   Appendix 1: Standard FORTRAN and INTERVAL number represenatains ... 58
   Appendix 2: Package modules 59
   Appendix 3: Fault information 65
   Appendix 4: COMMON block information 66
   Appendix 5: Description decks 77
   Appendix 6: Sample Interval Arithmetic program 78
   Appendix 7: Run stream for the MRC UNIVAC 1110 version 81



THE *INTERVAL* ARITHMETIC PACKAGE

J. M. YOHE

1. Introduction:

The purpose of this paper is to provide documentation for the INTERVAL arithmetic package -- a collection of FORTRAN subroutines for performing interval arithmetic calculations.

Interval arithmetic is a means of obtaining information about the accuracy of computed results by calculating with pairs of approximate real numbers rather than a single approximation. The first member of the pair is a lower bound for the true result; the second, an upper bound. Uncertainty in data can be taken into account by the use of an appropriate interval; for example, if we are given the value 2.05, good to three significant digits, we would use the interval [2.045, 2.055] instead of the approximation to 2.05. The package itself takes roundoff error into consideration; when a new interval is computed, the left endpoint is always rounded down, and the right endpoint is rounded up. Thus, at each stage of the computation, the interval result contains the true result.

This paper is intended to be a reference document for the user of the INTERVAL package. As such, its organization may seem cumbersome to one who is not familiar with interval arithmetic. Section 2 is the user's reference manual; it presumes familiarity with the basic concepts, which are not presented until Section 3. It is suggested that the uninitiated reader begin with Section 3, and then read Section 2. Section 4 discusses the design of the package, and is intended to provide the information necessary for modifying and maintaining the package, should this be necessary. Section 5 contains instructions for adapting the package to a different host computer.

The appendices contain technical information such as structure of external data, tables of available functions, error indications, COMMON block contents, a sample program, and the UNIVAC 1110 runstream for using the package. A listing of the package and test program for the UNIVAC 1110 version is provided on the microfiche included with this report.

Sponsored by the United States Army under Contract Number DAAG29-75-C-0024.

2. Using the INTERVAL arithmetic package:

This section is designed to be the user's manual for the INTERVAL arithmetic package. The package is most easily used with the aid of the AUGMENT precompiler; this use will be described first. Following that, we discuss the use of the package without benefit of the precompiler; essentially, this is just a discussion of how the user should go about doing the things that the precompiler would do were it available.

2.1 Using the package with the AUGMENT precompiler:

Declaring INTERVAL variables: If X is an interval variable, Y is an interval vector of length 10, and Z is a 10 × 10 interval matrix, the following type declaration will both establish them as INTERVAL variables and reserve storage for them:

```
INTERVAL X, Y(10), Z(10,10)
```

Restrictions:

Identifiers should not begin with the prefixes 'INT' or 'BPA'; this will help avoid conflicts with the names of subroutines in the package.

If the FORTRAN compiler limits the number of dimensions of an array, that limit must be decreased by 1 for INTERVAL variables; AUGMENT will declare an undimensioned INTERVAL variable as a vector.

Options: None

Assigning constant values to INTERVAL variables: This may be accomplished by a statement which assigns the value of an appropriate Hollerith string to the interval variable. For example, to assign the value (.1, .1) to the INTERVAL variable X, one would write

```
X = 9H(.1, .1)$
```

Restrictions:

The Hollerith string must represent an INTERVAL as defined in Appendix 1, except that embedded blanks are permitted and will be ignored.

The Hollerith string must be terminated by a field separator: $, #, or =.

The length of the Hollerith string is limited to the length of the length of the formatted read buffer (132 characters in the UNIVAC 1110 version).

The number of significant digits permitted for each endpoint of the

interval is limited to the value of ESDMAX (60 in the UNIVAC 1110 implementation).

The user should be aware that a statement of the form

```
X = .1
```

will cause a value to be assigned to X, but the resulting interval may not contain .1. The above statement will cause X to be set equal to a degenerate interval each of whose endpoints is the REAL approximation to .1; in order to obtain a rigorous interval approximation to .1 on most computers, the interval would need to be nondegenerate. Thus shortcutting the conversion from Hollerith to INTERVAL can be dangerous.

Options:

Quoted Hollerith literals may be used if the host compiler accepts them.

If the host compiler generates a sentinel for Hollerith literals, and if the INTUPK primitive recognizes this sentinel (as is the case in the UNIVAC 1110 implementation), the terminal field separator may be omitted.

Implicit conversions from Hollerith to INTERVAL are allowed (e. g., a Hollerith literal may appear in an arithmetic expression).

Reading INTERVAL variables -- free format: The free format read will obtain the next data field from the input stream on the specified unit, convert it, and store the resulting value in the specified INTERVAL variable. This may be accomplished by a statement of the form

```
CALL INTRDF(UNIT, X)
```

Restrictions:

Only one value is read by each call to INTRDF.

The basic package recognizes units 5 (standard input) and 0 (standard reread).

The Hollerith string read by INTRDF must represent an INTERVAL as defined in APPENDIX 1.

The length of the Hollerith string is limited to the length of the *formatted* read buffer (132 characters in the UNIVAC 1110 version)

The number of significant digits permitted for each endpoint of the interval is limited to the value of ESDMAX (60 in the UNIVAC 1110 version).

The end of an input record does *not* terminate the input stream.

A call to the free-format read routine with a different unit number specified, or *any* call to the formatted read routine (see below), will terminate the input stream on the current unit. Once the input stream has been terminated, INTRDF begins a new input stream with a new record.

INTRDF will read only from units designated as read units by the information in UNITBL (see Appendix 4).

Options:

The standard unit designations may be changed by altering UNITBL, which is located in the COMMON block INTCOM. (This change would normally be accomplished at the time of adaptation, however.)

Descriptions of additional I/O units may be entered in UNITBL. To do this, NUNITS must be changed to allow the I/O routines to scan all information in the table (increase NUNITS by 1 for each unit added). Each unit description consists of a row of UNITBL: Column 1 is the unit number; Column 2 is the length of the record (limited to the length of the free format read buffer, which is 132 characters in the 1110 [illegible]); Column 3 is the number of characters in each record to be scanned; Column 4 is 0 if the record does not contain a carriage control character and 1 if it does; Column 5 is 1 if the unit is read only, 2 if write only, and 3 if read/write.

Units may be referenced by their position in the table rather than their logical unit numbers. -1 refers to the unit described in the first row of the table, and so on.

The number of characters of each record which are actually scanned may be changed if desired; this allows one to use only the first 72 characters in a card image, for example. Change UNITBL(I, 3) to the appropriate value to accomplish this.

The end of a record can be treated as the end of the data field by setting UNITBL(I, 3) to one greater than the length of the physical record and inserting a field termination character in the corresponding character of RBUFRE. The length of this buffer must not be exceeded in doing this, however.

The remainder of the current input record may be skipped by setting OLDUNT to -1 prior to calling INTRDF.

Each input record may be echoed on the standard print unit as it is read. To accomplish this, set ECHOS to .TRUE. in COMMON block INTCRD.

Each data field may be echoed on the standard print unit as it is converted. To do this, set ECHOD to .TRUE. in COMMON block INTCRD.

Delimiters for interval data may be altered as desired by changing the appropriate value in ICHR, located in COMMON block INTCCM. Note that this will also affect the Hollerith strings used to assign constant values to interval variables. If a character is duplicated in this array, its interpretation will be governed by its first appearance.

Data fields may be separated by blanks (as many as desired).

Blanks occurring between matching parentheses will be ignored.

Commas within matching pairs of parentheses are regarded as endpoint separators (unless the ICHR array is redefined).

A field may be terminated by

1. A blank which is neither a leading blank nor located between matching parentheses;
2. Any of the characters '$', '#', or '=';
3. A comma occurring outside of matching parentheses;
4. Any nonblank character following a matching right parenthesis; if this is not one of the characters mentioned in (2) or (3) above, then it will be regarded as the first character of the next field.

Any null field or subfield is taken to represent the number 0.

A field containing a single number will be converted to the smallest interval containing that number. The resulting interval may or may not be degenerate.

Reading INTERVAL variables -- formatted: The formatted read routine reads and converts a vector of data; the vector may, of course, be of length 1. The user supplies a format, which is an array of length 3: The first element is the number of data fields in each record; the second element is the number of characters to be skipped before beginning the scan on each field; and the third is the number of characters in each data field. The calling sequence is

CALL INTRD (UNIT, FMT, A, N)

UNIT is the unit number, FMT is the format as described above; A is the

first location of the vector into which the data is to be read; and N is the length of the vector.

Restrictions:

The basic package recognizes units 5 (standard input) and 0 (reread).

The contents of each field must represent an INTERVAL as defined in Appendix 1. Field termination characters are not permitted.

The length of each record is limited to the length of the formatted read buffer.

The number of significant digits permitted for each endpoint is limited to the value of ESDMAX.

INTRD will read only from units designated as read units in UNITBL.

Options:

The standard unit designations may be changed as in INTRDF.

I/O units may be added to UNITBL; see discussion of INTRDF.

Units may be referenced by their positions in UNITBL as discussed under INTRDF.

The number of characters in a record which are actually used may be altered as discussed under INTRDF.

Input data may be echoed as discussed under INTRDF.

Delimiters for interval data may be altered as discussed under INTRDF.

Note: Any of the above modifications will affect both of the read routines.

Blanks may be embedded in the field; they will be ignored.

The use of parentheses is optional.

A comma will be interpreted as an endpoint separator regardless of whether parentheses are used.

If information is read from a unit on which a carriage control character is indicated, the first character of the record will be ignored.

Computing with INTERVAL variables: Expressions involving INTERVAL variables are written in standard FORTRAN syntax, just as though INTERVAL were a standard FORTRAN data type. A list of the operations and functions available in this package may be found in Appendix 2.

The list of operations and functions available includes all appropriate ANSI standard FORTRAN operations and functions -- i. e., all which have natural

interval extensions. Other operators and functions peculiar to interval arithmetic are implemented; examples include the union of two intervals, the midpoint of an interval, and various measures of the size of an interval. Relational operators are also implemented, but these take on different meanings in the context of interval arithmetic, and therefore have been given slightly different operator symbols.

Restrictions:

Variable and subprogram names beginning with INT or BPA should be avoided to preclude conflicts with the package.

If COMMON block declarations are included in the program for the purpose of exercising various options, then the variable names associated with these COMMON blocks must likewise be avoided.

Options:

Mixed mode expressions are permitted, although their use is discouraged due to the high probability of introducing hidden error. For example, the expression

```
Y = 0.1 * X
```

where X and Y are INTERVAL variables, will not yield a correct value for Y: 0.1 will first be converted to REAL by the compiler, and AUGMENT will then cause that REAL number to be converted to a degenerate interval which will not contain .1 (unless the computer base is decimal-related); multiplication will then take place using the erroneous interval. If mixed mode expressions are used, INTERVAL takes precedence over REAL, INTEGER, and DOUBLE PRECISION data types. The data type to be used in evaluating each subexpression is determined by normal heierarchy rules.

New special function routines may be added. If this is done, the following conventions and rules should be observed in order to preserve the integrity of the package:

1. The name of the new routine should begin with INT. Up to three more characters may be appended to form the name of the routine. Conflicts with existing names must be avoided.
2. The calling sequence for the new routine should adhere to the standards of the package: the arguments are listed first, followed by the result. Every effort should be made to minimize the amount

of information passed.

3. It is the user's responsibility to insure that the endpoints of the result interval are computed and rounded correctly. All existing package routines are available to the user in writing the new routine, just as they are in writing an applications program.
4. If the new routine is to be invoked by the AUGMENT precompiler, the description deck for AUGMENT must be modified to include the new routine. Consult [3] for details.
5. If errors are possible in the new routine, INTRAP should be used to handle the error message rather than including an error printout in the new subprogram. This is done as follows:
   a. The suffix of the name of the new routine is stored in the next available location in the NAMES array.
   b. Appropriate values to indicate the data types of the arguments to the new routine are stored in TYPA, TYPB, and TYPR. Note that it is assumed that the routine will have at most two arguments and one result.
   c. The INTCOM declarations should be included in the new routine. If any argument or result is of a type other than INTERVAL or EXTENDED, a variable of the appropriate type must be defined and EQUIVALENCEd to TA, TB, or TR respectively; storage of the data in the COMMON block is then done by using the variable name of the appropriate type.
   d. INTRAP should be called with the statement

      CALL INTRAP

      immediately prior to the RETURN statement.
   e. Immediately prior to the call on INTRAP, the values of the arguments must be stored in TA and TB and the value of the result must be stored in TR. INTFLT must be set to the appropriate fault indicator (If no existing fault value is appropriate, a new fault value will need to be defined and INTRAP will have to be modified to provide the proper printout -- this constitutes a major modification of the package). Finally, the value of ID must be set equal to the index of

the name of the new routine in the NAMES array.

Writing INTERVAL variables: The write routine will convert a vector (possibly of length 1) of INTERVAL variables to external format and write it on the specified unit according to the given format. The format is now an integer array of length 4: the first three elements of the array are the same as for the formatted read, except that unused characters between fields are filled with blanks; FMT(4) is a Hollerith carriage control character (' ' or '0' in the 1110 implementation) for use where appropriate. The calling sequence is

CALL INTWR (UNIT, FMT, A, N)

where the arguments are as described in the formatted read.

The external representation of each interval is guaranteed to contain the interval, and is the smallest interval representable in the given format which does so.

Restrictions:

The basic package recognizes units 6 (standard print unit) and 1 (standard punch unit).

The length of each record is limited to the length of the write buffer (132 characters in the UNIVAC 1110 implementation).

The width of each data field specified by the format must be at least great enough to permit the package to convert one significant digit. In the 1110 version, this is 15 digits, assuming a 2-digit exponent. Add two characters for each digit of exponent above 2.

The number of significant digits allowed for each endpoint is limited to the value of ESDMAX (60 in the 1110 implementation).

INTWR will write only on units designated as write units in UNITBL.

Note: If any of the restrictions are violated, INTWR will use a standard format and/or the standard print unit, as necessary, to insure that the designated information is written out in *some* form.

Options:

The standard unit designations may be changed (see INTRDF).

I/O units may be added to UNITBL (see INTRDF).

Units may be referenced by their positions in UNITBL (see INTRDF).

Delimiters for interval data may be changed by altering the appropriate values in OCHR, located in COMMON block INTCCM.

Carriage control specifications may be changed by altering the appropriate element of UNITBL.

Errors: The package is designed to detect all errors as they occur. The INTRAP routine will print an error message and halt the computation except in those cases where a viable alternative exists. Those cases are few indeed; they comprise arithmetic underflows (where the offending value is set equal to zero or to the properly signed number of smallest magnitude, as appropriate) and errors occurring on output (where the write routine uses standard modes of output in lieu of the erroneous information). In the former case, computation proceeds without notice to the user; in the latter case, a message is produced on the standard print unit after output is complete.

The possible errors and the response to each are listed in Appendix 3.

Restrictions: None.

Options:

The default response to any fault except 32 (Conversion array overflow) may be changed. To alter the response to Fault number I, change the value of MON(I+1) as desired. This array is located in INTRCM. MON(33) should never be changed, since this could result in looping under some circumstances.

INTRAP may be used as a trace routine of sorts by changing the response to a successful operation [MON(1)]. This will cause INTRAP to produce output even after a successful operation; however, this output is, of course, limited to those routines which call INTRAP. In cases where errors are not possible, INTRAP is usually bypassed.

Producing an object program: The generation of an object program is a two-step process.

1. Use AUGMENT to translate the source program into a FORTRAN program compatible with the host FORTRAN compiler. This can be accomplished with a run stream of the following type (see Appendix 7):

```
invoke AUGMENT
[description decks for INTERVAL and BPA (see Appendix 5)]
*BEGIN
```

[source program]

*END

AUGMENT will write the translated program on unit 20.

2. Compile the output of AUGMENT using the standard FORTRAN compiler; then load and execute as with any FORTRAN program. The user must insure that the INTERVAL package library is available to the linkage editor, and that the BLOCK DATA modules are included with the program by the linkage editor.

The run stream for the UNIVAC 1110 version at the University of Wisconsin is shown in Appendix 7. A sample program is provided in Appendix 6.

2.2 Using the package without benefit of AUGMENT:

If necessary, the INTERVAL package may be used directly, without using AUGMENT to preprocess the source code. If this is done, certain of the things that AUGMENT would normally take care of must be done instead by the user. In this section, we indicate the changes that are necessary in the previous instructions if AUGMENT is not available.

Declaring INTERVAL variables: The number of words of storage that must be reserved for an INTERVAL variable will be twice the number of words needed to store each endpoint. In the UNIVAC 1110 implementation, each INTERVAL variable requires two words of storage. For a particular installation or application, however, the package may have been revised to produce greater accuracy, so the local documentation should be consulted.

The package assumes that the words containing an INTERVAL number will occupy consecutive locations in storage.

The standard data type used to reserve storage for INTERVAL variables is irrelevant so long as consistency is maintained. COMPLEX or DOUBLE PRECISION may be used for simplicity whenever an INTERVAL variable occupies two words; however, to preserve flexibility, AUGMENT has been instructed to declare INTERVAL variables as REAL arrays of length 2. If the latter convention is used, a dimensioned array of INTERVAL variables must be declared as a REAL array of one more dimension, the first dimension always being 2.

If X is an interval variable, Y is an interval vector of length 10, and Z is a 10×20 interval matrix, they could be declared in either of the

following ways:

```
REAL X(2), Y(2, 10), Z(2, 10, 20)
COMPLEX X, Y(10), Z(10, 20)
```

Assigning constant values to INTERVAL variables: This must now be accomplished by a call on the subroutine INTASG. Depending on the data type used to declare the INTERVAL variables and the whims of the compiler, it may be necessary to refer to, say, X(1) rather than X. The rules for forming the Hollerith strings are the same whether or not AUGMENT is used.

If intervals are declared as REAL arrays of length 2, then in order to assign the value of (.1, .1) to X, one would write

```
CALL INTASG(9H(.1, .1)$, X(1))
```

while if intervals are declared as COMPLEX, one could write

```
CALL INTASG(9H(.1, .1)$, X)
```

Restrictions: No change.

Options: No change.

Reading INTERVAL variables -- free format: No change, unless the compiler requires explicit subscripts for dimensioned arrays and INTERVAL variables are declared as arrays. In that case, the subscript must be furnished.

Reading INTERVAL variables -- formatted: See comments for free format read.

Computing with INTERVAL variables: Here, several pitfalls await the unwary user. First, of course, is the necessity of parsing the arithmetic expressions. While this is of vital importance, we assume that the user knows how to do it correctly.

Next is the necessity of expressing INTERVAL variables in a form compatible with the declarations and the restrictions of the compiler. We have touched on this above, and will not iterate here.

Third is the problem of making sure that we are doing exactly what we want to do. For example, if we wish to multiply the interval X by 2.0, we can not write

```
CALL INTMUL(X, 2.0, Y)
```

since INTMUL would multiply X by an "interval" whose left endpoint was 2.0 and whose right endpoint was whatever happened to be lying around in the next storage location. The package, of course, has no way of detecting such blunders; the user alone is responsible for avoiding them.

Thus the user is responsible for employing explicit data type conversions where they are needed, and insuring that all evaluations of expressions are performed using the correct data types.

Restrictions: No change.

Options: No change.

Writing INTERVAL variables: See comments for free format read.

Errors: No change.

Producing an object program: The AUGMENT phase will, of course, not apply. The remaining remarks are valid.

2.3 Using BPA variables in the source program:

The data type of the endpoints of an interval, and of the results of certain INTERVAL functions, is a nonstandard type named BPA. If BPA format is the same as REAL, computations with these quantities can usually be executed using REAL arithmetic. If a REAL variable is set equal to a BPA expression, AUGMENT will perform an automatic type conversion; if AUGMENT is not used, BPACBR should be called to effect this conversion. If mixed mode expressions involving BPA and standard data types are written, AUGMENT will cause the expression to be evaluated using BPA arithmetic, converting the result to a standard data type if necessary.

Computation using BPA routines may be included in the source program, either implicitly, as above, or explicitly. *The user is cautioned that most BPA routines require that a rounding option be stored in BPACOM prior to invoking the routines (see pp. 30 - 34 and Appendix 2).* Thus, for example, in order to add BPA numbers A and B, using upward-directed rounding, and store the result in C, one would need the following statements in the program:

```
COMMON /BPACOM/ ... (declarations from Appendix 4)
BPA A, B, C
 .
 .
 .
OPTION = RDU
C = A + B
```

If OPTION is not set explicitly, it will have whatever value it may previously have had, and the rounding may not be as expected. This could result in erroneous results.

3. Theoretical basis:

The standard reference work in Interval Analysis is the book by Moore [12]; the reader is referred to that book as a supplement to the following discussion.

The underlying set for interval arithmetic is the set of all closed intervals on the real line $\mathbb{R}^1$. An element of this set is denoted by $[a, b]$, where a and b are real numbers with $a \leq b$. This set can be given a metric topology by defining $d([a, b], [c, d])$ to be $\max(|c - a|, |d - b|)$.

An interval of the form $[a, a]$ is said to be a *degenerate* interval; this terminology arises from the topological definition of a degenerate set as being a set consisting of a single point. The set of real numbers may be identified with the set of degenerate intervals by the map $\iota: a \to [a, a]$.

The four basic arithmetic operations are defined over the space of intervals (except, of course, for division by an interval containing zero). The abstract definition is as follows: If $[a, b]$ and $[c, d]$ are intervals, and $\circ$ is one of the above operations, then

$$[a, b] \circ [c, d] = \{x \circ y \mid x \in [a, b], y \in [c, d]\}, \qquad (3.1)$$

where $\circ$ on the right-hand side of the equation is the corresponding operation in the real numbers. Elementary topological considerations guarantee that these sets are again intervals, implying that the space of intervals is closed under the arithmetic operations as defined above.

Definition (3.1) is theoretically correct, but computationally useless. However, the endpoints of the result intervals may be calculated directly from the endpoints of the operand intervals by using the following formulae:

$$\left.\begin{aligned}
[a, b] + [c, d] &= [a + c, b + d] \\
[a, b] - [c, d] &= [a - d, b - c] \\
[a, b] \times [c, d] &= [\min(a \times c, a \times d, b \times c, b \times d), \\
&\qquad \max(a \times c, a \times d, b \times c, b \times d)] \\
[a, b] \div [c, d] &= [\min(a \div c, a \div d, b \div c, b \div d), \\
&\qquad \max(a \div c, a \div d, b \div c, b \div d)], \\
&\qquad \text{provided } 0 \notin [c, d].
\end{aligned}\right\} \quad (3.2)$$

Again, the operations on the right-hand sides are real operations.

The computational efficiency of multiplication and division can be enhanced by determining in advance, in most cases, which products or quotients to use. This can be done via the case analysis shown in Table 3.1; a similar case analysis may be found in [5]. Note that for multiplication there is no need to

TABLE 3.1

SIGN OF a	b	c	d	PRODUCT [a, b] × [c, d]	QUOTIENT [a, b] ÷ [c, d]
+	+	+	+	[a × c, b × d]	[a ÷ d, b ÷ c]
+	+	0	+	[0, b × d]	Undefined
+	+	-	+	[b × c, b × d]	Undefined
+	+	-	0	[b × c, 0]	Undefined
+	+	-	-	[b × c, a × d]	[b ÷ d, a ÷ c]
-	+	+	+	[a × d, b × d]	[a ÷ c, b ÷ c]
-	+	0	+	[a × d, b × d]	Undefined
-	+	-	+	[min(a × d, b × c), max(a × c, b × d)]	Undefined
-	+	-	0	[b × c, a × d]	Undefined
-	+	-	-	[b × c, a × c]	[b ÷ d, a ÷ d]
-	-	+	+	[a × d, b × c]	[a ÷ c, b ÷ d]
-	-	0	+	[a × d, 0]	Undefined
-	-	-	+	[a × d, a × c]	Undefined
-	-	-	0	[0, a × c]	Undefined
-	-	-	-	[b × d, a × c]	[b ÷ c, a ÷ d]

Case analysis for multiplication and division

distinguish between zero and nonzero endpoints, although it may be advantageous to do so in certain contexts.

Other functions of one or more real variables can be extended to interval-valued functions of interval variables in the same manner. The theoretical definition is entirely analogous to the above: if f is a real-valued function of a real variable, its united interval extension F is defined by

$$F([a, b]) = \{f(x) \mid x \in [a, b]\}. \tag{3.3}$$

If f is defined and continuous on $[a, b]$ then $F([a, b])$ will again be an interval. The definition of an interval-valued function of several interval variables is entirely analogous.

As in the case of the arithmetic operations, the basic definition of the united interval extension of a real function is computationally unsatisfactory; we need a formula for deriving the endpoints of $F([a, b])$. Elementary topological considerations guarantee that these endpoints will be the images under f of *some* points of $[a, b]$; if we can determine these points, we will then be able to formulate a computationally useful definition of F.

The points in [a, b] which map to the endpoints of F([a, b]) under the function f (we call these points the *preimages* of the endpoints) will, of course, depend on f itself. If f is a monotone function, then it is clear that f(a) and f(b) will be the endpoints of F([a, b]) -- in the given order if f is monotone increasing, and in the opposite order if f is monotone decreasing. For example, if f is the exponential function exp, then we have

$$EXP([a, b]) = [\exp(a), \exp(b)]. \tag{3.4}$$

For functions which are not monotonic, case analyses will be required. The usual method of evaluating these functions is to partition the domain of f so that f is monotonic on each portion of the domain. The function is then evaluated on each partition of the argument interval, and the value of the function is then taken to be the union of the values on the individual partitions. For example, if f is the hyperbolic cosine (cosh) function, the domain is partitioned into two sub-domains: $[-\infty, 0]$ and $[0, +\infty]$. Note that cosh is monotone decreasing on the first sub-domain, and monotone increasing on the second. We then evaluate COSH on the intersection of each of these sub-domains with [a, b], and finally take the union of the two resulting intervals (one of which may be empty). In this case, as in most situations, it is more convenient to derive explicit formulas for the function based on the nature of these intersections. Thus, we have

$$COSH([a, b]) = \begin{cases} [\cosh(b), \cosh(a)] & \text{if } b < 0 \\ [1, \max\{\cosh(a), \cosh(b)\}] & \text{if } 0 \in [a, b] \\ [\cosh(a), \cosh(b)] & \text{if } a > 0 \end{cases} \tag{3.5}$$

Other methods of function evaluation are possible, too. Some functions (square root, for example) can be evaluated by applying the interval analysis version of Newton's method. This consists of starting with an initial interval of sufficiently small length which is known to contain the root, and using the contractive properties of the Newton iteration to shrink the interval to an acceptably small length. The details of this method are presented in [12]. The convergence of this method is entirely analogous to that of its real counterpart, and the intervals thus obtained are mathematically rigorous.

For rational functions, direct evaluation using the basic arithmetic operations is possible, although the result may be unduly pessimistic due to the *dependency problem*. If the same interval appears in two different places

in an expression, the dependency will not be recognized, and the result of the calculation will be the same as it would have been had the two intervals been completely independent. The simplest example of this is the calculation of $X \times X$. If X is taken to be the interval $[-2, 2]$, then by definition of multiplication we would calculate $X \times X \equiv \{x \times y \mid x \in X, y \in X\} = [-4, 4]$. If the dependency were taken into account, we would compute $X \times X = X^2 \equiv \{x^2 \mid x \in X\} = [0, 4]$.

Relational operators are often useful even though the space of intervals is not totally ordered. The lack of total ordering, however, precludes any "correct" definition of relational operators. Any of several viewpoints may be useful, depending on the context.

One definition that seems computationally useful, even though it does not induce even a partial ordering on the space of intervals, arises from the philosophy that intervals are bounds on exact (but undetermined) real numbers; hence a given relation should hold between two intervals if and only if the corresponding relation holds between every pair of real numbers, the first taken from the first interval and the second from the second interval. This is expressed by the following set of definitions:

$$\left.\begin{array}{ll}
[a, b] = [c, d] & \text{if and only if } a = b = c = d \\
[a, b] \neq [c, d] & \text{if and only if } [a, b] \text{ and } [c, d] \text{ are disjoint} \\
[a, b] < [c, d] & \text{if and only if } b < c \\
[a, b] \leq [c, d] & \text{if and only if } b \leq c \\
[a, b] > [c, d] & \text{if and only if } a > d \\
[a, b] \geq [c, d] & \text{if and only if } a \geq d
\end{array}\right\} \quad (3.6)$$

One way of inducing a partial ordering on the space of intervals is the following:

$$\left.\begin{array}{ll}
[a, b] = [c, d] & \text{if and only if } a = c \text{ and } b = d \\
[a, b] \neq [c, d] & \text{if and only if } a \neq c \text{ or } b \neq d \\
[a, b] < [c, d] & \text{if and only if } b < c \\
[a, b] \leq [c, d] & \text{if and only if } a \leq c \text{ and } b \leq d \\
[a, b] > [c, d] & \text{if and only if } a > d \\
[a, b] \geq [c, d] & \text{if and only if } a \geq c \text{ and } b \geq d
\end{array}\right\} \quad (3.7)$$

Yet another way of inducing a partial ordering is by using the subset relationship: equality and nonequality would be as in (3.7), and inequality

would be containment -- for example, [a,b] < [c,d] would mean that [a, b] is a proper subset of [c, d].

From the standpoint of extending the usual ordering of the real numbers to the space of intervals, none of these definitions is completely satisfactory. Definition (3.6) has the unpleasant property that if [a, b] and [c, d] are non-degenerate intervals with a = c and b = d, then neither [a, b] = [c, d] nor [a, b] ≠ [c, d] is true. This is consistent with the hypotheses of the definition, since if these intervals were obtained by different computations, they might represent the same real number, but they need not do so. Definition (3.7) allows the possibility that if [a, b] ≤ [c, d] is true, [a, b] < [c, d] and [a, b] = [c, d] may still both be false; and the definition based on the subset relationship implies that two degenerate intervals are either equal or incomparable. One must, therefore, select the most nearly appropriate definition for the given application.

Forming new intervals: New intervals may be formed either from real numbers or by applying set-theoretic operations to other intervals. We note the following operations:

Compose is a function of two real variables which yields an interval. The arguments of this function are the left endpoint and the right endpoint of the interval, respectively.

Union is a function of two interval variables which yields the smallest interval containing the set theoretic union of the two argument intervals.

Intersect is a function of two interval variables which yields the largest interval common to the two argument intervals. The result may, of course, be empty; this is regarded as an error.

Measures of intervals: In many applications, it is necessary to have information about the character of an interval. In [13], Ris defines the following functions. Note that, in some of these functions, the absolute value is used -- this is the united interval extension of the real absolute value, given by ABS([a, b]) ≡ {abs(x) | x ∈ [a, b]}.

Length of [a, b] is the real number b - a.

Half-length of [a, b] is the real number (b - a)/2.

Midpoint of [a, b] is the real number (b + a)/2.

Size of [a, b] is the real number (|a| + |b|)/2.

Supremum of [a, b] is the real number b.

Infimum of [a, b] is the real number a.

Magnitude of [a, b] is defined to be SUP(ABS([a, b])).

Mignitude of [a, b] is defined to be INF(ABS([a, b])).

Pivot of [a, b] is defined to be $\sqrt{MAG([a, b]) \times MIG([a, b])}$.

Sign of [a, b] is $\begin{cases} -1 & \text{if } b < 0 \\ 0 & \text{if } a \leq 0 \text{ and } b \geq 0 \\ 1 & \text{if } a > 0 \end{cases}$

Logical operators: These operators provide the means for the user to test intervals for specified properties.

Bad is a logical function of one interval [a, b]. Its value is TRUE if $a > b$, and FALSE otherwise.

OK is a logical function of one interval [a, b]. Its value is TRUE if $a \leq b$, and FALSE otherwise.

Element of is a logical function of a real variable x and an interval [a, b]. Its value is TRUE if $x \in [a, b]$; FALSE otherwise.

Subset is a logical function of two interval variables [a, b] and [c, d]. Its value it TRUE if [a, b] is a subset of [c, d], and FALSE otherwise.

Superset is a logical function of two interval variables [a, b] and [c, d]. Its value is TRUE if [c, d] is a subset of [a, b], and FALSE otherwise.

We turn now to the question of manipulating real numbers. Since intervals are expressed as pairs of real numbers, it is clear that we will need to be able to perform computations in the real number field if we are to be able to perform any of the interval operations. Of course, we will eventually want to limit our attention to those computations that can be performed on a computer; however, it is desirable to define the theoretical basis for calculations with abstract real numbers first so that we can put the finite-precision calculations on a rigorous mathematical basis.

Real numbers can be theoretically realized as (possibly infinite) radix polynomials. If $\beta \geq 2$ is an integer (β represents the number base of the radix polynomial) and x is a real number, we can write

$$x = \sum_{i=-\infty}^{\infty} a_i \beta^i, \quad 0 \leq a_i \leq \beta - 1, \tag{3.8}$$

where all but a finite number of the a_i with positive index are zero. With suitable restrictions, the representation is unique. A rule can be given for the addition of radix polynomials (see, *e. g.*, [6]); the rule is not an algorithm because in general it does not terminate. Thus, even though we know the rule for addition, we can not normally complete an addition, any more than we can write down the radix polynomial for π. The purpose here is to study the properties of a theoretical realization which is so closely related to the computer manipulations that the transition can be made relatively easily. For example, if we restrict our attention to finite radix polynomials (in which all but a finite number of the a_i are zero), the rule for addition of infinite radix polynomials becomes an algorithm.

Since we will be using floating point arithmetic for operations on the endpoints of intervals, we must consider not the general radix polynomials given by (3.8), but normalized floating point radix polynomials, which are of the form

$$x = \beta^e \sum_{i=-\infty}^{-1} a_i \beta^i, \quad 0 \leq a_i \leq \beta - 1, \quad e \text{ an integer} \tag{3.9}$$

where $a_{-1} \neq 0$ unless $x = 0$. The theory of floating point radix polynomials is accessible in the literature (see, for example, [6], [7], [11], and [16]); we shall not belabor it here.

We now come to the question of how interval arithmetic is to be approximated on the computer. We want this approximation to be the best possible in some sense; this means that it should be a *faithful approximation* in the sense of [18]. Briefly, this means that the approximation must have the following two properties:

1) If an abstract interval can be represented exactly on the machine, then the approximation to that interval must have the same value as the abstract interval.

2) If F is an abstract interval function and $\tilde{F}$ is its machine approximation, then the value of $\tilde{F}$ applied to machine representable

intervals must be the same as would be obtained by applying F to the corresponding abstract intervals and then applying the approximation rule to the result.

These two properties are extremely strong; indeed, the second property virtually precludes the use of hardware floating point arithmetic for endpoint calculations, since it is not uncommon to find that, although (for example), a, b, and a + b are all machine representable, the floating point addition will not yield a + b, but only some approximation to the sum.

In some cases, we find that adhering strictly to the conditions of a faithful approximation would be prohibitive by reason of cost. In such cases, we may elect to make some concessions; however, we insist that any deviation from a faithful approximation must be purposeful rather than accidental.

Since the purpose of interval arithmetic is to provide rigorous bounds on computed results, we insist on a third property of the approximation:

3) An abstract interval must be contained in its approximation.

Indeed, if the approximation is to be faithful, an abstract interval must be approximated by the smallest approximate interval containing it. Note that the approximation to a degenerate interval may be nondegenerate, and that if a' is the closest finite precision approximation to a, the interval [a', a'] need not be a valid approximation to the abstract interval [a, a].

Property (3) leads naturally to the concept of directed roundings, which was developed in [8]; a short discussion may be found in [17]. Briefly, a rounding ρ is *downward directed* if $\rho(a) \leq a$, and *upward directed* if $\rho(a) \geq a$. Of course, if a is representable in the data format being used, we would insist on having $\rho(a) = a$. From this, we see that downward directed roundings should be used in the calculation of the left endpoint of an interval, and upward directed roundings in the calculation of the right endpoint. (Other roundings are also possible; in this package, numbers of the type of interval endpoints may also be rounded to the nearest representable number, rounded toward zero [truncated], or rounded away from zero. See [17] for details.)

Out-of-range numbers pose another problem. If, for example, the right endpoint of an abstract interval is larger than the largest positive machine number, then no machine number is a valid bound for it. The attempt to approximate such an interval, be it initial data or the result of an operation in the package,

must result in an error indication if the package is to be trustworthy.

We recognize three types of exponent range faults: overflow, which means that the exponent is too large, but there is a representable number which is a valid bound for the result, using the specified rounding; infinity, which means that the exponent is too large and no valid bound exists; and underflow, which means that the exponent is too small. In the latter case, either zero, the smallest positive normalized representable number, or its negative will be a valid bound.

Thus we see that we need two more properties in our approximate floating point arithmetic scheme: the arithmetic must be capable of directed roundings, and the arithmetic algorithms must be capable of detecting out-of-range results and providing the means by which the package can control the response to these faults. Even when computer hardware is capable of producing a faithful approximation to real arithmetic, it is unlikely that it has these additional properties; we know of none that has. There is some current research aimed at creating more hospitable architecture; see, for example, Lang and Shriver [9] and Ris [14].

In Section 4, we discuss in detail how we have dealt with these problems in this package.

The remaining problem is that of converting between the internal base (usually some power of 2) and the decimal base, which is used for input data and for expressing the results of the computation. Various aspects of number base conversions have been treated by Matula in [10] and by this author in [15].

The only unusual feature of number base conversions for interval arithmetic is that the left endpoint of an interval must be rounded down and the right rounded up whenever conversion is not exact. This preserves the integrity of the guaranteed bounds. No standard conversion routine known to us is capable of these roundings; thus, as we shall see in the next section, we have provided the package with its own conversion routines.

With this theoretical foundation, we are now in a position to discuss the design and implementation of the package.

4. Design and implementation of the package:

We sought to make the INTERVAL arithmetic package *complete, accurate, convenient to use, fail-safe,* and *transportable.*

Completeness: The package includes interval extensions of all appropriate ANSI Standard FORTRAN [1] operations and functions, along with some which are not ANSI standard but are normally implemented in the FORTRAN language anyhow. In addition, two sets of relational operators [those defined in (3.6) and (3.7)] are included, as are all of the special interval functions described on Pages 18 and 19. Finally, input/output routines and conversions between INTERVAL and standard data types (where appropriate) are provided. A complete list of the capabilities of the package may be found in Appendix 2.

Accuracy: The arithmetic operations and number base conversions were written from scratch in order to obtain the greatest possible accuracy and to maintain control over the exponent range faults. In every case, the result of one of these operations is the nearest number representable in the chosen format which provides the desired bound on the true result. For the mathematical functions, we obtained bounds by first evaluating the functions in higher precision arithmetic, then adding or subtracting a relative error term based on the accuracy of the higher precision routines, and finally taking as the result the nearest number in the chosen format which provides the desired bound on the modified higher precision result. The accuracy information for the higher precision functions was obtained from the supplier of that software; such information may not be completely rigorous (even when available), and this naturally detracts from the rigor of the package. In the 1110 version, we used a statistical analysis of accuracy, and made further adjustments to increase the probability that the bound we obtained is indeed rigorous. The results obtained in these functions are not always the closest possible to the true result, but in virtually all cases they differ from the best approximation by at most 1 in the least significant digit.

Convenience: By itself, no collection of routines to perform nonstandard arithmetic is really convenient to use. Each operation must be performed by a call on one of the subprograms in the package; this means that the user must parse every expression himself and write his program in what amounts to assembly language. The best that can be done in this setting is to minimize the incon-

venience. To this end, we have kept the package as internally consistent as possible. All entry points to the package bear the prefix INT; routines used by the package itself are prefixed with INT or BPA, according to their level. Thus, by avoiding identifiers beginning with these prefixes, the user may be assured of avoiding conflicts. Calling sequences for the routines in the package are consistent and concise. No information is transmitted which is not absolutely essential to the function being performed; where it seems appropriate, some information is transmitted within the package via COMMON storage. Where the result is a standard FORTRAN data type, the routine is implemented as a function of the appropriate type whose arguments are simply the operands. When the result is a nonstandard data type, the routine is implemented as a subroutine whose arguments are the operands together with the result. In the interest of flexibility, the endpoints of an interval and the results of such functions as midpoint, half-length, etc. are regarded as being a nonstandard data type.

Convenience of use of any nonstandard arithmetic is increased dramatically if an appropriate precompiler is available. This package is specifically designed to be used with the AUGMENT precompiler, which allows the source FORTRAN code to be written as though FORTRAN recognized INTERVAL as a standard data type. In this case, just as above, the user must avoid name conflicts with the package; although the source code will not contain references to most of the routines of the package, the output from AUGMENT, of course, will. In addition, the user must also avoid the function names and operators shown in Appendix 2 (except, of course, those which may be regarded as generic), since these become reserved words in the extension of FORTRAN; in most cases, this should not be an onerous task.

Fail-safe: Errors can occur in many of the operations of the INTERVAL package, just as they can in REAL operations. It is our opinion that errors should not be ignored. Each subprogram in which an error can occur will call the error-handling routine, INTRAP, prior to returning control to the calling program. If no error has occurred, INTRAP will simply return control to the routine which called it. Otherwise, INTRAP takes the action specified in a table which resides in a COMMON block; the response depends on the error which has occurred, but usually includes a print-out which gives the user complete

information on the error. The user may, if desired, alter the response by changing the table.

Transportability: This is closely linked with flexibility of representation. This package is based on three nonstandard data types: BPA (mnemonic for Best Possible Answer), which is the data type of the endpoints of intervals, but is otherwise undefined except in a few primitive routines; INTERVAL, which is defined to be a BPA array of length 2; and EXTENDED, which is used for the evaluation of the mathematical functions. In the UNIVAC 1110 version of the package, the representation of BPA is the same as that of REAL, and EXTENDED is a synonym for DOUBLE PRECISION.

The AUGMENT precompiler is used to extend the definitions of these data types throughout the package. The output of the AUGMENT precompiler is a set of routines which, apart from the arithmetic primitives (which are written in assembly language) conforms as closely as possible to ANSI standard FORTRAN.

There are less than twenty program modules which depend on the representations of BPA and EXTENDED numbers; many of these will need no alteration for most implementations. Adaptation of the INTERVAL package to other hardware and/or data representations is discussed thoroughly in Section 5.

Package structure and routine hierarchies: Broadly speaking, the INTERVAL package uses three hierarchical levels. The topmost of these is, quite obviously, the INTERVAL routines themselves; these are the modules of interest to the user. In evaluating INTERVAL functions and operations, the INTERVAL routines call on the second-level BPA routines, which perform endpoint evaluations with appropriate roundings. Finally, some of the BPA routines, notably those which perform mathematical function evaluations, call on the lowest-level EXTENDED routines. The entire package is, of course, based on the FORTRAN library routines, which we regard as being a part of the host system.

Insofar as possible, the routines in each major level are written in a manner which is independent of the data representations used at the lower levels. Certain assumptions are made concerning the relationships between BPA and EXTENDED data representations, and between these data types and the standard FORTRAN data types; these are detailed in Section 5, and are of concern only when the package is to be modified. The relative independence of the three

levels makes it convenient to change the accuracy of the package by replacing the BPA and EXTENDED routines with other routines of the desired accuracy, or to take advantage of available software or other system features in improving the performance of the package. For example, one may use existing DOUBLE PRECISION routines in the evaluation of mathematical functions (as was indeed done in the UNIVAC 1110 implementation) rather than writing a fresh set of mathematical library routines to support the EXTENDED data type; if directed roundings are available on the host computer, the BPA section of the package can be rewritten to take advantage of this capability.

Within each of the major levels, there are sub-levels; the more complex modules in each major level tend to depend on progressively simpler ones just as the applications programs depend on the package. In this manner, dependencies on data representations are concentrated in a relatively few modules in each major level.

Figure 4.1 shows the hierarchical structure of the INTERVAL and BPA portions of the package; since DOUBLE PRECISION routines are used for EXTENDED calculations, these are not included in the figure. The routines are shown by level and function, and major dependency paths are indicated. For simplicity, minor dependency paths are omitted, as are all dependency paths between non-adjacent levels. In order to save space in the diagram, we have referred to all package modules by the suffix of the module identifier: for example, INTADD is referred to simply as ADD. In most cases, the suffix will be sufficient to indicate the module's function; the largest group of modules for which this may not be the case is the group of conversions. For conversion modules, the first letter of the suffix is normally C, and the next two letters indicate the argument type and the result type, respectively -- the letter will be the first letter of the type name for most data types (standard and nonstandard), and X for INTERVAL. In case of doubt, the function of a module can be determined from the table in Appendix 2.

In Figure 4.1, Level 1 represents the lowest level; i. e., the level which depends on no other module of the given section of the package. A module is assigned to Level i if and only if it depends on one or more modules at Level (i - 1) but on no modules at Level i or greater. All BPA modules are at a lower position in the hierarchy than any INTERVAL module.

FIGURE 4.1
HIERARCHIES OF INTERVAL ROUTINES

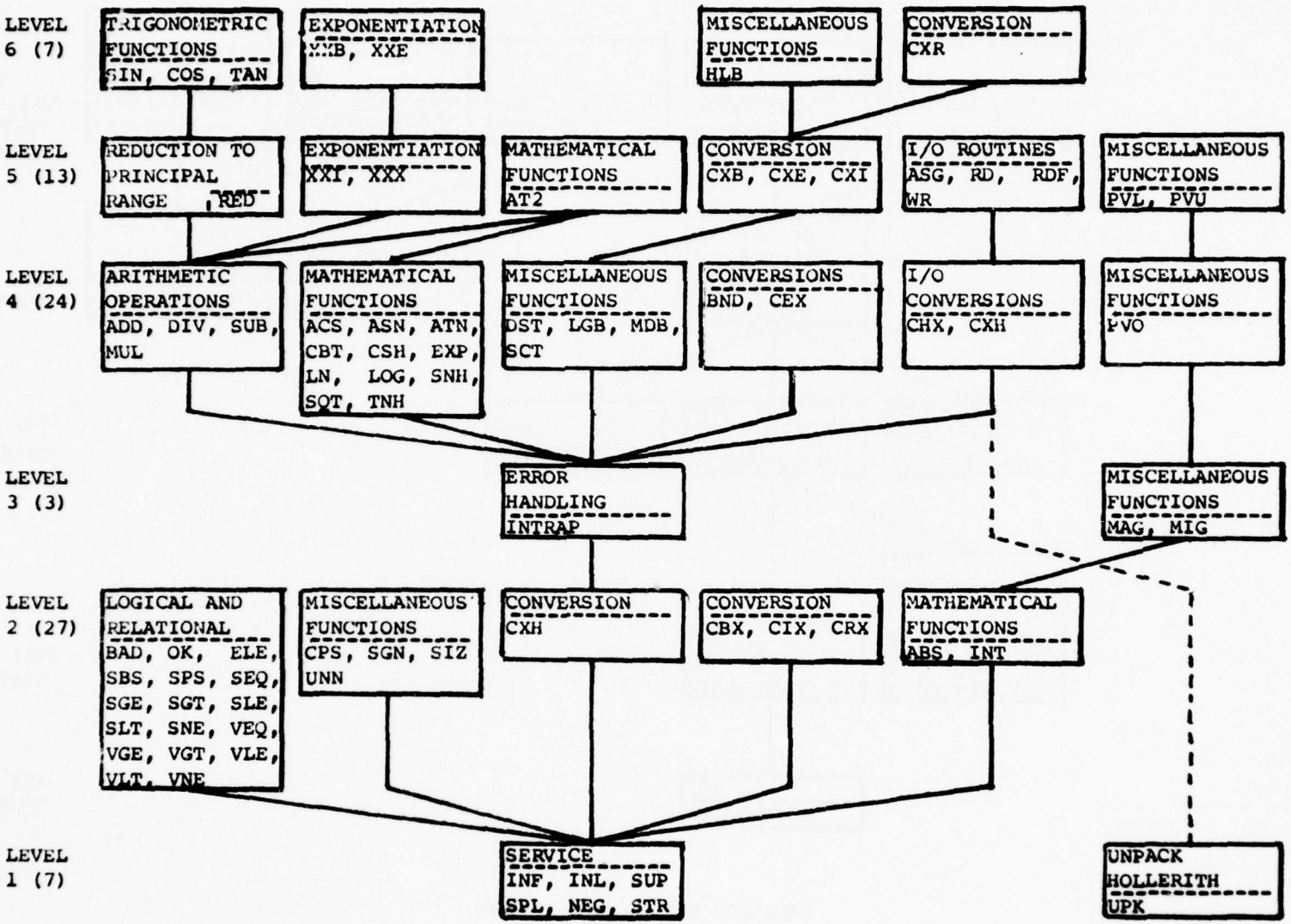
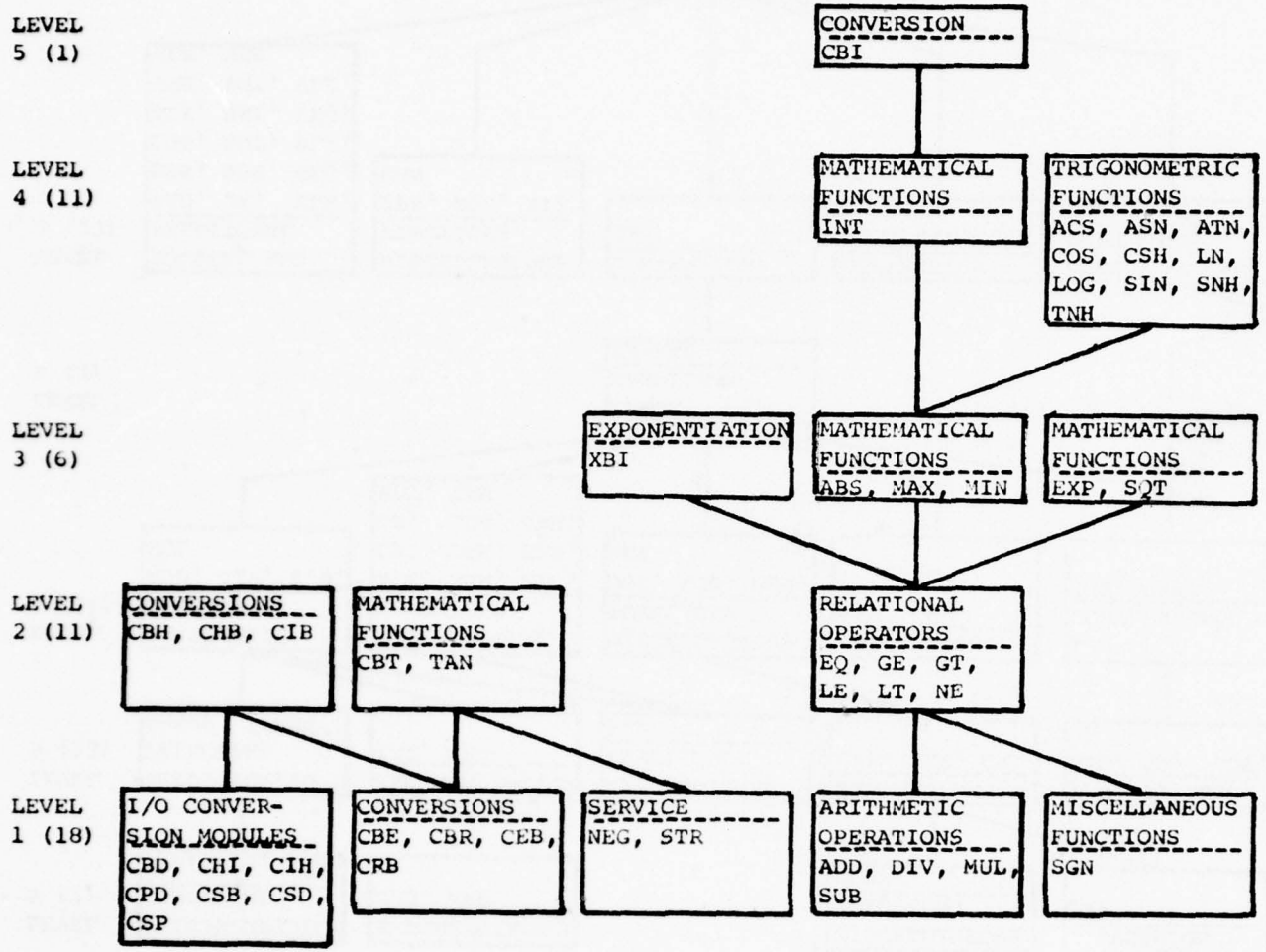


FIGURE 4.1 (CONTINUED)
HIERARCHIES OF BPA ROUTINES



Communication: Communication between the user's program and the INTERVAL package is accomplished entirely by passing parameters in the FORTRAN calling sequences for the INTERVAL routines.

Within the INTERVAL portion of the package, all communication is via the calling sequences *except for communication with INTRAP, which is entirely via COMMON blocks*. Each routine which calls INTRAP must store the following information in the COMMON block INTCOM prior to executing the CALL: INTFLT must be set to one of the values from Appendix 3, reflecting the success or failure of the computation; ID must be set to the appropriate index to identify the calling routine as specified in Appendix 4, description of COMMON block INTRCM; and the arguments and result of the calculation must be stored in TA, TB, and TR respectively. If any of the latter three quantities are of a data type other than INTERVAL, variables of the appropriate type must be EQUIVALENCEd to the proper identifiers in INTCOM and the replacement must be made using the identifier of the appropriate type. INTRAP determines the type of each of these quantities by referring to the appropriate table in INTRCM. The only exception to this is the case of Hollerith strings (encountered only in the I/O routines and in INTASG) which are passed to INTRAP in the array HSTR, located in the COMMON block INTCOM.

All communication with the BPA portion of the package involves the COMMON block BPACOM. Prior to calling any BPA routine, the desired rounding must be specified by setting OPTION to one of the values RDU, RDL, RDT, RDN, or RDA, indicating upward-directed rounding, downward-directed rounding, rounding toward zero, rounding to the nearest BPA number, or rounding away from zero, respectively. Numeric values for these options are stored in BPACOM. In addition, if BPACEB is called, the relative accuracy of the EXTENDED number which is to be converted to BPA by that routine must be specified in ACC; this is simply set to the number of digits of the fraction which are known to be valid. Upon return from the BPA routine, BPAFLT will contain an integer which indicates the success or failure of the BPA calculation, according to the table given in Appendix 4, description of COMMON block BPACOM. All other information is passed to the BPA routines via the calling sequences.

In order to simplify modification of the package, information which depends on the representation of BPA and EXTENDED numbers is collected in the COMMON

blocks; this information is, of course, communicated to the modules needing it by including the appropriate COMMON declarations in the program modules.

Implementation of the package: We turn now to the discussion of the implementation of the package. Although the starting point of the design of the package was the user's viewpoint, we adopt a different perspective in implementing it: we begin with the FORTRAN language, and create progressively more powerful extensions until we arrive at the interval arithmetic capability we seek. Thus, the implementation of the package is a bottom-up process, and we describe it as such. We shall use Figure 4.1 as a guide for this description, although we do not follow it blindly, since routines on different levels can sometimes be lumped together for description -- the prime example of this is the set of mathematical function routines, which appear on three different levels in the BPA portion of the package, yet can be described together.

EXTENDED routines: These are higher-precision mathematical function subroutines, used by the BPA mathematical function routines. In the UNIVAC 1110 version of the package, we use the DOUBLE PRECISION library routines from the FORTRAN library. For each EXTENDED routine, information about the relative accuracy of the routine is required; in certain cases (LOG, LOG10, SINH, and TANH) accuracy may decrease near critical points due to cancellation error, and in these cases two parameters must be supplied -- accuracy near the critical point, and accuracy in the rest of the range. The accuracy information consists of the number of digits (in the base of the EXTENDED numbers) which are known to be valid. For the UNIVAC 1110 version, this information was obtained from a statistical analysis provided by the University of Wisconsin Computing Center [2].

BPA routines: These routines form the basis for endpoint calculations in the INTERVAL package. In the UNIVAC 1110 version of the package, BPA format is identical to REAL format, and the major distinction between BPA routines and their REAL counterparts is the capability of performing directed roundings in the BPA routines. Each BPA routine which may need to round its result is given a rounding option in the COMMON block BPACOM; the result is calculated and rounded according to the given option. In other implementations, BPA format may be specified at will, but whatever the format, the directed roundings must be available.

Arithmetic operations: These four operations (ADD, SUB, MUL, and DIV) are the heart of the BPA package. In writing these routines, we used the algorithms given in [17], which yield the best possible approximation to the true result of the operation with the specified rounding. Since these algorithms are available in the open literature, we do not include them here; however, the general method of fault detection can be seen in Figure 4.2.

The SGN function: This function yields an integer result which is +1 if the argument is positive, -1 if the argument is negative, and 0 if the argument is zero. In the 1110 version of the package, we used the fact that BPA and REAL formats are the same, and employed a three-branch IF statement to test the argument.

Service routines: These are the STR routine to perform a replacement of BPA values and the NEG routine to change the sign of a BPA value. We included these routines in the package, although when BPA and REAL formats are the same, they are not necessary; the normal FORTRAN operations on REAL numbers would have the same effect. In the UNIVAC 1110 version, the arguments and results of these routines are declared REAL and the body of each of these routines consists simply of the corresponding FORTRAN statement. However, by judicious preparation of the AUGMENT description deck, AUGMENT can be made to use the REAL counterparts in FORTRAN to accomplish these tasks. If BPA format is not the same as REAL, these service routines may be required components of the package.

Relational operators: These routines implement the relational operators .EQ., .GE., .GT., .LE., .LT., and .NE.; in each case, the algorithm is the same: the second argument is subtracted from the first using BPA arithmetic with rounding away from zero, and the SGN of the result is checked to see if it bears the given relationship with 0. The result is .TRUE. if it does; .FALSE. otherwise. It is important that BPA arithmetic with rounding away from zero be used; if REAL arithmetic were used, underflows might be set to zero, yielding erroneous results, and overflows might even result in the opposite sign for the result!

Exponentiation: This routine raises a BPA number to an integer power, using successive squaring and multiplication. The BPA multiply routine is used to perform the multiplications, and the indicated rounding is performed after each multiplication step. Possible errors include 0.0^{0} and 0.0^{-n}, where n is a positive integer.

Absolute value, Max, and Min functions: These functions are implemented in a straightforward manner, using the relational operators. No errors are possible.

Integer function: This function yields the nearest BPA integer to the argument, rounded in the indicated direction. The argument is first tested to see whether it already is an integer; if so, nothing is done. Otherwise, the argument is made positive, and is then added to the smallest BPA number having no fractional part in the BPA format; the result is then rounded according to specification, the "fudge factor" is subtracted out again, and any necessary end sign correction is made. No errors are possible.

Mathematical functions: This category includes all trigonometric and inverse trigonometric functions, the hyperbolic functions, logarithmic functions, the exponential function, square root, and cube root. The basic algorithm for all of these is the same: the argument is first converted to EXTENDED; the corresponding EXTENDED routine is called to obtain an approximation; and the resulting value is then converted to BPA using the relative accuracy information and the routine which converts EXTENDED to BPA (BPACEB). If the range of the function is bounded, the result is adjusted as necessary; for example, if application of the algorithm yields a value greater than 1.0 for SIN, the result is set to 1.0. In cases where exponent range faults are possible, the argument is checked prior to computation; if the given argument would result in an out-of-range result, the appropriate value is stored in the BPAFLT cell in BPACOM.

Conversions: Conversions between REAL and BPA are straightforward as long as REAL and BPA have the same format; thus, in the 1110 version, BPACRB and BPACBR are essentially replacement operators. On a computer which uses two's complement representation, however, conversion from REAL to BPA must report an error if an attempt is made to convert those REAL numbers whose negatives are not representable; the set of BPA numbers is assumed to be closed under negation. If BPA format is chosen to have greater accuracy than REAL, then conversion from BPA to REAL must be rewritten to check for range faults and set the appropriate value in BPAFLT if they occur.

Conversion from BPA to EXTENDED is straightforward, since EXTENDED is

assumed to be at least as accurate as BPA. Conversion from EXTENDED to BPA, on the other hand, is a more complicated situation. The BPACEB routine uses an accuracy constant (which, if zero, is interpreted to mean that the argument is exact); this is an integer which indicates the number of digits which are valid. BPACEB adds or subtracts a 1 in this digit position, according to the rounding specified, before converting the EXTENDED number to BPA format and rounding the result as indicated. If the argument is identically zero, it is assumed to be exact regardless of the value of the accuracy constant, and no relative accuracy correction is applied. Rounding is accomplished by adding 1 in the least significant digit as required; fault determination is similar to that in Figure 4.2.

Conversion from INTEGER to BPA is accomplished by first converting the INTEGER to EXTENDED, then converting the result to BPA; this provides the proper rounding in the event that the integer is not exactly representable in BPA format. Conversion from BPA to INTEGER is accomplished by first using the INT function, then converting the resulting BPA number to INTEGER using the standard FORTRAN REAL to INTEGER conversion.

Conversion from Hollerith to BPA is a five-step process; most of the steps are designed to be independent of the internal representations of BPA numbers. The first step is to convert the Hollerith string to an array format. Each Hollerith character is converted to its corresponding numerical value using BPACHI, which is merely a table look-up routine. These digits are stored, one per word, in positions 11 and following of the EX array; positions 1 through 10 of that array contain information such as the number base, the location of the radix point, the location of the most significant and least significant digits, the sign of the number, and the exponent of the number (see Appendix 4, description of BPACCM). In the second step, BPACSP is called to convert this source number to a fixed-point number in a base which is a power of 10; this is stored in the array ECX. In the third step, BPACSD is called to convert this fixed-point number in a power-of-10 base to a fixed-point number in a power of the internal base. The fourth step is to employ BPACPD to convert the fixed-point result of BPACSD to a floating point number in the internal base, still in a one digit per word format, and round it. Finally, BPACSB detects range faults and packs the number into BPA format. The first four of these routine
forward conversion routines using standard conversion algorithms; the on

unusual features being the array format of the numbers and the inclusion of an additional digit to indicate the presence of a remainder. BPACSB is a bit more complicated; since it is a primitive which must be written anew for each implementation, we have included a flow chart of it as Figure 4.2.

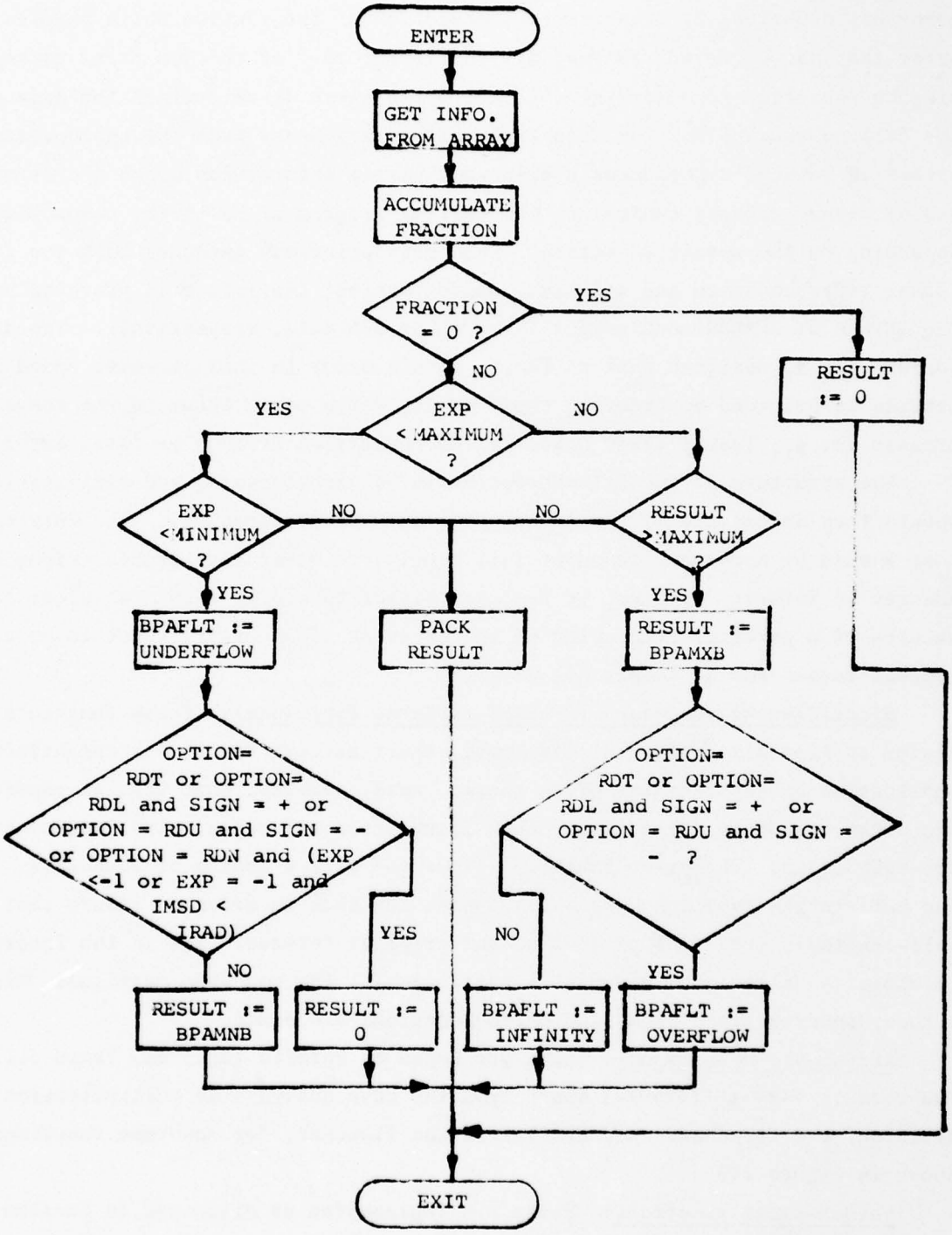
Conversion from BPA to Hollerith is just the inverse of the above process; here, the routines used are BPACBD, BPACSP, BPACSD, BPACPD, and BPACIH in that order. Note that the three "middle" routines in the sequence are the same, regardless of the direction of conversion; these routines determine the source and destination bases from the arrays themselves, and are extremely general.

INTERVAL routines: These are the routines which are visible to the user of the INTERVAL package. An INTERVAL is defined to AUGMENT to be a BPA array of length 2; hence, whatever data representation was used for BPA numbers is carried directly into the INTERVAL portion of the package. As explained in Section 3, operations and functions on intervals are calculated by finding the endpoints of the result, often from the endpoints of the argument intervals. Errors in interval calculations can occur at one or both endpoints of the result, and in some cases other errors can also occur. Any INTERVAL routine in which an error can occur will call the error-handling routine INTRAP prior to returning control to the calling program; INTRAP will test to see whether an error has occurred and take appropriate action if it has.

Service routines: These include four routines to extract and insert values in the left and right endpoints of intervals (INF, INL, SUP, and SPL) respectively, and the NEG and STR routines whose functions are analogous to the corresponding BPA routines. The former are straightforward; they rely on the definition of an interval as a BPA array of length 2 -- the first element of the array being the left endpoint, or Inf, of the interval, and the second being the right endpoint, or Sup. The STR routine is also straightforward, being simply a data-moving routine. The NEG routine is likewise straightforward, but depends on the definition $-[a, b] \equiv [-b, -a]$. These routines call on BPASTR and BPANEG as appropriate.

Logical and relational operators: These routines are straightforward applications of the definitions in Appendix 2; the BPA relational operators are used in determining whether the required relationships hold. No errors are possible in any of these routines. The operators given in (3.6) and (3.7) are implemented.

FIGURE 4.2
FLOWCHART FOR BPACSB



Error handling (INTRAP): This routine 1) checks INTFLT to see whether an error has occurred; 2) branches to the section of the routine which handles the error that has occurred, if any; 3) checks the cell of the MON array corresponding to the error to determine what action to take; 4) determines the name of the calling routine and the data types of its arguments from the appropriate arrays in INTRCM; 5) produces a print-out giving information about the error; and 6) either returns control to the calling program or halts the computation, depending on the specified action. Each data print-out includes both the internal representation and a decimal approximation; the latter is produced by the INTCXH or BPACBH modules for INTERVAL or BPA data, respectively. The data formats are so designed that no faults should occur in this process; added protection is provided by treating those faults which could occur in the conversion process (e. g., insufficient space in the conversion arrays) as fatal errors.

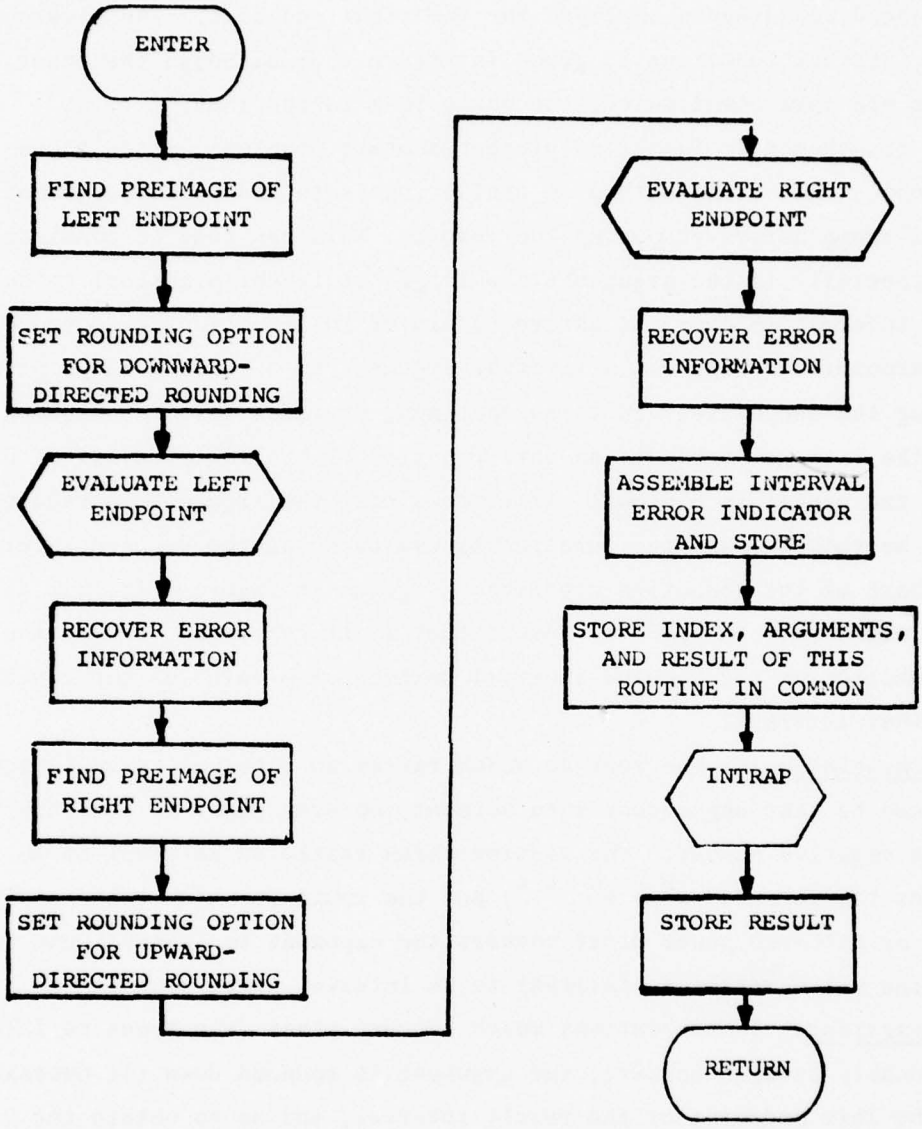
The structure of the INTRAP routine is straightforward, and modifications, should they be necessary, should not pose any serious problems. The only changes that should be necessary to adapt this routine to other host systems might be changes in formats; however, if the user wishes to modify INTRAP to alter the results of a particular calculation in the event of an error, there is no conceptual reason why he should not do so.

Miscellaneous functions (Special INTERVAL functions): These functions are listed in Appendix 2; for the most part, their calculation is a straightforward application of the formulas given there. Note, however, that special roundings are often employed; for example, such functions as distance, size, and length are rounded up. The pivot function allows the user a choice of roundings. The half-length function uses the midpoint function in order to insure that the half-length is great enough so that the original interval lies in the interval [midpoint - halflength, midpoint + halflength]. The compose, magnitude, mignitude, intersection, sign, and union functions are exact.

Arithmetic operations: These are based on Formula (3.2) and Table 3.1. The code is straightforward; apart from the case analysis in multiplication and division, the flowcharts are similar to the flowchart for monotone functions, shown in Figure 4.3.

Mathematical functions: These are implemented as discussed in Section 3; the preimages of the endpoints of the result interval are located and the BPA

FIGURE 4.3
FLOWCHART FOR MONOTONE INTERVAL FUNCTIONS



mathematical functions are used to calculate the values of the endpoints. Of course, downward-directed rounding is specified for the left endpoint, and upward directed rounding is employed for the right endpoint. The flow chart for monotone interval functions is given in Figure 4.3; although the other interval functions are more complicated, the basic idea is the same.

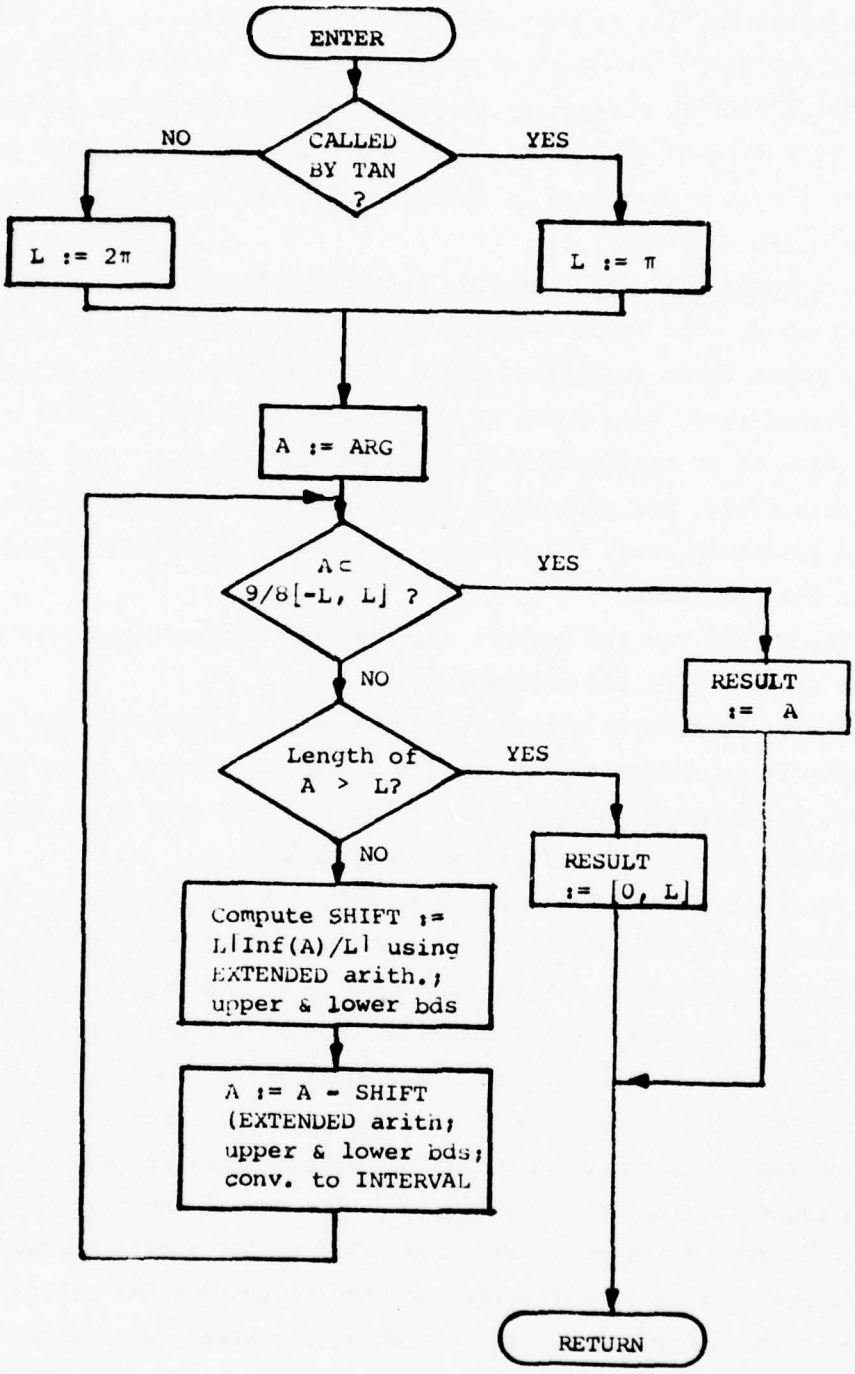
The trigonometric functions present another problem: since these functions are periodic, the normal method of evaluation is to reduce the argument to the principal range before computing the result. This can lead to considerable error, especially if the arguments are large; it is not practical to derive accuracy information over the entire domain of the functions. Consequently, it seems reasonable to reduce the interval argument to a limited range prior to evaluating the endpoints. In these routines, we first test the argument to see whether the interval includes an entire period of the function; if it does, the value of the result is obvious. If it does not, the argument is reduced using EXTENDED arithmetic, and the function is evaluated on the reduced interval. A flow chart of the reduction algorithm is given in Figure 4.4; this algorithm has been rigorously analyzed to insure that 1) it converges, and 2) the value of the function on the reduced interval contains the value of the function on the original interval.

Exponentiation: The routine which raises an interval to an integer power is designed to take dependency into account; no even power of any interval can contain a negative number. The routine which raises an interval to an interval power uses the formula $a^b = e^{b \ln a}$; and the routines which raise an interval to a BPA or EXTENDED power first convert the exponent to INTERVAL and then use the routine which raises an interval to an interval power.

Conversions: Those routines which convert other data types to INTERVAL are reasonably straightforward; the argument is rounded down (if necessary) to obtain the left endpoint of the result interval, and up to obtain the right. Conversion from INTERVAL to EXTENDED, BPA, REAL, or INTEGER is available, although the user should be aware that information is lost. These conversions consist of taking the number of the result type nearest the midpoint of the argument; note that in the case of conversion to INTEGER, the result may not even lie in the argument interval. The actions of the conversions between INTERVAL and Hollerith are documented in Section 2; these conversions employ the BPA routines.

FIGURE 4.4

FLOWCHART FOR INTRED



Unpack routine: This routine unpacks a packed Hollerith string and stores it, one character per word, in the array specified. A complete description of this routine is provided in Section 5, since it is a primitive which must be rewritten for each new implementation. In the UNIVAC 1110 version, the UNPACK routine recognizes the end-of-Hollerith-string sentinel, which consists of one minus zero word. The unpack routine tests each word for zero, and stops when such a word is encountered if it has not previously found an end-of-field sentinel.

I/O routines: The functions of these routines are adequately described in Section 2. The input routines check the input buffer to see whether another record needs to be read, and, if so, initiate the reading action. (For the free-format read, this check is made prior to retrieving each character, allowing fields to be continued across record boundaries.) They then isolate the next data field, and pass it to INTCHX for conversion to INTERVAL. In the case of the formatted read, the process is repeated until the specified number of fields has been read.

The INTASG routine unpacks the specified packed Hollerith string and passes it to INTCHX for conversion.

The write routine clears the write buffer, retrieves the specified data, converts it to Hollerith, and stores it in the buffer. When the buffer is filled, or when the specified number of INTERVAL data items have been converted, the routine causes the buffer to be written.

The isolation of data fields in the free format read routine is controlled by a decision table (as is the separation of endpoints in INTCHX and the conversion from Hollerith to BPA in BPACHB). We describe the procedure for INTRDF here; the algorithms for the other two routines are entirely analogous.

The decision table consists of two parts: a State table and a Response table. Initially, the STATE is defined to be 1. When a new character is encountered, it is used to derive an integer which determines the column of the State table (STATBL) to be consulted. The row of the State table is determined by the current value of STATE. The value in the specified row and column of the Response table is retrieved and used to control the processing, while the value in the specified row and column of the State table becomes the new value of STATE. Table 4.1 displays the State and Response tables for INTRDF.

TABLE 4.1

STATE AND RESPONSE TABLES FOR INTRDF

STATE TABLE

CHARACTER ENCOUNTERED	(	:	)	blank	,	# = $	other	MEANING IF EXIT OCCURS IN THIS STATE
STATE								
1	2	4	4	1	1	1	4	null field
2	2	2	3	2	2	2	2	incomplete
3	3	3	3	3	3	3	3	complete
4	4	4	4	4	4	4	4	complete

RESPONSE TABLE

1	1	1	1	4	3	3	1
2	1	1	1	4	1	3	1
3	2	2	2	2	3	3	2
4	2	1	1	3	3	3	1

RESPONSE	MEANING
1	Copy character into conversion buffer and increment pointers for input buffer and conversion buffer.
2	Decrement pointer for conversion buffer, set the number of characters to be converted equal to the new value of the pointer, and call INTCHX to convert string (Exit).
3	Increment pointer for input buffer, then follow the action specified in Response 2. This response is used to bypass the field separator characters on input.
4	Increment pointer for input buffer but do not copy character. This response is used to skip blanks.

Describing the package to AUGMENT: Since the INTERVAL package is designed to be preprocessed by AUGMENT, the components of the package must be described to AUGMENT via a description deck. This deck informs AUGMENT about the data representations used; the modules available in the package, their calling sequences, the data types of the arguments and results, the structures of the modules; and various other things AUGMENT needs to know in order to translate the source programs. Instructions for preparing the description deck may be found in [3]; the description decks used for the UNIVAC 1110 version of the INTERVAL package are listed in Appendix 5.

Since the use of AUGMENT is central to the processing of the package, it is advisable to have complete information on AUGMENT. In addition to the user documentation [3], the reader is advised to consult the technical documentation [4].

5. Adapting the package to other hardware:

Adapting the INTERVAL package to other hardware should not be extremely difficult provided that the AUGMENT precompiler is available. The necessary steps are

1) Determine the representations for BPA and EXTENDED numbers;
2) Code or revise primitives as necessary;
3) Process the package using AUGMENT and the FORTRAN compiler;
4) Check the revised package;
5) Tune and recheck the package (if tuning is desired).

This section provides detailed instructions for accomplishing these tasks.

The use of the AUGMENT precompiler preserves both naturality of expression and flexibility. As we have seen, most of the INTERVAL package is written in terms of the nonstandard data types INTERVAL, EXTENDED, and BPA. AUGMENT operates on the package just as it would on an applications program, converting operations on variables of these nonstandard data types to calls on other routines in the package. By this technique, we confine the binding to specific data representations to a handful of primitive routines, together with constants stored in COMMON storage blocks.

Every effort has been made to write the nonprimitive parts of INTERVAL so that the FORTRAN program produced by AUGMENT will conform to ANSI standards. Because of this, each logical module is a separate subprogram; any existing module that does not meet the needs of a different system may simply be replaced with one that does.

The package has been designed so that the representation of BPA numbers may occupy two or more words if that is desired. In order to support this, there are subprograms in the package which may be extraneous if BPA and REAL formats are the same; this will lead to inefficiency in the object code unless changes are made. Consequently, the package may well benefit from some amount of tuning to the host system once AUGMENT processing has been completed.

Data representations: The first step in adaptation of the package is to decide on data representations for the types BPA and EXTENDED. As noted above, these data types have been left undefined throughout most of the package in order to allow the user flexibility in setting the precision of the data; however, there are several implicit assumptions which will influence the choice of

these representations to some extent:

1. All operations and functions on BPA numbers will be provided explicitly in the BPA portion of the package.

2. Since EXTENDED is used in evaluating BPA mathematical functions, and since it is desirable to achieve the greatest possible amount of accuracy in the BPA function evaluations, it is advisable to bind EXTENDED to a higher precision data representation than that of BPA.

3. A complete supporting package for EXTENDED (including all mathematical functions) is available.

4. Every BPA number has an exact representation in EXTENDED format.

5. Bounds are available for the accuracy of EXTENDED library routines. (The inaccuracies in EXTENDED function evaluations are treated by adding a "fudge factor" to [or subtracting it from] the result of an EXTENDED function evaluation.)

If single precision interval arithmetic is adequate for the purposes for which the INTERVAL package will be used, adaptation of the package will be simplified. Some existing primitives make use of the following additional assumptions, which, of course, normally include Assumptions 2, 3, and 4 above:

5. REAL and BPA formats are identical;

6. DOUBLE PRECISION and EXTENDED formats are identical;

7. Every FORTRAN integer has an exact representation in EXTENDED format.

Throughout the package, we assume that INTERVAL numbers are BPA arrays of length 2.

Representation dependent primitives: These are the parts of the package that may need to be revised or rewritten for a new host computer system. The nineteen primitives are listed in Table 5.1; for convenience, we have divided them into five types:

1. FORTRAN BLOCK DATA modules containing constants which depend on the word length of the machine and the particular representations of BPA, EXTENDED, and INTERVAL numbers.

2. FORTRAN primitives which should need no modification if REAL format is the same as BPA and DOUBLE PRECISION format is the same as EXTENDED.

TABLE 5.1

NAME	TYPE	FUNCTION
BPACOMMON	1	BLOCK DATA module for BPA constants
BPAADD	5	Form sum of two BPA numbers
BPACBD	3	Convert BPA number to internal array representation
BPACBE	2	Convert BPA number to EXTENDED format
BPACBI	2	Convert BPA number to FORTRAN INTEGER format
BPACBR	2	Convert BPA number to FORTRAN REAL format
BPACEB	5	Convert EXTENDED number to BPA format
BPACIB	2	Convert FORTRAN INTEGER to BPA format
BPACRB	2	Convert FORTRAN REAL number to BPA format
BPACSB	3	Convert internal array representation to BPA format
BPADIV	5	Form quotient of two BPA numbers
BPAMUL	5	Form product of two BPA numbers
BPANEG	2	Unary minus operator
BPASGN	2	Signum function for BPA numbers
BPASTR	2	Replacement operator for BPA numbers
BPASUB	5	Form difference of two BPA numbers
INTCOMMON	1	BLOCK DATA module for INTERVAL and EXTENDED constants
INTUPK	3	Unpack packed Hollerith string
INTRAP	4	Detect errors and print messages

REPRESENTATION DEPENDENT PRIMITIVES

3. FORTRAN primitives which use the field (FLD) function; they also depend on word size and BPA format (BPACBD and BPACSB) or the internal representation of Hollerith strings (INTUPK)

4. This primitive (INTRAP) contains FORMAT statements which may be system or representation dependent.

5. Assembly language primitives; these must be rewritten for any host computer which is not compatible with the UNIVAC 1110.

In addition to these primitives, INTRD and INTRDF contain nonstandard READ statements which recognize an END OF FILE condition. If the host compiler does not allow this form of READ statement, those statements should be replaced by the statements given in the adjacent COMMENT cards.

We now examine in detail the modifications that may be required. If any primitive must be rewritten, it should be written as a SAFE SUBROUTINE in the

sense of AUGMENT; that is, it should function properly even in the case that its operands and results overlap in storage.

Type 1 primitives: There are two BLOCK DATA modules, called BPACOMMON and INTCOMMON, which contain the various COMMON blocks used in the BPA and INTERVAL portions of the pacakge, respectively. The COMMON blocks are described in Appendix 4; we list here the changes that might be required in the adaptation procedure.

In some adaptations, it may be necessary to change the lengths of certain arrays. Appendix 4 also contains a list of the package modules and the COMMON blocks used in each, so that all uses of any specific COMMON block can be located easily for revision.

The following is a list of those variables which need to be verified or altered for a different host computer system or data representation:

BPACOMMON

BPACOM - No change

BPACON - All variables listed in Appendix 4 depend on the BPA data format; appropriate values for these constants in the selected format must be supplied.

BPAACC - All constants listed depend on the EXTENDED library routines. The appropriate values must be stored in the IACC array; values for the other variables must be determined and supplied. In addition, if library functions other than the ones supplied with the package are to be implemented, corresponding cells in the IACC array must be assigned and the appropriate accuracy information must be inserted. Note that IACC gives the number of internal digits of the result of the corresponding function which are guaranteed to be accurate.

BPACCM - EXWDTH, ESDMAX, EXMAX, and EXMIN must be determined. The dimensions of the arrays IX, ICX, ECX, and EX will need to be increased if larger numbers are allowed; for each of these arrays, the dimension must be 10 greater than the maximum number of digits in the given base that will be stored in the array. For the fixed point arrays,

the dimension must be adequate to handle the largest integer and the smallest fraction that can be expressed in BPA format, with enough digits left over to insure proper rounding. The dimension of each of these arrays must be stored in the first cell of the array. The second cell of each array must be set equal to the number base of that array. For EX, this will normally be 10; for IX, it will be the base of the BPA number representation. The base of ECX must be a power of the external base, and the base of ICX must be a power of the internal base. Finally, the base of ECX times the base of ICX must be a FORTRAN integer. The appropriate powers of the base of IX and EX must be stored in ICX(3) and ECX(3), respectively; IX(3) and EX(3) must be 1. The location of the radix point in the fixed point arrays ICX and ECX must be stored in ICX(4) and ECX(4), respectively; this must be determined so that the largest integer to be handled in the array (i. e., the largest BPA number or the largest external number whose exponent does not exceed EXMAX, whichever is greater) will have no more than ICX(4)-10 or ECX(4)-10 integer digits, respectively. Finally, IX(9) must be set to the number of significant digits in BPA format, and ICX(9) must be determined so that the given number of digits in the base of ICX will yield at least IX(9) digits in the base of IX.

INTCOMMON

INTCOM - No change, unless the number of cells occupied by INTEGER, REAL, or DOUBLE PRECISION data exceeds the number of cells occupied by the longer of INTERVAL or EXTENDED. In this case, the longest data type should be declared and variables of that type EQUIVALENCEd to TA, TB, and TR.

INTCON - All constants listed must be determined for the data representations being used and stored in this COMMON block. Note that, in the UNIVAC 1110 version, REAL

arrays are EQUIVALENCEd to the EXTENDED constants; the names of the REAL arrays are the same as the names of the EXTENDED constants except that they are prefixed with 'R'. This was done because the 1110 compiler will not allow octal data to be supplied for DOUBLE PRECISION quantities.

INTCCM - The table of I/O units recognized may be expanded if desired; if more than 8 units are to be recognized, the dimension of the table will need to be changed. See the user instructions in Section 2 for the method of adding new units. Standard unit numbers, record lengths, and numbers of characters to be scanned may all be changed if desired; however, note that if records longer than 132 characters are to be processed, the dimensions of the buffers and the related constants must also be changed. The minimum field width for interval output may need to be changed (it will, if 3-digit exponents are allowed). Other changes are discretionary. WARNING: If the buffer lengths are altered, do not neglect to change the dimension of the array HSTR.

INTCRD - Changes are discretionary.

INTRCM - No changes are needed unless additional special functions are implemented. See Section 2 for the method of adding new special functions.

Type 2 primitives: These routines are principally data moving and conversion routines, and should need no modification if Assumptions 5, 6, and 7 are valid. Otherwise, rewriting may be necessary, and the potential for fault conditions may be introduced in the rewriting. We list the Type 2 primitives, indicate the function of each, and note whether errors are possible *in the current version of the package.* We do not restate the assumptions for each primitive, and we do not indicate what errors might be introduced for different data representations, since the possibilities are virtually endless. It is the responsibility of the person modifying the package to ascertain what errors, if any, are possible in the new primitives and to link the primitives

with the remainder of the package whenever errors are possible in the new version where none were in the basic package.

BPACBE - converts the BPA argument to an EXTENDED result. No errors are possible. Currently implemented as a DOUBLE PRECISION function.

BPACBI - converts the BPA argument to an INTEGER result. The result is the nearest integer to the argument in the direction specified by the rounding option OPTION, located in BPACOM. Overflow and infinity faults are possible. Currently implemented as an INTEGER function.

BPACBR - converts the BPA argument to a REAL result. No errors are possible. Currently implemented as a REAL function.

BPACIB - converts the first argument (an INTEGER) to BPA and stores the resulting value in the second argument. The BPA result is the nearest BPA number to the given INTEGER in the direction specified by OPTION. Sufficiently small integers are converted exactly. No errors are possible. Currently implemented as a SUBROUTINE.

BPACRB - converts the first argument (a REAL number) to BPA and stores the resulting value in the second argument. No errors are possible, although all package routines using this routine check for errors because errors could occur if the computer operates in two's complement mode. The BPA numbers must be closed under negation; therefore, in a two's complement format, real numbers which have no additive inverse must not be converted to BPA -- these cases must result in an error indication. Currently implemented as a SUBROUTINE.

BPANEG - The negative of the value of the first argument is stored in the second argument; both arguments are BPA. No errors can occur. Currently implemented as a SUBROUTINE.

BPASGN - Signum function for BPA numbers. The result is set equal to -1, 0, or +1 according as the argument is less than, equal to, or greater than zero. No errors are possible. This primitive uses an arithmetic IF statement. Currently implemented as an INTEGER function.

BPASTR - The value of the first argument is stored in the second argument; both arguments are BPA. No errors are possible. Currently implemented as a SUBROUTINE.

Type 3 primitives: These primitives are used in number base conversions. Although they are written in FORTRAN, they use nonstandard features of the UNIVAC 1110 FORTRAN compiler, and they are dependent upon data representations. No attempt was made to render these primitives portable in any situation other than transfer of the package to a like computer system.

BPACBD - The normalized floating point BPA number specified by the first argument is converted to array format and stored in the array specified by the second argument. If errors occur, the third argument is set to an appropriate integer value; otherwise it must be set to zero. This subroutine simply unpacks the fraction of the BPA number and stores it, one digit per word, in the designated array, beginning with the cell specified as the location of the radix point in the array (actually the location of the first fraction digit). This index is stored in the fourth cell of the destination array itself. The true (unbiased) exponent of the BPA number must be stored in the seventh location of the destination array as a signed integer quantity; the eighth cell of the array is set to +1 if the given number is positive or zero, and -1 if the given number is negative. In addition, the routine must store the index of the most significant digit of the fraction in the fifth cell of the array, and the index of the least significant digit in the sixth cell. In the UNIVAC 1110 implementation, no errors can occur; none should be possible in any adaptation, unless improper floating point numbers are a possibility.

BPACSB - This routine is the inverse of BPACBD. The first argument specifies an array which contains an unpacked number; this routine packs it, checks for exponent range faults, and stores the resulting value in the second argument. Overflow, infinity, and underflow faults are possible, and must be reported in the BPAFLT cell of BPACOM, as indicated in Appendix 4. The routine expects to find the sign of the number (+1 or -1) in the eighth cell of the array; the unbiased exponent in the seventh cell; the location of the most significant fraction digit in the fifth cell; and the location of the least significant fraction digit in the sixth cell. See the flow chart for this routine given in Figure 4.2.

INTUPK - The packed Hollerith string which begins at the location specified by the first argument is unpacked and stored, one character per word, beginning at the location specified by the second argument. The maximum number of characters to be unpacked is specified by the third argument, and the fourth argument is set equal to the number actually unpacked. The unpacked characters must be compatible with characters read in A1 format. This routine stops unpacking when a field delimiter character (as defined by the ICHR array in BPACOM) is encountered (such character is not stored in the array of unpacked characters or included in the character count), or when the maximum number of characters is encountered. If the FORTRAN compiler provides a sentinel for packed hollerith strings, INTUPK may be written to recognize that sentinel; in this case, the user of INTASG need not provide a field delimiter at the end of the Hollerith string.

Type 4 primitive:

INTRAP - This primitive depends on data formats and, in some cases, uses FORTRAN I/O conversion routines. The output formats should be examined to ascertain that the decimal and Hollerith formats are adequate to contain the numbers which will be represented and that the formats which display the internal form of the information are consistent with that form. If EXTENDED is represented in an internal format other than that of DOUBLE PRECISION, code may have to be added to INTRAP to perform the necessary conversions.

Type 5 primitives: These are the primitives required for BPA arithmetic and conversion from EXTENDED to BPA. Each of these primitives should be capable of upward and downward directed roundings, and of indicating underflow, overflow, and infinity faults. The rounding option is communicated to each of these routines via the OPTION cell in BPACOM, and the appropriate fault indicator is stored in the BPAFLT cell of the same COMMON block by each of these routines. Appendix 4 lists the possible values of these variables and their meanings; a short discussion of directed roundings and faults may be found in Section 3. More detailed discussions, together with algorithms for the arithmetic operations, may be found in [17], and the person who is going to write the arithmetic primitives for a new adaptation is urged to consult that paper.

BPAADD - Forms the sum of the first two arguments and stores the resulting value in the third argument. Possible errors are underflow, overflow, and infinity.

BPADIV - Divides the first argument by the second argument and stores the quotient in the third argument. Faults are the same as for BPAADD, plus division by zero.

BPAMUL - Forms the product of the first two arguments and stores the resulting value in the third argument. Possible errors are the same as for BPAADD.

BPASUB - Subtracts the second argument from the first and stores the difference in the third argument. Faults are the same as for BPAADD.

BPACEB - Converts the first argument (an EXTENDED number) to BPA under the assumption that the first ACC digits are accurate and that the result is to be rounded according to OPTION (ACC and OPTION are located in BPACOM). The resulting value is stored in the second argument. If ACC is zero, all digits of the EXTENDED number are assumed to be accurate, and only rounding is done; otherwise, 1 is added or subtracted at the ACCth digit (depending on the direction of the rounding) before rounding takes place. This routine assumes that an EXTENDED zero is accurate regardless of the value of ACC; thus, if a bound is desired for a number whose computed value is zero, that bound will have to be obtained by another device.

AUGMENT processing: The INTERVAL package is written under the assumption that the AUGMENT precompiler will be available. The source code for the package is processed in exactly the same manner as any other program using interval arithmetic; see Section 2 for details. Description decks, which inform AUGMENT of the structure of INTERVAL, EXTENDED, and BPA data and give it information about the INTERVAL package, are provided with the package; however, they must be carefully checked to make sure they correspond with the revised package before AUGMENT processing is undertaken. Any JCL control cards needed are, of course, the responsibility of the user.

If AUGMENT is not available, it will be necessary to seek assistance from another installation where AUGMENT is available. The hardware need not be compatible, since the output of AUGMENT is a FORTRAN source program. Once the

FORTRAN version of the package has been prepared, further modifications may be made directly, without need of reprocessing by AUGMENT.

Checking the package: A test program is supplied with the INTERVAL package, and sample output is included on the microfiche that accompanies this report. Insofar as possible, this test program utilizes data from the package itself or from run-time conversion of Hollerith strings, so the changes required for a new implementation should be minimal. However, if a broader exponent range is allowed in the modified version of the package, certain of the input data which cause exponent range faults on conversion in the 1110 version may give acceptable results in the revised version.

Tuning the package: Due to the flexibility of the representation of BPA numbers, superfluous calls on subroutines may be generated. The tuning operation consists primarily of removing unnecessary subroutine calls and using more efficient means of accomplishing the same result.

If BPA variables are declared as undimensioned variables of a standard FORTRAN data type, say REAL or DOUBLE PRECISION, and if the standard format is compatible with BPA format, the BPASTR and BPANEG routines may be unnecessary. AUGMENT can be instructed to use the standard replacement and unary minus operators instead of generating calls on BPASTR and BPANEG; this has been done in the UNIVAC 1110 version. This step in itself trims unnecessary overhead significnatly.

In cases where conversion between a standard data type and BPA amounts to nothing more than a replacement operation, the calls on the BPA conversion routines can be replaced by the appropriate in-line FORTRAN statements.

If INTERVAL is declared to be an array of any standard data type, calls on the INF and SUP routines can be replaced by direct references to the first and second elements of the array, respectively. This modification will have to be made for each reference to the INF and SUP functions after AUGMENT processing has been completed, however, since AUGMENT does not have the capability of doing this.

If BPA format is the same as REAL, another possibility is open unless the compiler is overzealous about type checking: INTERVAL can be declared COMPLEX. If this is done, the REAL and AIMAG functions of standard FORTRAN can be used to extract the INF and SUP, respectively; however, this will have no effect on

insertion of the INF and SUP, so the comments of the previous paragraph apply to the case where INF and SUP appear on the left of the = sign.

Other tuning procedures may be possible, depending on the data representations used. The only restriction is that the rigor of the package must not be compromised. For example, the routines which evaluate the BPA relational operators call on the BPA subtract routine; this should not be altered unless the hardware subtract instruction always produces a result of the same sign as the true result, even in cases of underflow and overflow. If the hardware sets an underflow to zero, or gives garbage when overflow occurs, the hardware subtract must not be used, since its use could result in incorrect evaluation of these operators.

Needless to say, the package must be rechecked whenever any changes are made.

6. Conclusion:

We have attempted to provide both user documentation and technical documentation for the INTERVAL Arithmetic Package in this report. We have tried to strike a balance between brevity and completeness; it is likely that we have not always succeeded. Any errors or inadequacies in either this document or the package itself should be reported to the author.

The author is pleased to acknowledge the contributions of many people to this effort: Tom Ladner collaborated with the author in the development of an earlier version of this package, and the current package is based in part on that effort. F. D. Crary and S. T. Jones contributed materially to the design of the package and made numerous constructive suggestions regarding this document. The Waterways Experiment Station, U. S. Army Corps of Engineers, provided the impetus for the development of this package through their interest in the application of Interval Analysis to engineering problems; they also provided funds for the partial support of the development and testing of the package. Our primary contacts there were William Boyt and James B. Cheek. The Waterways Experiment Station also provided grants for the implementation of this package on the MULTICS (Honeywell) system at the University of Southwest Louisiana under the direction of Prof. Bruce D. Shriver; on the IBM System/370 and Digital Equipment Corp. DEC-10 at the University of Texas at Arlington, Texas under the direction of Dr. Ronnie G. Ward; on the Honeywell 635 at the University of Kansas under the direction of Dr. Dick Hetherington; and on the CDC Cyber series machine at Southern Methodist University under the direction of Prof. Myron Ginsberg.

DISCLAIMER

This program has been tested and is believed to be correct. However, in any program of this size, residual errors are likely. Neither the author, nor the Mathematics Research Center, nor the University of Wisconsin, nor the United States Army, nor any other party, conceivable or inconceivable, will assume any responsibility for damages, whether direct, indirect, consequential, or inconsequential, arising from errors, omissions, malfunctions, irregularities, insufficiencies, or any other sins of omission or commission in this program and/or its documentation.

REFERENCES

1. *ANSI Standard FORTRAN*, American National Standards Institute, New York, 1966.
2. *FORTRAN V Library Functions Reference Manual for the 1108*, The University of Wisconsin Computing Center, Madison, Wisconsin, October, 1969.
3. F. D. Crary, *The AUGMENT precompiler I: user information*, The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report # 1469, December, 1974; revised April, 1976.
4. _____, *The AUGMENT precompiler II: technical documentation*, The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report # 1470, October, 1975.
5. Donald I Good and Ralph L. London, *Computer interval arithmetic: definition and proof of correct implementation*, J. Assoc. Comput. Mach. 17 (1970), 603 - 612.
6. Jan Kent, *Theoretical definition, analysis, and comparison of floating point instructions*, Norsk Regnesentral Publication # 425, Oslo, 1973.
7. Donald E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison - Wesley Publishing Company, Reading, Mass., 1969.
8. U. Kulisch, *An axiomatic approach to rounded computations*, Numer. Math. 18 (1971), 1 - 17.
9. Allan L. Lang and Bruce D. Shriver, *The design of a polymorphic arithmetic unit*, Third IEEE-TCCA Symposium on Computer Arithmetic, November, 1975, 48 - 55.
10. David W. Matula, *In-and-out conversions*, Comm. ACM 11 (1968), 47 - 50.
11. _____, *Foundations of finite precision arithmetic*, Proceedings of the IEEE-TCCA Symposuim on Computer Arithmetic, May, 1972.
12. Ramon E. Moore, *Interval Analysis*, Prentice - Hall, Inc., Englewood Cliffs, N. J., 1966.
13. Frederic N. Ris, *Tools for the analysis of interval arithmetic*, Lecture Notes in Computer Science 29: Interval Mathematics, Springer - Verlag, New York, 1975.
14. _____, *A unified decimal floating-point architecture for the support of high-level languages (extended abstract)*, SIGNUM Newsletter 11, 3 (October, 1976), 18 - 32.

15. J. M. Yohe, *Accurate conversion between number bases,* The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report # 1109, October, 1970.

16. _____, *Foundations of floating point computer arithmetic,* The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report # 1302, January, 1973.

17. _____, *Roundings in floating-point arithmetic,* IEEE Trans. Computers C-22 (1973), 577 - 586.

18. _____, *Implementing a nonstandard data type,* The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report # 1545, April, 1975.

APPENDIX 1

STANDARD FORTRAN NUMBER AND INTERVAL NUMBER REPRESENTATIONS

```
DIGIT        ::= 0|1|2|3|4|5|6|7|8|9

SIGN         ::= +|-

INTEGER      ::= NULL|<SIGN>|<INTEGER><DIGIT>

RADIX        ::= .

FIXEDPOINT   ::= <INTEGER><RADIX>|<FIXEDPOINT><DIGIT>

EXPSEP       ::= E|D

EXPONENT     ::= <SIGN>|<EXPSEP>|<EXPSEP><SIGN>|<EXPONENT><DIGIT>

NUMBER       ::= <INTEGER>|<FIXEDPOINT>|<INTEGER><EXPONENT>|
                 <FIXEDPOINT><EXPONENT>

ENDPTSEP     ::= :

COMMA        ::= ,

INTERVAL     ::= <NUMBER>|(<NUMBER>)|<NUMBER><ENDPTSEP><NUMBER>|
                 (<NUMBER><ENDPTSEP><NUMBER>)|(<NUMBER><COMMA><NUMBER>)
```

APPENDIX 2

PACKAGE MODULES

This appendix contains condensed information on the modules of the INTERVAL package. The first two pages list the modules of the BPA part of the package; the next three pages list the modules of the INTERVAL part of the package. The first column of the table lists the function of the module; the second provides an expanded explanation of the function or, in some cases, the definition of the function. The third column lists the result type of the module. The fourth column gives an example of how the module might be invoked via AUGMENT; the reference name of the module is indicated in italics and variables are indicated in standard type. The fifth column indicates how the module would be invoked in the absence of AUGMENT; if the routine is a subroutine (indicated by an S in column 6), the expression given must be preceded by the word CALL; otherwise, the expression given is complete. The sixth column indicates the type of the routine, according to the description given below.

Data types: Data types are indicated by one or two letter abbreviations. The abbreviations used are as follows:

B - BPA
D - DOUBLE PRECISION
E - EXTENDED
H - HOLLERITH
I - INTEGER
IA - INTEGER ARRAY
L - LOGICAL
PH - PACKED HOLLERITH
R - REAL
UH - UNPACKED HOLLERITH
X - INTERVAL

Routine types: S indicates subroutine; any other letter indicates a function of the indicated type (see above). If the routine type is suffixed with an A, the 1110 version was written in assembly language; otherwise it was written in extended FORTRAN. If the routine type is suffixed with an asterisk, the routine is a primitive.

Variable names: The first letter (or, sometimes, two letters) indicates the type of the variable. The last letter is R for result, A or B for argument. Other letters may be used for special meanings; for example, E by itself denotes an error indicator. Explanations not given here will be found in the text.

OPERATION	DEFINITION/EXPLANATION	RESULT TYPE	ROUTINE INVOCATION VIA AUGMENT	ROUTINE INVOCATION DIRECT	ROUTINE TYPE
ARITHMETIC					
Add	Sum of two BPA numbers	B	BA + BB†	BPAADD(BA, BB, BR)†	SA*
Subtract	Difference of two BPA numbers	B	BA - BB†	BPASUB(BA, BB, BR)†	SA*
Multiply	Product of two BPA numbers	B	BA * BB†	BPAMUL(BA, BB, BR)†	SA*
Divide	Quotient of two BPA numbers	B	BA / BB†	BPADIV(BA, BB, BR)†	SA*
EXPONENTIATION					
to INTEGER	Raise BPA number to INTEGER power	B	BA ** IB†	BPAXBI(BA, IB, BR)†	S
MATHEMATICAL					
Absolute value	Absolute value of BPA number	B	*ABS*(BA)	BPAABS(BA, BR)	S
Arc cosine	Arc cosine of BPA number	B	*ACOS*(BA)†	BPAACS(BA, BR)†	S
Arc sine	Arc sine of BPA number	B	*ASIN*(BA)†	BPAASN(BA, BR)†	S
Arc tangent	Arc tangent of BPA number	B	*ATAN*(BA)†	BPAATN(BA, BR)†	S
Cube root	Cube root of BPA number	B	*CBRT*(BA)†	BPACBT(BA, BR)†	S
Cosine	Cosine of BPA number	B	*COS*(BA)†	BPACOS(BA, BR)†	S
Hyperbolic cosine	Hyperbolic cosine of BPA number	B	*COSH*(BA)†	BPACSH(BA, BR)†	S
Exponential	e ↑ BA	B	*EXP*(BA)†	BPAEXP(BA, BR)†	S
Integer	Nearest integer in specified direction	B	*INT*(BA)†	BPAINT(BA, BR)†	S
Natural logarithm	Log to the base e of BPA number	B	*LN*(BA)† or *LOG*(BA)†	BPALN (BA, BR)†	S
Common logarithm	Log to the base 10 of BPA number	B	*LOG10*(BA)†	BPALOG(BA, BR)†	S
Maximum	Maximum of two BPA numbers	B	*MAX*(BA, BB)	BPAMAX(BA, BB, BR)	S
Minimum	Minimum of two BPA numbers	B	*MIN*(BA, BB)	BPAMIN(BA, BB, BR)	S
Sign	Sign of BPA number: +1 if BA > 0; -1 if BA < 0; and 0 if BA = 0	I	*SGN*(BA)	BPASGN(BA)	I*
Sine	Sine of BPA number	B	*SIN*(BA)†	BPASIN(BA, BR)†	S
Hyperbolic sine	Hyperbolic sine of BPA number	B	*SINH*(BA)†	BPASNH(BA, BR)†	S
Square root	Square root of BPA number	B	*SQRT*(BA)†	BPASQT(BA, BR)†	S
Tangent	Tangent of BPA number	B	*TAN*(BA)†	BPATAN(BA, BR)†	S
Hyperbolic tangent	Hyperbolic tangent of BPA number	B	*TANH*(BA)†	BPATNH(BA, BR)†	S
CONVERSION					
B → E	BPA to EXTENDED conversion	E	*CTE*(BA)	BPACBE(BA)	D*
B → UH	BPA to unpacked Hollerith (width IB)	UH	-	BPACBH(BA, HR, IB)†	S
B → I	BPA to INTEGER	I	*CTI*(BA)†	BPACBI(BA)†	I*
B → R	BPA to REAL	R	*CTR*(BA)	BPACBR(BA)	R*
E → B	EXTENDED to BPA	B	*CTB*(EA)†§	BPACEB(EA, BR)†§	SA*
UH → B	Unpacked Hollerith to BPA	B	-	BPACHB(HA, IB, BR)†	S
I → B	INTEGER to BPA	B	*CTB*(IA)†	BPACIB(IA, BR)†	S*
R → B	REAL to BPA	B	*CTB*(RA)	BPACRB(RA, BR)	S*

† Rounding option desired must be stored in OPTION, located in BPACOM, prior to calling this routine.
§ Relative accuracy of EXTENDED number must be stored in ACC, located in BPACOM, prior to calling this routine.

OPERATION	DEFINITION/EXPLANATION	RESULT TYPE	ROUTINE INVOCATION VIA AUGMENT	DIRECT	ROUTINE TYPE
SERVICE					
Store	Replacement operator	B	BR = BA	BPASTR(BA, BR)	S*
Negate	Unary minus	B	-BA	BPANEG(BA, BR)	S*
RELATIONAL					
Equal	BA = BB	L	BA .EQ. BB	BPAEQ(BA, BB)	L
Greater or equal	BA $\geq$ BB	L	BA .GE. BB	BPAGE(BA, BB)	L
Greater than	BA > BB	L	BA .GT. BB	BPAGT(BA, BB)	L
Less or equal	BA $\leq$ BB	L	BA .LE. BB	BPALE(BA, BB)	L
Less than	BA < BB	L	BA .LT. BB	BPALT(BA, BB)	L
Not equal	BA $\neq$ BB	L	BA .NE. BB	BPANE(BA, BB)	L
HOLLERITH $\leftrightarrow$ BPA CONVERSION SUBMODULES					
Unpack BPA	Unpack BPA number into IAR array (E=error)	IA	-	BPACBD(BA, IAR, E)	S*
H → I	Convert Hollerith character to INTEGER	I	-	BPACHI(HA)	I
I → H	Convert INTEGER to Hollerith character	H	-	BPACIH(IA)	H
Float	Convert fixed point array number to floating point array number and round	IA	-	BPACPD(IAA, IAR, E)†	S
Pack BPA	Pack IAA array into BPA format	B	-	BPACSB(IAA, BR, E)†	S*
Convert	Number base conversion from IAA to IAR (fixed point)	IA	-	BPACSD(IAA, IAR, E)	S
Fix	Convert floating point array number to fixed point array number	IA	-	BPACSP(IAA, IAR, E)	S

† Rounding option desired must be stored in OPTION, located in BPACOM, prior to calling this routine.

OPERATION	DEFINITION/EXPLANATION	RESULT TYPE	ROUTINE INVOCATION VIA AUGMENT	DIRECT	ROUTINE TYPE
ARITHMETIC					
Add	Sum of two intervals	X	XA + XB	INTADD(XA, XB, XR)	S
Subtract	Difference of two intervals	X	XA - XB	INTSUB(XA, XB, XR)	S
Multiply	Product of two intervals	X	XA * XB	INTMUL(XA, XB, XR)	S
Divide	Quotient of two intervals	X	XA / XB	INTDIV(XA, XB, XR)	S
EXPONENTIATION					
to BPA	Raise interval to BPA power	X	XA ** BB	INTXXB(XA, BB, XR)	S
to EXTENDED	Raise interval to EXTENDED power	X	XA ** EB	INTXXE(XA, EB, XR)	S
to INTEGER	Raise interval to INTEGER power	X	XA ** IB	INTXXI(XA, IB, XR)	S
to INTERVAL	Raise interval to INTERVAL power	X	XA ** XB	INTXXX(XA, XB, XR)	S
MATHEMATICAL					
Absolute value	$\{\|x\| : x \in XA\}$	X	*ABS*(XA)	INTABS(XA, XR)	S
Arc cosine	Arc cosine of interval XA	X	*ACOS*(XA)	INTACS(XA, XR)	S
Arc sine	Arc sine of interval XA	X	*ASIN*(XA)	INTASN(XA, XR)	S
Arc tan (2 args)	Arc tangent of XA / XB	X	*ATAN2*(XA, XB)	INTAT2(XA, XB, XR)	S
Arc tangent	Arc tangent of interval XA	X	*ATAN*(XA)	INTATN(XA, XB)	S
Cube root	Cube root of interval XA	X	*CBRT*(XA)	INTCBT(XA, XR)	S
Cosine	Cosine of interval XA	X	*COS*(XA)	INTCOS(XA, XR)	S
Hyperbolic cosine	Hyperbolic cosine of interval XA	X	*COSH*(XA)	INTCSH(XA, XR)	S
Exponential	e ↑ XA	X	*EXP*(XA)	INTEXP(XA, XR)	S
Integer	Smallest interval with integer endpoints containing the interval XA	X	*INT*(XA)	INTINT(XA, XR)	S
Natural logarithm	Log to the base e of interval XA	X	*LN*(XA) or *LOG*(XA)	INTLN (XA, XR)	S
Common logarithm	Log to the base 10 of interval XA	X	*LOG10*(XA)	INTLOG(XA, XR)	S
Sine	Sine of interval XA.	X	*SIN*(XA)	INTSIN(XA, XR)	S
Hyperbolic sine	Hyperbolic sine of interval XA	X	*SINH*(XA)	INTSNH(XA, XR)	S
Square root	Square root of interval XA	X	*SQRT*(XA)	INTSQT(XA, XR)	S
Tangent	Tangent of interval XA	X	*TAN*(XA)	INTTAN(XA, XR)	S
Hyperbolic tangent	Hyperbolic tangent of interval XA	X	*TANH*(XA)	INTTNH(XA, XR)	S
FIELD					
Infimum	Left endpoint of interval XA	B	*INF*(XA)	INTINL(BA, XR) (insertion)	S
				INTINF(XA, BR) (extract'n)	S
Supremum	Right endpoint of interval XA	B	*SUP*(XA)	INTSPL(BA, XR) (insertion)	S
				INTSUP(XA, BR) (extract'n)	S
SPECIAL INTERVAL FUNCTIONS					
Compose	Form interval from two BPA endpoints	X	*COMPOS*(BA, BB)	INTCPS(BA, BB, XR)	S
Distance	Max($\|$Inf(XA)-Inf(XB)$\|$, $\|$Sup(XA)-Sup(XB)$\|$)	B	*DIST*(XA, XB)	INTDST(XA, XB, BR)	S
Half length	(Sup(XA)-Inf(XA))/2, rounded up	B	*HLGTH*(XA)	INTHLB(XA, BR)	S
Length	Sup(XA)-Inf(XA), rounded up	B	*LGTH*(XA)	INTLGB(XA, BR)	S

OPERATION	DEFINITION/EXPLANATION	RESULT TYPE	ROUTINE INVOCATION VIA AUGMENT	DIRECT	ROUTINE TYPE
SPECIAL INTERVAL FUNCTIONS (continued)					
Magnitude	Sup(Abs(XA))	B	*MAG*(XA)	INTMAG(XA, BR)	S
Midpoint	(Sup(XA)+Inf(XA))/2, rounded nearest	B	*MDPT*(XA)	INTMDB(XA, BR)	S
Mignitude	Inf(Abs(XA))	B	*MIG*(XA)	INTMIG(XA, BR)	S
Pivot	$\sqrt{\text{Mag(XA)}\times\text{Mig(XA)}}$, rounded down	B	*PIVL*(XA)	INTPVL(XA, BR)	S
	same, rounded as specified by IB	B	-	INTPVO(XA, IB, BR)	S
	same, rounded up	B	*PIVU*(XA)	INTPVU(XA, BR)	S
Intersection	Set-theoretic intersection of XA and XB	X	XA.*INTSCT*.XB	INTSCT(XA, XB, XR)	S
Sign	+1 if Inf(XA) > 0; -1 if Sup(XA) < 0; 0 if 0 ε XA	I	*SGN*(XA)	INTSGN(XA)	I
Size	(Abs(Inf(XA)) + Abs(Sup(XA)))/2	B	*SIZE*(XA)	INTSIZ(XA, BR)	S
Union	Smallest interval containing both XA, XB	X	XA.*UNION*.XB	INTUNN(XA, XB, XR)	S
CONVERSION					
PH → X	Packed Hollerith to Interval	X	*CTX*(string)	INTASG(HA, XR)	S
E → X	Extended to Interval, bounded at IBth dig.	X	-	INTBND(EA, IB, XR)	S
B → X	BPA to Interval	X	*CTX*(BA)	INTCBX(BA, XR)	S
E → X	Extended to Interval	X	*CTX*(EA)	INTCEX(EA, XR)	S
UH → X	Unpacked Hollerith to Interval (width IB)	X	-	INTCHX(HA, IB, XR)	S
I → X	Integer to Interval	X	*CTX*(IA)	INTCIX(IA, XR)	S
R → X	Real to Interval	X	*CTX*(RA)	INTCRX(RA, XR)	S
X → B	Interval to BPA	B	*CTB*(XA)	INTCXB(XA, BR)	S
X → E	Interval to Extended	E	*CTE*(XA)	INTCXE(XA, ER)	D
X → UH	Interval to unpacked Hollerith (width IB)	UH	-	INTCXH(XA, HR, IB)	S
X → I	Interval to Integer	I	*CTI*(XA)	INTCXI(XA, IR)	I
X → R	Interval to Real	R	*CTR*(XA)	INTCXR(XA, RR)	R
SERVICE					
Store	Replacement operator	X	XR = XA	INTSTR(XA, XR)	S
Negate	Unary minus	X	-XA	INTNEG(XA, XR)	S
LOGICAL AND RELATIONAL					
Bad interval	Inf(XA) > Sup(XA)	L	*BAD*(XA)	INTBAD(XA)	L
Element of	BA ε XB	L	BA .*E*. XB	INTELE(BA, XB)	L
Good interval	Inf(XA) ≤ Sup(XA)	L	*OK*(XA)	INTOK (XA)	L
Subset of	XA contained in XB	L	XA.*SUBSET*.XB	INTSBS(XA, XB)	L
Set-equal	Inf(XA) = Inf(XB) and Sup(XA) = Sup(XB)	L	XA .*SEQ*. XB	INTSEQ(XA, XB)	L
Set-greater-equal	Inf(XA) ≥ Inf(XB) and Sup(XA) ≥ Sup(XB)	L	XA .*SGE*. XB	INTSGE(XA, XB)	L
Set-greater	Inf(XA) > Sup(XB)	L	XA .*SGT*.XB	INTSGT(XA, XB)	L
Set-less-equal	Inf(XA) ≤ Inf(XB) and Sup(XA ≥ Sup(XB)	L	XA .*SLE*. XB	INTSLE(XA, XB)	L
Set-less	Sup(XA) < Inf(XB)	L	XA .*SLT*. XB	INTSLT(XA, XB)	L
Set-not-equal	Inf(XA) ≠ Inf(XB) or Sup(XA) ≠ Sup(XB)	L	XA .*SNE*. XB	INTSNE(XA, XB)	L

OPERATION	DEFINITION/EXPLANATION	RESULT TYPE	ROUTINE INVOCATION VIA AUGMENT	DIRECT	ROUTINE TYPE
LOGICAL AND RELATIONAL(continued)					
Superset	XA contains XB	L	XA.*SPRSET*.XB	INTSPS(XA, XB)	L
Value-equal	Inf(XA) = Sup(XA) = Inf(XB) = Sup(XB)	L	XA .*VEQ*. XB	INTVEQ(XA, XB)	L
Value-greater-eq.	Inf(XA) $\geq$ Sup(XB)	L	XA .*VGE*. XB	INTVGE(XA, XB)	L
Value-greater	Inf(XA) > Sup(XB)	L	XA .*VGT*. XB	INTVGT(XA, XB)	L
Value-less-equal	Sup(XA) $\leq$ Inf(XB)	L	XA .*VLE*. XB	INTVLE(XA, XB)	L
Value-less	Sup(XA) < Inf(XB)	L	XA .*VLT*. XB	INTVLT(XA, XB)	L
Value-not-equal	XA does not intersect XB	L	XA .*VNE*. XB	INTVNE(XA, XB)	L
INPUT/OUTPUT					
Read	Read interval vector, formatted	X	-	INTRD (UNIT, FMT, XR, IN)	S
Read, free format	Read next interval from data stream	X	-	INTRDF(UNIT, XR)	S
Write	Write interval vector, formatted	-	-	INTWR (UNIT, FMT, XA, IN)	S
MISCELLANEOUS					
Reduce	Reduce argument of trig function to principal range	X	-	INTRED(XA, IB, XR)	S
Unpack	Unpack packed Hollerith	H	-	INTUPK(HA, HR, IB, IC)	S*
ERROR HANDLING					
Trap	Detect errors in interval package (arguments in COMMON)	-	-	INTRAP	S*

APPENDIX 3
FAULT INFORMATION

FAULT NUMBER	MEANING				ACTION CODE
0	No fault				0
1	Left endpoint -	no fault	Right endpoint -	overflow	4*
2		no fault		infinity	3
3		no fault		underflow	0
4		overflow		no fault	4*
5		overflow		overflow	4*
6		overflow		infinity	3
7		overflow		underflow	4*
8		infinity		no fault	3
9		infinity		overflow	3
10		infinity		infinity	3
11		infinity		underflow	3
12		underflow		no fault	0
13		underflow		overflow	4*
14		underflow		infinity	3
15		underflow		underflow	0
16	Division by zero				3
17	Zero raised to the zero power				1
18	Square root of a negative number				3
19	Logarithm of a nonpositive number				3
20	Underflow during computation of a BPA result				0
21	Overflow during computation of a BPA result				3
22	Intersection of disjoint intervals				3
23	Arc cosine or arc sine argument out of range				3
24	Inverted interval				4
25	Illegal input character				4
26	Illegal input format specification				4
27	Illegal output format specification				1
28	Input string too long				4
29	Illegal or unspecified input unit				4
30	End of file on input unit				1
31	Illegal or unspecified output unit				1
32	Conversion array overflow during base conversion				4†
33	Unrecognized error				4

* Denotes that the fault is logically impossible
† This action should not be changed, since any other action could result in a recursive call on INTRAP from INTCXH.

In the event that a fault occurs, the corresponding action code governs the response of the INTRAP routine. The action codes, and their responses, are:

0 Return to the calling program without taking any action
1 Print error message and return to the calling program
2 Print error message, trace call sequence, and return
3 Print error message, trace call sequence, step error counter in Executive program, and return
4 Print error message, trace call sequence, and halt computation

APPENDIX 4

COMMON BLOCK INFORMATION

This appendix contains information concerning the COMMON blocks used in the INTERVAL package. The information, which is intended to be used both in adaptation of the package to other host systems and in the production use of the package, is presented in tabular form.

The first part of the Appendix is a list of the COMMON block declarations for each of the COMMON blocks. The user who wishes to have access to any of the COMMON blocks for any reason should include these, or equivalent, declarations in the modules from which the COMMON blocks are to be accessed.

The second part of the Appendix is a list of the variables in each COMMON block, the data type of each variable, the function of each variable, and, where appropriate, the value assigned to each variable. Data types are indicated by a single character; the meanings of these characters are as follows:

I - FORTRAN INTEGER
R - FORTRAN REAL
D - FORTRAN DOUBLE PRECISION
L - FORTRAN LOGICAL
B - BPA
E - EXTENDED
X - INTERVAL

The last column indicates whether the variable may be modified at the time of use (U) or at the time of adaptation (A).

The third part of this Appendix is a table listing the modules of the INTERVAL package and indicating which COMMON blocks, if any, are used in each module. This table will be helpful in the case that the dimensions of any arrays are altered in the adaptation of the package to a new host environment. Dimensions of the arrays in these COMMON blocks may not be altered at the time of use.

COMMON BLOCK DECLARATIONS

BPA COMMON BLOCKS

1. COMMON block for defining rounding options and passing information to the BPA package:

```
COMMON /BPACOM/ OPTION, BPAFLT, ACC, RDU, RDL, RDT, RDN, RDA
INTEGER OPTION, BPAFLT, ACC, RDU, RDL, RDT, RDN, RDA
```

2. COMMON block for representation-dependent BPA constants:

```
COMMON /BPACON/ ZRO, ONE, ONM, TWO, PIO2, PI, BPAMNB, BPAMXB
BPA ZRO, ONE, ONM, TWO, PIO2, PI, BPAMNB, BPAMXB
```

3. COMMON block for parameters dealing with accuracy of library functions, and for other constants needed in computing special functions:

```
COMMON /BPAACC/ IACC(24), LNACC, LGACC, SNHACC, TNHACC, EXPMXA,
1               EXPMNA, FRACBD, MAXINT
INTEGER IACC
BPA LNACC, LGACC, SNHACC, TNHACC, EXPMXA, EXPMNA, FRACBD, MAXINT
```

4. COMMON block for passing information needed by BPA conversion routines:

```
COMMON /BPACCM/ EXWDTH, ESDMAX, EXMAX, EXMIN, IX(80), ICX(40),
1               ECX(40), EX(80)
INTEGER EXWDTH, ESDMAX, EXMAX, EXMIN, IX, ICX, ECX, EX
```

INTERVAL COMMON BLOCKS

5. COMMON block for passing parameters to INTRAP:

```
COMMON /INTCOM/ INTFLT, ID, TA, TB, TR
INTEGER INTFLT, ID
INTERVAL TA, TB, TR
EXTENDED TAE, TBE, TRE
EQUIVALENCE (TA, TAE), (TB, TBE), (TR, TRE)
```

6. COMMON block for machine dependent INTERVAL constants and EXTENDED constants for reduction of trig function arguments to principal range:

```
COMMON /INTCON/ PIO2I, PII, TPIO2I, TPII, NIN8TH, EPI, ETPI,
                DELTA, EONE
INTERVAL PIO2I, PII, TPIO2I, TPII, NIN8TH
EXTENDED EPI, ETPI, DELTA, EONE
```

7. COMMON block for INTERVAL I/O routines:

```
      COMMON /INTCCM/ NUNITS, UNITBL(8, 5), NOCHR, ICHR(10), OCHR(4),
     1                LRBFOR, RBFORM(132), LRBFRE, RBFREE(132), IRBFRE,
     2                OLDUNT, LWBFOR, WBFORM(132), SOFMT(4), IWMIN,
     3                NOCCC, CCC(2), KFMT(4), HSTR(132)
      INTEGER NUNITS, UNITBL, NOCHR, ICHR, OCHR, LRBFOR, RBFORM, LRBFRE,
     1        RBFREE, IRBFRE, OLDUNT, LWBFOR, WBFORM, SOFMT, IWMIN,
     2        NOCCC, CCC, KFMT, HSTR
```

8. COMMON block to control echoing of input characters and fields:

```
      COMMON /INTCRD/ ECHOS, ECHOD
      LOGICAL ECHOS, ECHOD
```

9. COMMON block for INTRAP parameters:

```
      COMMON /INTRCM/ MON(40), NAMES(40), TYPA(40), TYPB(40), TYPR(40)
      INTEGER MON, NAMES, TYPA, TYPB, TYPR
```

COMMON BLOCK INFORMATION

BLOCK/ VARIABLE	TYPE	EXPLANATION	VALUE	MODIFY U	MODIFY A
BPACOM		Defines rounding options and provides for communication with the BPA package.			
OPTION	I	Communicate rounding option selected	-	-	-
BPAFLT	I	Communicate BPA error information 0 - no fault 1 - overflow 2 - infinity 3 - underflow 4 - division by zero	-	-	-
ACC	I	Number of digits of EXTENDED number which are accurate (used only for BPACEB)	-	-	-
RDU	I	Rounding specification - upward directed	1	N	N
RDL	I	downward directed	2	N	N
RDT	I	toward zero	3	N	N
RDN	I	nearest number	4	N	N
RDA	I	away from zero	5	N	N
BPACON		Representation-dependent BPA constants			
ZRO	B	Number "0"		N	Y
ONE	B	Number "1"		N	Y
ONM	B	Number "-1"		N	Y
TWO	B	Number "2"		N	Y
PIO2	B	Number "π/2" (upper bound)		N	Y
PI	B	Number "π" (upper bound)		N	Y
BPAMXB	B	Maximum positive BPA number		N	Y
BPAMNB	B	Minimum positive normalized BPA number		N	Y
BPAACC		Parameters specifying accuracy of EXTENDED library functions, and other constants needed in computing special functions			
IACC(24)	I	Accuracy (in internal digits) of EXTENDED library functions. Correspondences are:			
		1 - Sine		N	Y
		2 - Cosine		N	Y
		3 - Tangent		N	Y
		4 - Arc sine		N	Y
		5 - Arc cosine		N	Y
		6 - Arc tangent		N	Y
		7 - Natural logarithm (away from 1)		N	Y
		8 - Natural logarithm (near 1)		N	Y
		9 - Common logarithm (away from 1)		N	Y
		10 - Common logarithm (near 1)		N	Y
		11 - Exponential		N	Y
		12 - Hyperbolic sine (away from 0)		N	Y

COMMON BLOCK INFORMATION (CONTINUED)

BLOCK/ VARIABLE	TYPE	EXPLANATION	VALUE	MODIFY U	MODIFY A
BPAACC		(continued)			
IACC		(continued)			
		13 - Hyperbolic sine (near zero)		N	Y
		14 - Hyperbolic cosine		N	Y
		15 - Hyperbolic tangent (away from zero)		N	Y
		16 - Hyperbolic tangent (near zero)		N	Y
		17 - Square root		N	Y
		18 - Cube root		N	Y
		19 - 24 - reserved for local use		N	Y
LNACC	B	Radius of nbd in which IACC(8) applies		N	Y
LGACC	B	Radius of nbd in which IACC(10) applies		N	Y
SNHACC	B	Radius of nbd in which IACC(13) applies		N	Y
TNHACC	B	Radius of nbd in which IACC(16) applies		N	Y
EXPMXA	B	Max x such that e^x does not overflow		N	Y
EXPMNA	B	Min x such that e^x does not underflow		N	Y
FRACBD	B	Minimum BPA number having its true radix point immediately following the low-order digit of the fraction		N	Y
MAXINT	B	Smallest positive BPA integer that can not be represented in FORTRAN INTEGER format		N	Y
BPACCM		Information and working storage for conversion between BPA and HOLLERITH			
EXWDTH	I	Number of decimal digits needed to express decimal exponent of max BPA number	2	N	Y
ESDMAX	I	Maximum number of significant digits allowed for an external number	60	N	Y
EXMAX	I	Maximum exponent allowed for a normalized external number on input; larger numbers will overflow.	40	N	Y
EXMIN	I	Minimum exponent allowed for a normalized external number on input; smaller numbers will underflow	-40	N	Y
IX(80)	I	Floating point internal base array			
		1 - Dimension of array	80	N	Y
		2 - Base of BPA numbers	2	N	Y
		3 - Power of base of BPA numbers	1	N	N
		4 - Location of radix point in array	12	N	N
		5 - Location of most significant digit	-	-	-
		6 - Location of least significant digit	-	-	-
		7 - Exponent	-	-	-
		8 - Sign	-	-	-
		9 - Number of sig. digits in BPA format	27	N	Y
		Remaining positions are work space	-	-	-

COMMON BLOCK INFORMATION (CONTINUED)

BLOCK/ VARIABLE	TYPE	EXPLANATION	VALUE	MODIFY U	MODIFY A
BPACCM		(continued)			
ICX(40)	I	Fixed point internal-related base array			
		1 - Dimension of array	40	N	Y
		2 - Base	65536	N	Y
		3 - Power of BPA base	16	N	Y
		4 - Location of radix point	21	N	Y
		5 - 8 -- see description of IX	-	-	-
		9 - Number of sig. digits needed	3	N	Y
		Remaining positions are work space	-	-	-
ECX(40)	I	Fixed point external-related base array			
		1 - Dimension of array	40	N	Y
		2 - Base	10000	N	Y
		3 - Power of external base	5	N	Y
		4 - Location of radix point	21	N	Y
		5 - 8 -- see description of IX	-	-	-
		9 - Number of sig. digits needed	-	-	-
		Remaining positions are work space	-	-	-
EX(80)	I	Floating point external base array			
		1 - Dimension of array	80	N	Y
		2 - Base	10	N	N
		3 - Power of external base	1	N	N
		4 - Location of radix point	12	N	N
		5 - 8 -- see description of IX	-	-	-
		9 - Number of sig. digits needed	-	-	-
		Remaining positions are work space	-	-	-
INTCOM		Communication with INTRAP			
INTFLT	I	Error indicator for INTERVAL operations and functions. See Table for possible values	-	-	-
ID	I	Identification of routine calling INTRAP. See description of INTRCM for possible values and their interpretations.	-	-	-
TA	*	First argument to routine calling INTRAP	-	-	-
TB	*	Second argument to routine calling INTRAP	-	-	-
TR	*	Result of routine calling INTRAP	-	-	-
		* TA, TB, and TR are EQUIVALENCEd to variables of the appropriate types			
INTCON		INTERVAL and EXTENDED constants required for trig functions			
PIO2I	X	Interval representation of $\pi/2$		N	Y
PII	X	Interval representation of π		N	Y
TPIO2I	X	Interval representation of $3\pi/2$		N	Y

COMMON BLOCK INFORMATION (CONTINUED)

BLOCK/ VARIABLE	TYPE	EXPLANATION	VALUE	MODIFY U	MODIFY A
INTCON		(continued)			
TPII	X	Interval representation of 2π		N	Y
NIN8TH	X	Interval representation of 9/8		N	Y
EPI	E	EXTENDED representation of π		N	Y
ETPI	E	EXTENDED representation of 2π		N	Y
DELTA	E	EXTENDED representation of relative accuracy of least accurate of INF(A)/ ETPI, INF(A)/EPI, ETPI*<INTEGER>, EPI*<INTEGER>		N	Y
EONE	E	EXTENDED representation of "1" Note: in the UNIVAC 1110 version, all EXTENDED variables are equivalenced to REAL arrays of length 2 when their values are defined by DATA statements.		N	Y
INTCCM		Information and working storage for INTERVAL conversion routines			
NUNITS	I	Number of I/O units recognized	4	Y	Y
UNITBL(8, 5)	I	Information for I/O units. Each row contains information for one unit:			
		Row Col. Meaning			
		1 1 Unit number (standard input)	5	N	Y
		2 Length of record	80	N	Y
		3 Number of chars to be scanned	80	Y	Y
		4 Carriage control does not apply	0	N	N
		5 Unit is read only	1	N	N
		2 1 Unit number (standard print)	6	N	Y
		2 Length of print line	132	N	Y
		3 Number of chars to be printed	132	Y	Y
		4 Carriage control char. applies	1	N	Y
		5 Unit is write only	2	N	N
		3 1 Unit number (standard punch)	1	N	Y
		2 Length of record	80	N	Y
		3 Number of chars to be punched	80	Y	Y
		4 Carriage control does not apply	0	Y	Y
		5 Unit is write only	2	N	N
		4 1 Unit number (standard reread)	0	N	Y
		2 Length of record	80	N	Y
		3 Number of chars to be scannned	80	Y	Y
		4 Carriage control does not apply	0	N	N
		5 Unit is read only	1	N	N
		5 - 8 Available for user definitions			
NOCHR	I	Number of special characters recognized by input scanners	10	N	N

COMMON BLOCK INFORMATION (CONTINUED)

BLOCK/ VARIABLE	TYPE	EXPLANATION	VALUE	MODIFY U	MODIFY A
INTCCM		(continued)			
ICHR(10)	I	Delimiting characters recognized by input scanner			
		1 - Left endpoint delimiter	'('	Y	Y
		2 - Endpoint separator (unconditional)	';'	Y	Y
		3 - Right endpoint delimiter	')'	Y	Y
		4 - Blank	' '	N	N
		5 - Endpoint separator within parens; field separator outside parens	','	Y	Y
		6 - Field delimiter (unconditional)	'='	Y	Y
		7 - " " "	'#'	Y	Y
		8 - " " "	'$'	Y	Y
		9 - Digit zero	'0'	N	N
		10 - Blank	' '	N	N
OCHR(4)	I	Delimiting characters used in interval output			
		1 - Left endpoint delimiter	'('	Y	Y
		2 - Endpoint separator	','	Y	Y
		3 - Right endpoint delimiter	')'	Y	Y
		4 - Blank	' '	N	N
LRBFOR	I	Length of formatted read buffer	132	N	Y
RBFORM(132)	I	Formatted read buffer (work space)	-	-	-
LRBFRE	I	Length of free format read buffer	132	N	Y
RBFREE(132)	I	Free format read buffer (work space)	-	-	-
IRBFRE	I	(Length + 1) of free format read buffer	133	N	Y
OLDUNT	I	Previous unit # used in free format read	-1	N	N
LWBFOR	I	Length of formatted write buffer	132	N	Y
WBFORM(132)	I	Formatted write buffer (work space)	-	-	-
SOFMT(4)	I	Standard output format			
		1 - Fields per record	2	N	Y
		2 - # of blanks preceeding each field	1	N	Y
		3 - Width of each field	37	N	Y
		4 - Carriage control character	' '	N	Y
IWMIN	I	Minimum width for interval output field	15	N	Y
NOCCC	I	Number of carriage control characters recognized by package	2	N	Y
CCC(2)	I	Carriage control characters recognized			
		1 - Single space	' '	N	Y
		2 - Double space	'0'	N	Y
KFMT(4)	I	Format in use (work space)	-	-	-
HSTR(132)	I	Storage for Hollerith string (work space)	-	-	-
INTCRD		Controls echoing of data read by interval read routines. If .TRUE., data will be echoed on standard output unit			
ECHOS	L	Echo source record	.FALSE.	Y	Y
ECHOD	L	Echo data field	.FALSE.	Y	Y

COMMON BLOCK INFORMATION (CONTINUED)

BLOCK/ VARIABLE	TYPE	EXPLANATION	VALUE	MODIFY U	MODIFY A
INTRCM		Information for INTRAP -- fault responses, calling routine names, and argument types			
MON(40)	I	Fault responses; MON(I+1) contains the response to the fault whose indicator value is I. See Table for details.		Y	Y
NAMES(40)	I	Names of routines which may call INTRAP		N	N
TYPA(40)	I	Type of first argument to specified routine		N	N
TYPB(40)	I	Type of second argument to specified routine		N	N
TYPR(40)	I	Type of result of specified routine		N	N

Arguments may be of following types:

0 - No argument
1 - INTEGER
2 - REAL
3 - DOUBLE PRECISION
4 - BPA
5 - EXTENDED
6 - INTERVAL

Values are defined as follows:

INDEX	NAMES	TYPA	TYPB	TYPR
1	ADD	6	6	6
2	SUB	6	6	6
3	MUL	6	6	6
4	DIV	6	6	6
5	SCT	6	6	6
6	XXI	6	1	6
7	SIN	6	0	6
8	COS	6	0	6
9	TAN	6	0	6
10	ASN	6	0	6
11	ACS	6	0	6
12	ATN	6	0	6
13	EXP	6	0	6
14	LN	6	0	6
15	LOG	6	0	6
16	SNH	6	0	6
17	CSH	6	0	6
18	TNH	6	0	6
19	SQT	6	0	6
20	CBT	6	0	6
21	MDB	6	0	4
22	HLB	6	0	4
23	LGB	6	0	4
24	CXI	6	0	1
25	BND	5	1	6
26	CEX	5	0	6

COMMON BLOCK INFORMATION (CONCLUDED)

INDEX	NAMES	TYPA	TYPB	TYPR
27	RD	0	0	0
28	RDF	0	0	0
29	ASG	0	0	0
30	CHX	0	1	6
31	WR	0	0	0
32	CXH	0	1	6
33 - 40	Reserved for local use			

Note: For the interval I/O conversion and service routines, the argument types shown do not agree with the argument types in the calling sequence. This is due to the special nature of the printouts for these routines.

PACKAGE MODULES AND ASSOCIATED COMMON BLOCKS

	BPACOM	BPACON	BPAACC	BPACCM
BLOCKDATA	X	X	X	X
BPAABS		X		
ACS	X	X	X	
ASN	X	X	X	
ATN	X	X	X	
CBD				
CBE				
CBH	X			X
CBI				
CBR				
CBT	X		X	
CHB	X			X
CHI				
CIB	X			
CIH				
COS	X	X	X	
CPD	X			
CRB				
CSB	X	X		
CSD				
CSH	X	X	X	
CSP				
EQ	X			
EXP	X	X	X	
GE	X			
GT	X			
INT		X	X	
LE	X			
LN	X	X	X	
LOG	X	X	X	
LT	X			
MAX				
MIN				
NE	X			
NEG				
SGN				
SIN	X	X	X	
SNH	X	X	X	
SQT	X	X	X	
STR				
TAN	X		X	
TNH	X	X	X	
XBI	X	X		
ADD	X			
CEB	X			
DIV	X			
MUL	X			
SUB	X			

	BPACOM	BPACON	BPAACC	BPACCM	INTCOM	INTCON	INTCCM	INTCRD	INTRCM
BLOCKDATA					X	X	X	X	X
INTABS		X							
ACS	X	X			X				
ADD	X				X				
ASG					X		X		
ASN	X	X			X				
AT2									
ATN	X				X				
BAD									
BND	X				X				
CBT	X				X				
CBX									
CEX	X				X				
CHX	X				X		X		
CIX	X								
COS	X	X			X	X			
CPS									
CRX									
CSH	X				X				
CXB									
CXE									
CXH	X				X		X		
CXI	X				X		X		
CXR									
DIV	X	X			X				
ELE									
EXP	X				X				
HLB	X				X				
INF									
INL									
INT	X								
LGB	X				X				
LN	X				X				
LOG	X				X				
MAG									
MDB	X	X			X				
MIG									
MUL	X	X			X				
NEG									
OK									
PVL	X								
PVO	X								
PVU	X								

	BPACOM	BPACON	BPAACC	BPACCM	INTCOM	INTCON	INTCCM	INTCRD	INTRCM
INTRAP					X		X		X
RD					X		X	X	
RDF					X		X	X	
RED	X	X			X	X			
SBS									
SCT					X				
SEQ									
SGE									
SGN									
SGT									
SIN	X	X			X	X			
SIZ	X	X							
SLE									
SLT									
SNE									
SNH	X				X				
SPL									
SPS									
SQT	X				X				
STR									
SUB	X				X				
SUP									
TAN	X	X			X	X			
TNH	X				X				
UPK							X		
VEQ									
VGE									
VGT									
VLE									
VLT									
VNE									
WR					X		X		
XXB									
XXE									
XXI	X	X			X				
XXX									

APPENDIX 5
DESCRIPTION DECKS

```
*DESCRIBE EXTENDED                                                              DSE00010
DECLARE DOUBLE PRECISION, KIND FUNCTION, PREFIX EXT                             DSE00020
COMMENT DUMMY DESCRIPTION SO COMPILER CAN BE USED                               DSE00030
FUNCTION EXP (*EXP,($),(DOUBLE PRECISION),DOUBLE PRECISION)                     DSE00040

*DESCRIBE BPA                                                                  DSB00010
DECLARE REAL, KIND SAFE SUBROUTINE, PREFIX BPA                                  DSB00020
OPERATOR +  (, NULL UNARY, PRV, $)                                              DSB00030
COMMENT  THE FOLLOWING DESCRIPTOR MUST BE INSERTED IN THE DECK AT THIS          DSB00040
 POINT IF THE REAL UNARY MINUS OF FORTRAN DOES NOT SUFFICE FOR THE BPA          DSB00050
 UNARY MINUS --                                                                 DSB00060
 OPERATOR -  (NEG, UNARY, PRV, $)                                               DSB00070
OPERATOR +  (ADD, BINARY 1, PRV, $, COMM), * (MUL), - (SUB,,,,NONCOMM),        DSB00080
        /  (DIV)                                                                DSB00090
OPERATOR ** (XRI, BINARY 3, PRV, $, INTEGER, $)                                 DSB00100
OPERATOR .EQ. (EQ, BINARY 2, PRV, $, LOGICAL), .NE. (NE), .LT. (LT),            DSB00110
        .LE. (LE), .GT. (GT), .GE. (GE)                                         DSB00120
FUNCTION ABS (ABS, ($), $), SIN (SIN), COS (COS), TAN (TAN), ASIN (ASN),        DSB00130
        ACOS (ACS), ATAN (ATN), LOG10 (LOG), LOG (LN), EXP (),                  DSB00140
        EXP (EXP), SINH (SNH), COSH (CSH), TANH (TNH), SQRT (SQT),              DSB00150
        CBRT (CBT)                                                              DSB00160
FUNCTION INT (INT, ($), $)                                                      DSB00170
FUNCTION MAX (MAX, ($, $), $), MIN (MIN)                                        DSB00180
FUNCTION SGN (SGN, ($), INTEGER)                                                DSB00190
COMMENT  THE FOLLOWING DESCRIPTOR MUST BE INSERTED IN THE DECK AT THIS          DSB00200
 POINT IF THE REAL REPLACEMENT OPERATOR IN FORTRAN WILL NOT SERVE AS THE        DSB00210
 BPA REPLACEMENT OPERATOR                                                       DSB00220
 SERVICE COPY(STR)                                                              DSB00230
CONVERSION CTB (CEB, DOUBLE PRECISION, $, DOWNWARD),                            DSB00240
           CTB (CRB, REAL, $, UPWARD),                                          DSB00250
           CTB (CIB, INTEGER, $, UPWARD),                                       DSB00260
           CTE (CBE, $, DOUBLE PRECISION, UPWARD),                              DSB00270
           CTR (CBR, $, REAL, DOWNWARD),                                        DSB00280
           CTI (CBI, $, INTEGER, DOWNWARD)                                      DSB00290
COMMENT  THE FOLLOWING TWO CARDS ARE USED TO FORCE AUGMENT TO USE THE           DSB00300
 REAL UNARY MINUS OPERATOR OF FORTRAN AS THE BPA UNARY MINUS OPERATOR --        DSB00310
*ENVIRONMENT                                                                    DSB00320
OPERATOR -  (UNARY, PRV, BPA)                                                   DSB00330

*DESCRIBE INTERVAL                                                             DSX00010
COMMENT  IN THIS DESCRIPTION DECK, 'DOUBLE PRECISION' IS USED AS A              DSX00020
 SYNONYM FOR 'EXTENDED'.  IF EXTENDED IS A TYPE OTHER THAN DOUBLE PRE-          DSX00030
 CISION, THE APPROPRIATE CHANGES WILL NEED TO BE MADE IN THIS DECK.             DSX00040
COMMENT  THE DECLARATION OF INTERVAL VARIABLES AS TYPE COMPLEX IS MORE          DSX00050
 EFFICIENT, BUT WAS NOT DONE IN THIS VERSION DUE TO THE TYPE-CHECKING           DSX00060
 FEATURES OF SOME FORTRAN COMPILERS.                                            DSX00070
DECLARE REAL(2), KIND SAFE SUBROUTINE, PREFIX INT                               DSX00080
OPERATOR +  (,NULL UNARY, PRV, $)                                               DSX00090
OPERATOR -  (NEG, UNARY, PRV, $)                                                DSX00100
OPERATOR +  (ADD, BINARY1, PRV, $, COMM), * (MUL), - (SUB,,,,NONCOMM),         DSX00110
        /  (DIV)                                                                DSX00120
OPERATOR ** (XXI, BINARY 3, PRV, $, INTEGER, $), ** (XXB,,,,BPA),               DSX00130
        ** (XXP,,,,DOUBLE PRECISION), ** (XYX,,,,$)                             DSX00140
OPERATOR .VEQ. (VEQ, BINARY 2, .EQ. , $, LOGICAL, COMM), .VNE. (VNE),           DSX00150
        .SEQ. (SEQ), .SNE. (SNE)                                                DSX00160
OPERATOR .VLT. (VLT, BINARY 2, .LT. , $, LOGICAL), .VLE. (VLE),                 DSX00170
        .VGT. (VGT), .VGE. (VGE), .SLT. (SLT), .SLE. (SLE),                     DSX00180
        .SGT. (SGT), .SGE. (SGE)                                                DSX00190
OPERATOR .UNION. (UUN, BINARY 1, .OR., $), .INTSCT. (SCT,, .AND.)               DSX00200
OPERATOR .SUBSET. (SBS, BINARY 2, .EQ., $, LOGICAL), .SPPSET. (SPS)             DSX00210
OPERATOR .E. (ELE, BINARY 3, .EQ., BPA, $, LOGICAL)                             DSX00220
FUNCTION ABS (ABS, ($), $), SIN(SIN), COS (COS), TAN (TAN), ASIN (ASN),         DSX00230
        ACOS (ACS), ATAN (ATN), LOG10 (LOG), LN (LN), EXP (EXP),                DSX00240
        SINH (SNH), COSH (CSH), TANH (TNH), SQRT (SQT), CBRT(CBT),              DSX00250
        LOG (LN)                                                                DSX00260
FUNCTION ATAN2 (AT2, ($, $), $)                                                 DSX00270
FUNCTION OK  (OK, ($), LOGICAL), BAD (BAD)                                      DSX00280
FUNCTION COMPOS (CPS, (BPA, BPA), $)                                            DSX00290
FUNCTION INT (INT, ($), $)                                                      DSX00300
FUNCTION DIST (DST, ($, $), BPA)                                                DSX00310
FUNCTION MDPT (MDP, ($), BPA), BLOTE (BLB), LENGTH (LGB),                       DSX00320
        MAG (MAG), MIG (MIG), PIVL (PVL), PIVP (PVP), SIZE (SIZ)                DSX00330
FUNCTION SGN (SGN, ($), INTEGER)                                                DSX00340
FUNCTION BOUND (BND, (DOUBLE PRECISION, INTEGER), $)                            DSX00350
FIELD SUP (SUP, SDL, ($), BPA), INF (INF, IDL)                                  DSX00360
SERVICE COPY (STR)                                                              DSX00370
CONVERSION CTX (CIX, INTEGER, $, UPWARD),                                       DSX00380
           CTX (CRX, REAL, $, UPWARD),                                          DSX00390
           CTX (CBX, BPA, $, UPWARD),                                           DSX00400
           CTX (CEX, DOUBLE PRECISION, $, UPWARD),                              DSX00410
           CTX (ASG, INTEGER, $, UPWARD),                                       DSX00420
           CTI (CXI, $, INTEGER, DOWNWARD),                                     DSX00430
           CTP (CXB, $, BPA, DOWNWARD),                                         DSX00440
           CTR (CXR, $, REAL, DOWNWARD),                                        DSX00450
           CTE (CXE, $, DOUBLE PRECISION, DOWNWARD)                             DSX00460
```

APPENDIX 6
SAMPLE INTERVAL ARITHMETIC PROGRAM

```
C     THIS PROGRAM READS THE COEFFICIENTS OF A QUADRATIC EQUATION AND
C  COMPUTES THE ROOTS OF THAT EQUATION.
C     THE PROGRAM IS INTENTIONALLY DESIGNED TO BE UNSOPHISTICATED, IN
C  ORDER TO (A) KEEP THE PROGRAM SIMPLE, AND (B) DEMONSTRATE THE
C  ARITHMETIC PACKAGE'S RESPONSE TO ERRORS.
C
C          DECLARATIONS
      INTERVAL TWO, FOUR, Y(3), X(2)
      INTEGER FORMRD(4), FORMWR(4)
      DATA FORMRD /3, 0, 20, 1H /, FORMWR /3, 4, 31, 1H /
C          INITIALIZE
      PRINT 1000
      TWO = '2.0$'
      FOUR = '4.0$'
C          READ NEXT DATA SET -- A = Y(1), B = Y(2), C = Y(3)
   10 CALL INTRD(5, FORMRD, Y, 3)
      PRINT 1001
      CALL INTWR(6, FORMWR, Y, 3)
C          FIND ROOTS
C          COMPUTE ROOT WITH LARGER ABSOLUTE VALUE BY QUADRATIC FORMULA
      IF(SUP(Y(2)) .LT. 0.) GO TO 20
      X(1) = (-Y(2)-SQRT(Y(2)**2-FOUR*Y(1)*Y(3)))/(TWO*Y(1))
      GO TO 30
   20 X(1) = (-Y(2)+SQRT(Y(2)**2-FOUR*Y(1)*Y(3)))/(TWO*Y(1))
C          COMPUTE ROOT WITH SMALLER ABSOLUTE VALUE
   30 X(2) = Y(3)/(Y(1)*X(1))
      PRINT 1002
      CALL INTWR(6, FORMWR, X, 2)
      GO TO 10
 1000 FORMAT('1SOLUTION OF QUADRATIC EQUATIONS')
 1001 FORMAT(//'0COEFFICIENTS', 7X, 'A', 34X, 'B', 34X, 'C')
 1002 FORMAT('0ROOTS', 14X, 'X1', 33X, 'X2')
      END
```

Program as written for processing by AUGMENT precompiler

```
C     THIS PROGRAM READS THE COEFFICIENTS OF A QUADRATIC EQUATION AND
C   COMPUTES THE ROOTS OF THAT EQUATION.
C     THE PROGRAM IS INTENTIONALLY DESIGNED TO BE UNSOPHISTICATED, IN
C   ORDER TO (A) KEEP THE PROGRAM SIMPLE, AND (B) DEMONSTRATE THE
C   ARITHMETIC PACKAGE'S RESPONSE TO ERRORS.
C
C         DECLARATIONS
C               ===== PROCESSED BY AUGMENT, VERSION 4N =====
C               ----- TEMPORARY STORAGE LOCATIONS -----
C                     BPA
      REAL BPATMP(2)
C                     INTERVAL
      REAL INTTMP(2,3)
C               ----- LOCAL VARIABLES -----
      INTEGER FORMRD(4), FORMWR(4)
C                     INTERVAL
      REAL FOUR(2), TWO(2), X(2,2), Y(2,3)
C               ----- SUPPORTING PACKAGE FUNCTIONS -----
      LOGICAL BPALT
C               ===== TRANSLATED PROGRAM =====
      DATA FORMRD /3, 0, 20, 1H /, FORMWR /3, 4, 31, 1H /
C         INITIALIZE
      PRINT 1000
      CALL INTASG ('2.0$',TWO)
      CALL INTASG ('4.0$',FOUR)
C         READ NEXT DATA SET -- A = Y(1), B = Y(2), C = Y(3)
   10 CALL INTRD(5, FORMRD, Y, 3)
      PRINT 1001
      CALL INTWR(6, FORMWR, Y, 3)
C         FIND ROOTS
C         COMPUTE ROOT WITH LARGER ABSOLUTE VALUE BY QUADRATIC FORMULA
      CALL INTSUP (Y(1,2),BPATMP(1))
      CALL BPACRB (3.,BPATMP(2))
      IF (BPALT (BPATMP(1),BPATMP(2)))  GO TO 20
      CALL INTNEG (Y(1,2),INTTMP(1,1))
      CALL INTXXI (Y(1,2),2,INTTMP(1,2))
      CALL INTMUL (FOUR,Y(1,1),INTTMP(1,3))
      CALL INTMUL (INTTMP(1,3),Y(1,3),INTTMP(1,3))
      CALL INTSUB (INTTMP(1,2),INTTMP(1,3),INTTMP(1,3))
      CALL INTSQT (INTTMP(1,3),INTTMP(1,3))
      CALL INTSUB (INTTMP(1,1),INTTMP(1,3),INTTMP(1,3))
      CALL INTMUL (TWO,Y(1,1),INTTMP(1,1))
      CALL INTDIV (INTTMP(1,3),INTTMP(1,1),X(1,1))
      GO TO 30
   20 CALL INTNEG (Y(1,2),INTTMP(1,1))
      CALL INTXXI (Y(1,2),2,INTTMP(1,2))
      CALL INTMUL (FOUR,Y(1,1),INTTMP(1,3))
      CALL INTMUL (INTTMP(1,3),Y(1,3),INTTMP(1,3))
      CALL INTSUB (INTTMP(1,2),INTTMP(1,3),INTTMP(1,3))
      CALL INTSQT (INTTMP(1,3),INTTMP(1,3))
      CALL INTADD (INTTMP(1,1),INTTMP(1,3),INTTMP(1,3))
      CALL INTMUL (TWO,Y(1,1),INTTMP(1,1))
      CALL INTDIV (INTTMP(1,3),INTTMP(1,1),X(1,1))
C         COMPUTE ROOT WITH SMALLER ABSOLUTE VALUE
   30 CALL INTMUL (Y(1,1),X(1,1),INTTMP(1,1))
      CALL INTDIV (Y(1,3),INTTMP(1,1),X(1,2))
      PRINT 1002
      CALL INTWR(6, FORMWR, X, 2)
      GO TO 10
 1000 FORMAT('1SOLUTION OF QUADRATIC EQUATIONS')
 1001 FORMAT(//'0COEFFICIENTS', 7X, 'A', 34X, 'B', 34X, 'C')
 1002 FORMAT('0ROOTS', 14X, 'X1', 33X, 'X2')
      END
```

Program as it would be written for the FORTRAN compiler (this sample generated by AUGMENT)

```
SOLUTION OF QUADRATIC EQUATIONS

COEFFICIENTS       A                                B                                C
    ( .100000000+01, .100000000+01)    ( .200000000+01, .200000000+01)    ( .100000000+01, .100000000+01)
ROOTS              X1                               X2
    (-.100000000+01,-.100000000+01)    (-.100000000+01,-.100000000+01)


COEFFICIENTS       A                                B                                C
    ( .100000000+01, .100000000+01)    ( .300000000+01, .300000000+01)    ( .100000000+01, .100000000+01)
ROOTS              X1                               X2
    (-.261803401+01,-.261803397+01)    (-.381966014+00,-.381966006+00)


COEFFICIENTS       A                                B                                C
    ( .100000000+01, .100000000+01)    ( .300000000+01, .300000000+01)    ( .200000000+01, .200000000+01)
ROOTS              X1                               X2
    (-.200000003+01,-.199999998+01)    (-.100000002+01,-.999999985+00)


COEFFICIENTS       A                                B                                C
    ( .100000000+01, .100000000+01)    ( .300000000+01, .300000000+01)    ( .300000000+01, .300000000+01)

**********************************************************************************************************************************
SQUARE ROOT OF A NEGATIVE NUMBER DURING INTSQT
ARG1 = (-.3000000000+01,-.3000000000+01) = (575177777777, 575177777777)
RES  = (-.3000000000+01,-.3000000000+01) = (575177777777, 575177777777)
INTRAP CALLED AT  SEQUENCE NUMBER   000124  OF  INTSQT
INTSQT CALLED AT  SEQUENCE NUMBER   000132  OF  MAIN PROGRAM
**********************************************************************************************************************************
REACHED ERROR LIMIT OF     1.  JOB TERMINATED.  CONTROL TRANSFERRED TO EXIT WITH ERROR FLAG SET.
**********************************************************************************************************************************
```


Output from sample Interval Arithmetic program

APPENDIX 7

RUNSTREAM FOR THE MRC UNIVAC 1110 VERSION

```
@ASG,A MRC*LIB.
@XQT MRC*LIB.AUGMENT
@ADD MRC*LIB.DESCRIPTION/INTERVAL
@ADD MRC*LIB.DESCRIPTION/FORTRAN-V
*BEGIN
*CONVERT EXTENDED - DOUBLE PRECISION
:FOR,options filename1.elementname,filename2.elementname,filename3.elementname
          (standard control card except for ':' in Column 1)
     source program module
:FOR,options etc.
     source program module

          .
          .
          .
*END
@ADD 20.
     (any other processing, such as compilation of nonAUGMENTed modules,
      @PREP of files, etc.)
@MAP*MAP.MAP,I
     IN TPF$.
     IN MRC*LIB.BPACOMMON
     IN MRC*LIB.INTCCOM
     LIB MRC*LIB.
@XQT
     (data)
@FIN
```

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: 1755	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: THE INTERVAL ARITHMETIC PACKAGE	5. TYPE OF REPORT & PERIOD COVERED: Summary Report - no specific reporting period
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s): J. M. Yohe	8. CONTRACT OR GRANT NUMBER(s): DAAG29-75-C-0024
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Mathematics Research Center, University of Wisconsin 610 Walnut Street Madison, Wisconsin 53706	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: Work Unit #8 Computer Science
11. CONTROLLING OFFICE NAME AND ADDRESS: U. S. Army Research Office P.O. Box 12211 Research Triangle Park, North Carolina 27709	12. REPORT DATE: June 1977 13. NUMBER OF PAGES: 81
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution 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)*

Interval Arithmetic Program Package

Portable software

User documentation

Technical documentation

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

This report provides user documentation and technical documentation for a package of FORTRAN subroutines for performing interval arithmetic calculations. Apart from a relatively small number of primitives and constants, the package is directly transferrable to most large scale computing systems, and information for implementing the package on other systems is included. The implementation described herein is that for UNIVAC 1110, but the package has also been implemented on CDC, DEC, Honeywell, and IBM equipment.

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

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*