LBL-9881 c, 2
UC-4
CONF-790779

NRCC NATIONAL RESOURCE FOR COMPUTATION IN CHEMISTRY

SOFTWARE STANDARDS IN CHEMISTRY

Proceedings
of the Conference held at University of Utah
July 25-27, 1979

NRCC Proccedings No. 7

LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA

Prepared for the U.S. Department of Energy under Contract W-7405-ENG-48 and for the National Science Foundation under Interagency Agreement CHE-7721305

PROCEEDINGS

of the conference

SOFTWARE STANDARDS IN CHEMISTRY

Sponsored by the

NATIONAL RESOURCE FOR COMPUTATION IN CHEMISTRY

Lawrence Berkeley Laboratory
Berkeley, California 94720

Held at
the University of Utah
July 25-27, 1979

NRCC Proceedings No. 7

Table of Contents

FOREWORD

The National Resource for Computation in Chemistry (NRCC) was established as a division of Lawrence Berkeley Laboratory (LBL) in October 1977. The functions of the NRCC may be broadly categorized as follows: (1) to make information on existing and developing computational methodologies available to all segments of the chemistry community, (2) to make state-of-the-art computational facilities (both hardware and software) accessible to the chemistry community, and (3) to foster research and development of new computational methods for application to chemical problems.

Conferences are one facet of the NRCC's program for both obtaining and making available information on new developments in computationally oriented subdisciplines of chemistry. The goal of this conference was to discuss and recommend standards for machine-independent, modular, and well documented software suitable for use on the large minicomputers that are rapidly becoming available to chemists.

The conference began with a plenary session comprising talks of general interest on various aspects of portability and standardization. There was general agreement among the speakers as to the virtues of structured programming and the value of software tools in attaining good programming practice and portability. Fortran preprocessors were recommended both as an aid to producing structured programs and as a means of attaining machine-independent code. Verification programs, such as PFORT, were suggested as tools for testing existing code for portability. Two talks in this session dealt with examples of applications in the two main areas of interest. Dr. G. Diercksen of the Max-Planck Institute for Physics and Astrophysics (Munich) reported on his system of data interfaces and data bases for quantum chemistry, and Prof. James Stewart of the University of Maryland talked about his efforts in the area of portable crystallographic software.

After the general session, the participants divided into several smaller working groups in order to discuss topics and formulate recommendations relevant to their own interests. The topics of the working groups were:

1. Portability considerations and standards
2. Documentation and coding standards
3. Machine specification: what is a minimal computer configuration?
4. Quantum chemistry software base
5. Quantum chemistry data interfaces
6. Small molecule crystallography
7. Large molecule crystallography

Each working group formulated a preliminary report, presented it to a general session for discussion, and then reconvened to draft a final report.

The present volume consists of the edited summaries of the invited talks, final reports of the working groups, and reports generated by subcommittees of the working groups.

A notable aspect of the conference was the confluence of quantum chemists and crystallographers. Professor George Jeffery of the University of Pittsburgh expressed the hope of growing exchange between the two groups. In his recollection, this was the first such joint gathering to take place in over 20 years. It was felt that such exchanges could lead to agreement on common formats for related chemical structural data bases, and could be of substantial benefit to both fields.

The crystallographers, who initially divided into small- and large-molecule working groups, reconvened as a single working group to draft their final report. There was widespread agreement among them on the steps that both the NRCC and the crystallographic community should take in attaining software standardization and machine-independent portability. Several of their recommendations focused on the demonstration of the feasibility of standardized programming systems and data formats through the NRCC sponsorship of workshops to produce useful crystallographic code. They settled upon a specific preprocessor language and set of program conventions and data file formats to be used in such pilot demonstrations.

In the area of quantum chemistry, the working group on the quantum chemistry software base drafted a list of program types and program capabilities that should be included in a software base. The working group on standard data interfaces prepared a preliminary draft of a standard data interface for this software base. A committee was then established to develop a detailed standard data interface. This committee, which consists of authors and users of major quantum chemistry systems such as GAUSSIAN 70, ALIS, and MUNICH, will make final recommendations within a year.

Both the crystallographers and the quantum chemists felt that a standard set of bit primitives should be defined, and a committee was established to accomplish this. They also agreed on the need for upward compatibility of new versions of ANSI Fortran. They recommended that NRCC communicate this position to the ANSI Standards Committee X3J3, and also sponsor a member on this committee who would lobby for the interests of computational chemists.

The NRCC is indebted to Prof. Frank E. Harris of the University of Utah for helping to organize this workshop, to the University of Utah for making their facilities available, and to Dr. Nelson H. F. Beebe for his efforts in preparing and editing the final conference report. We also thank Drs. Arthur J. Olson and John J. Wendoloski of the NRCC for their efforts in organizing this volume.

The National Resource for Computation in Chemistry is supported in part by a grant from the National Science Foundation (Grant No. CHE-7721305) and the Basic Energy Sciences Division of the U.S. Department of Energy (Contract No. W-7405-ENG-48).

William A. Lester, Jr.
Director, NRCC

EDITORS' NOTE

The report which follows is the final version of the Proceedings of the NRCC Conference on Software Standards in Chemistry held at the University of Utah in Salt Lake City, July 25-27, 1979. The time available for the Conference was rather short, and the last morning was devoted to a meeting in which each of the working groups presented their final reports. Some of these evoked intense discussions, and it was felt advisable that the participants should be able to review the Proceedings before they were published for the general public. In addition, some topics were assigned to subcommittees who were requested to prepare written reports for this review and for the final Proceedings.

Consequently, a preliminary version of the Proceedings was prepared and distributed to participants in late November, along with a request that comments and criticisms be communicated to the editors by mid-December. Many helpful comments were received, and we wish to thank all of those who responded. This review has enabled us to improve and clarify the presentation in a number of sections, and also to remove typographical errors which had escaped our careful (we thought) proofreading. Almost the entire manuscript has been prepared in machine-readable form. The DEC SPELL utility has been used to check for spelling errors, and the text has been formatted for publication by the DOCUMENT utility. All of the editing has been carried out on the DECSYSTEM-20 at the University of Utah, but the final manuscript has been formatted and printed on the NRCC VAX-11/780 computer.

A substantial portion of the time allotted for the Conference was devoted to sessions of six working groups. Participants were free to move from session to session, and many took the opportunity to do so. No record was kept of who contributed exactly what, so except for subcommittee reports, only the name of the chairman appears at the end of each working group report. The chairmen had the responsibility of formulating the reports, but in most cases many people contributed to the final versions presented here. As in any group of people with similar interests, it was not always possible to obtain unanimous agreement, and these Proceedings should be regarded as a broad consensus from which individual participants are free to dissociate themselves.

In addition to these final proceedings, a set of preliminary working papers was distributed to the participants. Final versions for most of these documents are included in these Proceedings. Not included, however, is the Bibliography on Software Portability, and the Programmer's Guide to Portable Software, both by N.H.F. Beebe, and SFTRAN3, Programmers Reference Manual, by C.L. Lawson and J.A. Flynn. These documents are available upon request from the NRCC.

Frank E. Harris
Nelson H.F. Beebe

CONFERENCE PARTICIPANTS

Lawrence Andrews
15707 37th Street NE
Seattle, Washington 98155
(206) 364-9698 (home)
237-7880 (office)

J. Stephen Binkley
Mellon Institute
4400 Fifth Avenue
Pittsburgh, PA 15213
(412) 621-1133
578-3132

D.S. Boudreaux
Inorganic and Solid
State Chemistry
Allied Chemical Corporation
P.O. Box 1021R
Morristown, NJ 07960

Ernest R. Davidson
Department of Chemistry
University of Washington
Seattle, WA 98195
(206) 543-1767

Geerd Diercksen
Max Planck Institut fur
Physik und Astrophysik
Fohringer Ring 6
8000 Munchen 40, West Germany
EUROPE
(089) 32 70 01 (Institut)
96 76 22 (home)

David J. Duchamp
Physical and Analytical
Division
Upjohn Company
301 Henriette St.
Kalamazoo, MI 49001
(616) 323-4000

Thom Dunning
Chemistry Division
Argonne National Laboratory
Argonne, IL 60439
(312) 972-3594
972-3576

Clifford Dykstra
Department of Chemistry
University of Illinois
Urbana, IL 61801
(217) 333-6589

Stephen T. Elbert
Ames Laboratory
US Department of Energy
Iowa State University
Ames, IA 50010
(515) 294-2582

Stephen T. Freer
Department of Chemistry B-017
University of California
at San Diego
La Jolla, CA 92093
(714) 452-2427

Eric Gabe
Division of Chemistry
National Research Council
of Canada
Ottawa, Ontario K1A 0R6
CANADA
(613) 993-2527

Allen C. Larson
P.O. Box 5898
Santa Fe, NM 87502
(preferred mailing address)

James E. George
411 Cheryl
Los Alamos, NM 87544
(505) 672-9688 (home)
(714) 452-0170 (San Diego)
(415) 326-7300 (Livermore)
(505) 667-7356 (LASL)

Charles L. Lawson
MS125-128
Jet Propulsion Laboratory
California Inst. of Technology
Pasadena, CA 91103
(213) 354-4266
-4761 (alternate)

Jonathan Hanson
Biology Division
Brookhaven National Laboratory
Upton, NY 11973
(516) 345-3422

A. Douglas McLean
Department K34/281
IBM Research Laboratory
5600 Cottle Road
San Jose, CA 95193
(408) 256-2674

G.A. Jeffrey
Department of Crystallography
University of Pittsburgh
Pittsburgh, PA 15260

Cleve B. Moler
Dept. of Math. and Statistics
University of New Mexico
Albuquerque, NM 87106
(505) 277-4110 (office)
268-8631 (home)

G.G. Johnson, Jr.
164 Materials Research Lab.
Pennsylvania State University
University Park, PA 16802
(814) 865-1637 (office)
238-0608 (home)

Robert J. Munn
Computer Science Center
University of Maryland
College Park, MD 20740
(301) 454-2620

George D. Purvis, III
Battelle Memorial Institute
505 King Avenue
Columbus, OH 43201
(614) 424-7276

Richard C. Raffenetti
Applied Mathematics Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439
(312) 972-3584

George N. Reeke, Jr.
The Rockefeller University
1230 York Avenue
New York, NY 10021
(212) 360-1339

Edward S. Sachs
Science Applications Inc.
Suite 901
1211 West 22 St
Oak Brook, IL 60521

Norman Schryer
Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
(201) 582-3000 (Lab)
582-2912 (office)
(201) 464-1541 (home)

Isaiah Shavitt
Battelle Memorial Institute
505 King Avenue
Columbus, OH 43201
(614) 424-7293

Robert L. Snyder
Division of Engineering
and Science
New York State College of
Ceramics at Alfred University
Alfred, NY 14802
(607) 871-2438

James M. Stewart
Computer Science Center
University of Maryland
College Park, MD 20740
(301) 454-4623

Lynn F. TenEyck
Institute of Molecular Biology
University of Oregon
Eugene, OR 97403
(503) 686-5151

Keith D. Watenpaugh
Department of Biological
Structure SM-20
University of Washington
Seattle, WA 98195
(206) 543-1421

CONFERENCE STAFF

Nelson H. F. Beebe
Department of Physics
University of Utah
Salt Lake City, UT 84112
(801) 581-5254

Amy Dominguez
Department of Physics
University of Utah
Salt Lake City, UT 84112
(801) 581-8626

John W. Downing
Department of Chemistry
University of Utah
Salt Lake City, UT 84112
(801) 581-7486

Jane Greer
NRCC
Lawrence Berkeley Laboratory
Building 50D
Berkeley, CA 94720
(415) 486-6722

Stan Hagstrom
NRCC
Lawrence Berkeley Laboratory
Building 50D, Room 135
Berkeley, CA 94720
(415) 486-6722

Frank E. Harris
Department of Physics
University of Utah
Salt Lake City, UT 84112
(801) 581-8445

William A. Lester, Jr.
NRCC
Lawrence Berkeley Laboratory
Building 50D, Room 129
Berkeley, CA 94720
(415) 486-6722

Hendrik J. Monkhorst
Quantum Theory Project
Williamson Hall
University of Florida
Gainesville, FL 32611
(904) 392-1597

Arthur Olson
NRCC
Lawrence Berkeley Laboratory
Building 50D, Room 141
Berkeley, CA 94720
(415) 486-6316

Ron Shepard
Department of Chemistry
University of Utah
Salt Lake City, UT 84112
(801) 581-8479

Dale Spangler
NRCC
Lawrence Berkeley Laboratory
Building 50D, Room 111
Berkeley, CA 94720
(415) 486-6316

John Wendoloski
NRCC
Lawrence Berkeley Laboratory
Building 50D, Room 121
Berkeley, CA 94720
(415) 486-6316

CONFERENCE PROGRAM

NRCC Conference on Software Standards in Chemistry

Final Program

University of Utah
Salt Lake City, Utah 84112

July 25-27, 1979

= =

Wednesday, July 25, 1979

8:30 am	Registration and coffee in Carlson Hall lounge (corner of University St and 4-th South on University of Utah campus). Sessions in adjacent Room 115. Coffee will be available in the lounge during all the sessions, and additional refreshments may be available at scheduled coffee break times.
9:00 am	W.A. Lester WELCOMING REMARKS.

S E S S I O N 1. A.D. McLean, Chairperson

9:20 am	C.B. Moler ROBUST AND PORTABLE MATHEMATICAL SOFTWARE - EISPACK, LINPACK, AND OTHERS.
10:00 am	C.L. Lawson STRUCTURED PROGRAMMING PREPROCESSORS .
10:40 am	C O F F E E B R E A K

S E S S I O N 2. I. Shavitt, Chairperson

11:10 am	N. Schryer PORTABLE SOFTWARE TOOLS.
11:50 am	J.E. George PORTABLE GRAPHICS SOFTWARE AND GRAPHICS STANDARDIZATION.
12:30 pm	L U N C H

S E S S I O N 3. G.A. Jeffrey, Chairperson

2:00 pm	G.H.F. Diercksen DATA INTERFACES AND DATA BASES FOR CHEMISTRY.
2:40 pm	J.M. Stewart PORTABLE SOFTWARE, WITH EMPHASIS ON CRYSTALLOGRAPHY
3:20 pm	C O F F E E B R E A K

S E S S I O N 4.

3:50 pm	Introduction and formation of working groups.
4:00 pm	Working group sessions for WG1 (portability standards), WG2 (coding and documentation standards), and WG3 (machine specifications).
6:00 pm	Reception (transportation provided). Participants make own dinner arrangements.
Evening:	Informal discussion of WG1, WG2, and WG3 subjects and report preparation.

= =

Thursday, July 26, 1979

S E S S I O N 1. K. Watenpaugh, Chairperson

8:30 am	Preliminary reports from WG1, WG2, and WG3 and open discussion.
9:30 am	Sessions of WG4 (quantum chemistry software base), WG5 (quantum chemistry data interface), WG6 (small molecule crystallography), and WG7 (large molecule crystallography). [Coffee available during sessions].
12:30 pm	L U N C H

S E S S I O N 2. C. Dykstra, Chairperson

2:00 pm	Preliminary reports from WG4 and WG5 and open discussion.

S E S S I O N 3. J. Hanson, Chairperson

2:45 pm	Preliminary reports from WG6 and WG7 and open discussion.
3:30 pm	Session ends.

E X C U R S I O N T O U R A N D

S O C I A L E V E N I N G

4:00 pm	Transportation from dormitory to Snowbird.
4:45 pm	Arrival at Snowbird.
7:00 pm	Cocktail hour.
8:00 pm	D I N N E R

= =

Friday, July 27, 1979

S E S S I O N 1.

8:30 am	Final working group sessions.
10:00 am	C O F F E E B R E A K

S E S S I O N 2. E.R. Davidson, Chairperson

10:30 am	Final reports of working groups and open discussion.
12:30 pm	CLOSING REMARKS
1:00 pm	L U N C H
2:30 pm	Tour of Evans and Sutherland Computer Graphics Laboratory (transportation provided). Tour time approximately 2 hr. Allow about 45 minutes to reach airport from Laboratory.

= =

WORKING GROUPS

WG1	Portability considerations and standards. Chairperson: S. Elbert
WG2	Documentation and coding standards. Chairperson: C.B. Moler.
WG3	Machine specification: What is a minimal computer configuration? Chairperson: S. Hagstrom
WG4	Quantum chemistry software base. Chairperson: T. Dunning
WG5	Quantum chemistry data interfaces. Chairperson: G.D. Purvis

WG6	Small Molecule Crystallography. Chairperson: E. Gabe
WG7	Large Molecule Crystallography. Chairperson: S. Freer

ROBUST MATHEMATICAL SOFTWARE

Cleve B. Moler

Department of Mathematics and Statistics
University of New Mexico
Albuquerque, NM 87106

This note covers two topics very briefly:

--The LINPACK and EISPACK matrix software packages,

--Machine-dependent constants in numerical calculations.

LINPACK is a collection of FORTRAN subroutines for analyzing and solving various types of simultaneous linear equations. Under development for three years at Argonne National Laboratory and three universities, its public release was announced in January, 1979.

The package consists of 40 subroutines in each of four data types. Three of the data types -- single precision, double precision and complex -- are standard. The fourth -- double precision complex -- is not standard, but is available on many machines. Some of the highlights and novel features of the package include:

--Complete portability and machine independence. There are no precision constants or machine-dependent parameters of any kind. The programs for the three standard data types use the PFORT subset of standard FORTRAN.

--Uniform subroutine naming convention with names that indicate the computation done by the subroutine, rather than the method used to do it.

--Source code formatted by the TAMPR system so that it is easy for human readers to understand.

--Column orientation to enhance performance in virtual memory environments.

--Use of the BLAS, or Basic Linear Algebra Subprograms, to improve modularity of source code and to enhance performance on large matrices and sophisticated machine architectures.

--Basic documentation included as comments in source code.

--Extensive users' manual including examples, detailed descriptions of algorithms, discussion of numerical properties and timing results.

--Comprehensive test drivers distributed with source code.

--Fields tests completed on over 20 different machines before release.

--New algorithms included for estimating matrix condition and hence estimating accuracy of computed results.

--New algorithms implemented for symmetric, but not positive definite, matrices.

--New algorithms included for least squares solutions of overdetermined systems of equations.

--Routines included for band matrices, both symmetric and nonsymmetric, for triangular matrices, and for tridiagonal matrices.

--Complex matrices treated on an equal footing with real matrices.

--Careful programming and some new techniques used to avoid most overflows and destructive underflows. The users' guide [1] is a 368-page, paperbound book available from the Society for Industrial and Applied Mathematics, 33 South 17th St., Philadelphia. The BLAS are described in [2].

EISPACK is a collection of FORTRAN subroutines for solving various matrix eigenvalue problems. Most of the package consists of extensively-tested translations of Algol procedures developed by J. H. Wilkinson, C. Reinsch and their colleagues. A few additional programs, not in the Wilkinson-Reinsch Algol collection, are also included.

EISPACK was initially released in 1973 and a second, greatly-expanded, version released in 1976.

Different programs are included for real symmetric matrices, real general matrices, complex Hermitian matrices, complex general matrices and a few special types of matrices. Different programs are also included for handling situations when only a few of the eigenvalues or eigenvectors are required.

The source code includes machine-dependent constants defining accuracy and radix, but otherwise is portable. Versions are available which include the constants appropriate for IBM, CDC, Univac, Honeywell and a few other machine lines.

On IBM machines only, a special high-level control program known as EISPAC is also available. This program simply requires the user to specify a problem and matrix type. It dynamically loads the appropriate EISPACK routines.

Execution time efficiency and performance in paging operating systems were not the most important considerations during the development of EISPACK. It is usually quite satisfactory in this respect, but some effort is now underway to make some efficiency-oriented improvements in some of the key subroutines.

Basic documentation for EISPACK is included in source code comments, and in separate files on the distribution tape. In addition, a users' manual comes in two parts, [3] and [4]. The source tapes for both LINPACK and EISPACK are available from either of two sources:

National Energy Software Center
Argonne National Laboratory
Argonne, IL 60439

International Mathematical and Statistical Libraries, Inc.
Sixth Floor, GNB Building
7500 Bellaire Boulevard
Houston, TX 77036

From the point of view of portable software production in general, there are significant differences between the two packages. EISPACK, at least initially, was primarily an Algol-to-FORTRAN translation project. Since the Algol procedures included machine-dependent parameters, the same parameters occur in the FORTRAN. Consequently, there are different versions of EISPACK for different machines. The most common such parameter is "MACHEPS", the distance from 1.0 to the next largest floating-point number. A typical use of MACHEPS is in testing that the off-diagonal elements of a matrix being diagonalized are negligible. This might be accomplished with a test like

```
IF (ABS(A(I,J)) .LE. MACHEP*ANORM) ...
```

There are other uses of MACHEPS in EISPACK, but this is the most important.

LINPACK avoids the portability problems associated with machine-dependent constants by changing the form of these tests. The neglibility of a matrix element could be tested by

```
TEST = ANORM + ABS(A(I,J))
IF (TEST .EQ. ANORM) ...
```

Numerically, these two forms of the test have essentially the same effect, but the second form is portable whereas the first is not.

It has been suggested that the second form of the test may encounter difficulties on machines where the arithmetic register is longer than the storage word, but our LINPACK test results indicate this is not a problem. In fact, such machines present a serious question as to what numerically negligible actually means.

This kind of test can be made less stringent by using code similar to the following:

```
BIG = FLOAT(10*N)*ANORM
TEST = BIG + ABS(A(I,J))
IF (TEST .EQ. BIG) ...
```

I have used this kind of convergence test in many different numerical algorithms on many different computers. I strongly prefer it over the other alternatives such as constants in argument lists or data statements, insertion of constants by preprocessors, or reference to library functions.

I recommend that anyone interested in portability of numerical software seriously attempt to formulate algorithms in such a way that this kind of machine-independent test can be incorporated.

Additional discussion and more examples can be found in an elementary numerical methods textbook [5].

REFERENCES

[1] J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart, "LINPACK Users' Guide", SIAM Publications, 1979.

[2] C.L. Lawson, R.J. Hanson, D.R. Kincaid and F.T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," Assoc. Comput. Mach. Trans. on Math. Software,

5, 308-323 (1979). (Also available as Sandia Laboratory Report SAND 77-0898, Albuquerque, 1977.)

[3] B.T. Smith, J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema and C.B. Moler, "Matrix Eigensystem Routines -- EISPACK Guide", Springer Lecture Notes in Computer Science, Vol. 6, 2nd edition, 1976.

[4] B.S. Garbow, J.J. Dongarra, C.B. Moler and B.T. Smith, "Matrix Eigensystem Routines-- EISPACK Guide Extension", Springer Lecture Notes In Computer Science, Vol. 51, 1977.

[5] G.E. Forsythe, M.A. Malcolm and C.B. Moler, "Computer Methods for Mathematical Computations", Prentice-Hall, Inc., 1977.

Structured Programming in the Fortran Environment Using The SFTRAN3 Preprocessor

Charles L. Lawson

Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

The research described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under NASA Contract NAS7-100.

<u>STRUCTURED PROGRAMMING</u>

PURPOSE: <u>FIT THE PROGRAMMING PROCESS BETTER TO HUMAN CAPABILITIES.</u>

AIDS THE HUMAN ACTIVITIES OF

DESIGNING

IMPLEMENTING

READING AND UNDERSTANDING

MAINTAINING AND

MODIFYING PROGRAMS.

STRUCTURED PROGRAMMING & STRUCTURED FORTRAN

A VERY BRIEF HISTORY

- 1966 COMM ACM BOEHM & JACOPINI
- 1968 COMM ACM DIJKSTRA, "GO TO STATEMENT CONSIDERED HARMFUL"
- 1970 - 74 STRUCTURED FORTRAN PREPROCESSORS APPEAR
- 1975 - 76 MEISSNER & REIFER SURVEY:
  OVER 60 STRUCTURED FORTRAN PREPROCESSORS.
  PREPROCESSORS BEING USED ROUTINELY
  IN MANY FORMALLY MANAGED
  PROGRAMMING PROJECTS.

Structured Programming

1. Modularity.

   Each module (subprogram, procedure, etc.) no longer than one printed page, i.e. 20 to 50 lines.

2. Top-down organization of code.

3. Use of single-entry, single-exit control structures.

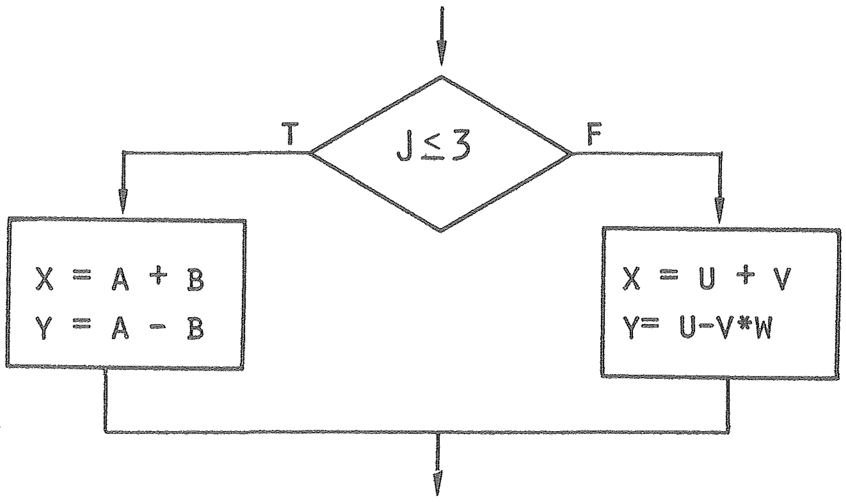
Top-down organization of code

```
DO (INPUT)
DO UNTIL (QUIT)
    DO (COMPUTE)
    DO (OUTPUT)
END UNTIL
```

```
DO FOREVER
    DO (INPUT)
    IF (QUIT) EXIT
    DO (COMPUTE)
    DO (OUTPUT)
END FOREVER
```

Single-entry, single-exit control structures

```
DO WHILE (     )
|
|    IF (    ) THEN
|    |
|    |    DO FOR I = 1, 10
|    |    |
|    |    END FOR
|    |
|    ELSE
|    |
|    |    DO WHILE (    )
|    |    |
|    |    END WHILE
|    |
|    END IF
|
END WHILE
```

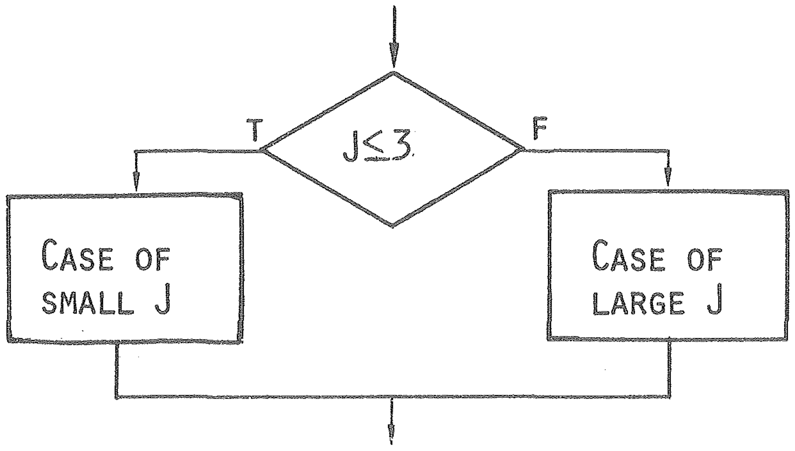


FORTRAN 66

```
      IF ( J .LE.  3) GO TO 10
         X = U + V
         Y = U - V * W
         GO TO 20
10       X = A + B
         Y = A - B
20    CONTINUE
```

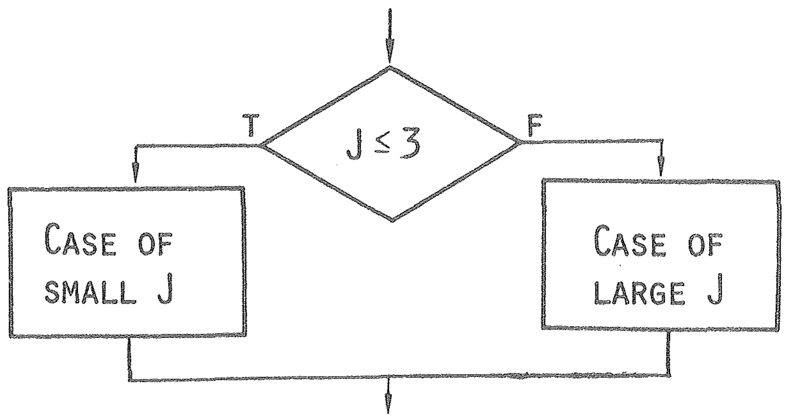
FORTRAN 77 OR SFTRAN3

```
      IF ( J .LE. 3 ) THEN
         X = A + B
         Y = A - B
      ELSE
         X = U + V
         Y = U - V * W
      END IF
```



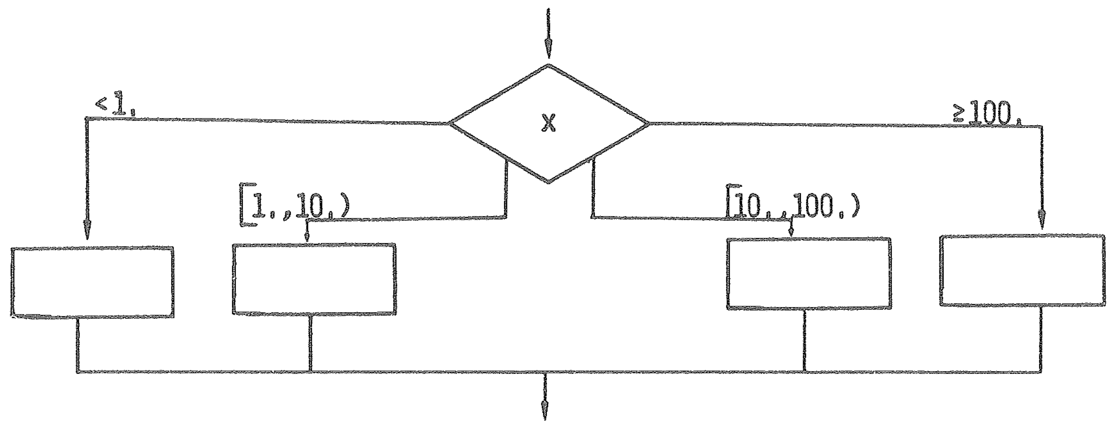
FORTRAN 77

```
IF ( J.LE. 3) THEN

   [100 LINES OF CODE]


ELSE


   [100 LINES OF CODE]


END IF
```



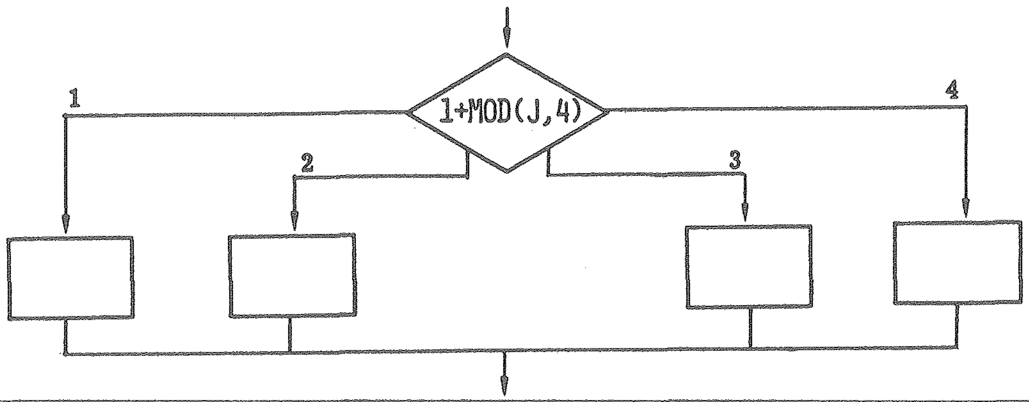
SFTRAN3

```
IF ( J.LE.3)  THEN

   DO(CASE OF SMALL J)

ELSE

   DO(CASE OF LARGE J)

END IF


PROCEDURE (CASE OF SMALL J)
    [100 LINES OF CODE]
END PROC

PROCEDURE (CASE OF LARGE J)
    [100 LINES OF CODE]
END PROC
```



<u>SFTRAN3</u>

```
IF( X  .LT. 1. ) THEN
   DO ( CASE  X < 1  )
ELSE IF ( X  .LT. 10.) THEN
   DO ( CASE  1 ≤ X  < 10 )
ELSE IF (  X  .LT.  100. ) THEN
   DO ( CASE  10 ≤  X  < 100 )
ELSE
   DO ( CASE  100 ≤ X )
END IF

PROCEDURE ( CASE X < 1 )
   .
   .
   .
END PROC
```



SFTRAN3

```
DO CASE (1 + MOD( J, 4 ),   4 )

CASE 1
  .
  .
  .
CASE 2
  .
  .
  .
CASE 3
  .
  .
  .
CASE 4
  .
  .
  .
CASE OTHER
  .
  .
  .
END CASE
```

INDEXED LOOP

FORTRAN 66 AND FORTRAN 77

```
    DO 10 I = N1, N2
        .
        .
        .
 10 CONTINUE
```

SFTRAN3

```
    DO FOR I = N1, N2
        .
        .
        .
    END FOR
```

	F66	F77	SF3
1. EXPRESSIONS FOR N1, N2, AND N3.		X	X
2. PERMIT $N3 < 0$.		X	X
3. POSSIBLE TO EXECUTE LOOP ZERO TIMES.		X	X
4. EXECUTE REMOTE CODE SEQUENCE.	X		X
5. NON-INTEGER LOOP INDEX		X	

OTHER LOOPS

```
LINK = LOC1
DO WHILE (LINK .NE. 0)
   A = DATA (LINK)
   .
   .
   .
   LINK = LIST(LINK)
END WHILE
```

```
LOGICAL QUIT
DO FOREVER
   READ (5,1000,QUIT = END) A,B,C
   IF (QUIT) EXIT
   .
   .
   .
END FOREVER
```

```
DO UNTIL (DIFF  .LE. EPS)
   .
   .
   .
   DIFF = ...
END UNTIL
```

```
DO BLOCK

    DO FOR I = 1, N

        IF ( J.EQ. KEY (I) ) THEN
            DO ( FOUND J AT INDEX I )

            EXIT BLOCK

        END IF

    END FOR

    DO ( J NOT IN TABLE )

END BLOCK
```

Evolution of the SFTRAN Language and Preprocessors

1973 SFTRAN1 John Flynn, JPL

1976 SFTRAN2 John Flynn, JPL

1977 LRC SFTRAN Ford & Fessler,
NASA Lewis Research Center
Cleveland

December 1978 SFTRAN3 JPL Univac 1108

June 1979 IBM 3032 & DEC 20

THE SFTRAN3 PREPROCESSOR

Portable baseline version

5	(IN)	Input source code
20	(IN)	Include modules
6	(IN)	Signon & Signoff messages
12	(OUT)	Indented SFTRAN3 listing
14	(OUT)	Generated Fortran code
15	(OUT)	Error messages, if any

Installation Requrements

- Machine specific subroutines CHGET3 & CHPUT3 to unpack & pack a single character.
- Change 3 FORMAT statements
- Change some DATA statements in a BLOCK DATA subprogram
- Need "END=label" in Fortran compiler

Programming Aids: Philosophy and Tools

N. L. Schryer

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

Most programmers are continually confused. The programming language they are using won't let them easily say what needs saying. Once written, the program won't compile. When it finally gets into execution, it aborts in some hidden place and doesn't say why. Then, when execution ends normally, the results are dreadfully wrong. Finally, after much effort, the program correctly runs for its creator. The programmer then ships the program off to a colleague across the ocean who can also use it. The colleague then reports that the program won't compile on the local machine. After many very-long-distance phone calls, it runs correctly abroad. The colleague then complains that the program runs far too slowly.

The above scenario is probably all too familiar to most programmers. It is so familiar, that people don't even realize the confusion it indicates. So much of life is lived under such circumstances, that people simply accept it into their programming lives.

Most people believe that their behavior, in many areas, falls barely short of perfection. They aspire to grand goals, and, lacking a guardian angel to tell them they have failed, assume they have succeeded. Programming is different, however. The guardian angel (named IBM, UNIVAC, etc.) never fails to remind us of our failings as programmers. Indeed, at times the guardian angel seems to rather enjoy this role of informer.

The rules by which programming perfection is judged are so complex, diverse and massive that no single human being can remember, let alone digest, them all. Luckily, the machine can remember such rules. The machine can also **enforce** many such rules. Those rules which can be enforced mechanically are rules we can tell whether we have broken. For example, rather than thinking we have a linguistically portable program, we can **know** it is portable.

This paper is about those rules of programming which can be mechanically verified by existing software tools. Such rules and tools form a very successful methodology for programming.

2. Rules and Tools

In this section, some rules of the programming game are given and the tools which support them are described.

Portability

This is a fundamental rule which guarantees maximal impact at minimal cost by making a program developed on one machine available to "all" machines. This exceedingly lofty goal is made concrete by using FORTRAN as the programming language. This rather unclean, ancient language is the only existing language with a subset which is supported by nearly all scientific computing centers. Other, prettier languages can only aspire to what FORTRAN has accomplished in portability.

The PFORT [3] subset of 1966 ANSI FORTRAN is portable and the PFORT Verifier [3] is a program which checks a given program unit for adherence to the PFORT subset. A program which successfully passes through the PFORT Verifier will compile on virtually any machine with a production FORTRAN compiler. That doesn't necessarily mean the program will run correctly, only that it will compile without a hitch.

The PFORT Verifier and the DAVE package [1] both perform checks which help ensure, but do not guarantee, that the program will also run correctly on all machines. For example, inter-program communication via calls and COMMON are checked for consistency and correctness, undefined variables are flagged, etc.

A program unit which has been passed through the PFORT Verifier and DAVE is both portable and syntactically correct. The program may not do what you want it to, but what it does do, it does correctly and consistently.

Good Programming Languages

Many programmers want to tear their hair and put on sack-cloth and ashes when faced with programming in FORTRAN. This is indeed the penance required for obtaining portability, a very worthy goal. However, there is help available. The programmer need not write the FORTRAN personally, rather, a program will write it. This may be accomplished by using one of the very large number of FORTRAN pre-processor languages available. Most of these have an Algol-Pascal flavor with *if ... else, while* and other modern control structures. Writing in such languages is much simpler and less prone to error than writing in FORTRAN. The pre-processor turns your nice modern program into ancient (yet vibrantly alive) FORTRAN. Only two FORTRAN pre-processors will be mentioned here, and these only because the author has seen them used by large populations successfully. They are Ratfor [4] and Efl [2]. If they are not immediately available in your shop, ask around, some useful pre-processor is nearby. Use it, you and your career will benefit.

There is a devilish and useful program called STRUCT [6] which does the inverse of what a pre-processor does: it converts FORTRAN into Ratfor. This makes the logical structure of the software apparent. It also often makes errors in logic obvious. STRUCT is invaluable when maintaining old FORTRAN programs. Once passed through STRUCT, they are written in a modern language and much easier to understand and modify. STRUCT-like programs for other pre-processor languages are in the works. Ask around in your shop about them.

Easier Debugging

All too often, when a program dies, its last words are something like

```
ERROR IHC123 at location 123ABC.
```

Even with several dictionaries and/or interpreters, such a message is usually useless. It typically means that the machine has encountered an addressing error in subprogram WOOPS. Why and how WOOPS got into such a predicament is left woefully unsaid. The standard programmer response is to put some print statements into WOOPS and the subprogram that called it.

Several computer manufacturers now offer a debugging facility, symbolic dumping, which makes this response unnecessary. Such symbolic dumping facilities give the chain of calls, from the main program, that led to the trouble. Also, for each program in the chain, all variables are listed by name and symbolic value (real, integer, ...). This information is usually sufficient to diagnose and repair the problem. Thus, one run with a symbolic dump serves the purpose of many runs without.

Honeywell provides this service and the Lawrence Radiation Lab at Livermore is working on providing it on CDC and Cray hardware. If your manufacturer doesn't support symbolic dumps, scream loudly until it does.

Testing

Complete testing of a program is beyond our present capabilities. However, we **can** tell if all paths through a program have been exercised by a data set. Since the vast majority of bugs involve blocks of code and/or combinations of If's not previously exercised, such information is invaluable.

Leon Osterweil and Lloyd Fosdick at the University of Colorado are working on profiler and instrumentation packages. These flag program paths not exercised during execution, report execution frequency of statements, allow testing of the internal status of a program (assertions), etc. While not yet fully complete, elements of these packages are available now. Another portable profiling mechanism has been described by Sande in [5].

Efficiency

A great buzz-word these days is efficiency. Use of the phrase

"I am making the program more efficient."

should notify the programmer's colleagues and supervision that the programmer is going to sleep at the wheel, since the programmer never seems to notice. Nowhere in life is the maxim "If it ain't broke, don't fix it" more applicable than programming.

The purpose of making a program more efficient is to get the **correct** answer **faster**. In practice, it usually means getting the correct answer not much faster using unspeakable coding practices. Even worse, sometimes it results in getting the **wrong** answer faster.

There are tools - timers - that can tell whether a program is efficient, and if it is not, where the inefficiencies are. Many manufacturers have timing mechanisms which give both the local and global (in and below) time spent within each subprogram, and the number of times each subprogram was called. The profiler [5] can also time programs, and it is portable. If no subprogram uses more than 30% or so of the time, then there is usually no need to change any one subprogram. Use a timer to find out where the run-time is being spent, and optimize (tinker with) only those subprograms.

Timers can also be used to check the computational complexity of the implemented algorithm. If the algorithm is, say, Gaussian elimination for dense, general n by n matrices, then it should run in $O(n^3)$ time. Running and timing the program with $n = 1, 2, 4, 8, 16, \cdots$ will confirm whether it behaves as it should.

3. Conclusion: Let the Tools do it

The following programming paradigm lets the tools do their job for you. It has been used very successfully by a number of mathematical programming groups.

- Write the program in RATFOR, EFL, ...
- Put it through the PFORT verifier and/or DAVE to check inter-program communications.
- Debug using symbolic dumps.
- Test using Profilers and Instrumenters.
- Using Timers and Profilers, find out where the run-time is being spent, and optimize (tinker with) only those subprograms.

Note that each of the above steps produces a higher quality program and documents it. This documentation may be referred to later (or concurrently) by the programmer, colleagues, supervision and successors.

Bibliography

[1] Leon Osterweil and Lloyd Fosdick, "DAVE - A Validation Error Detection and Documentation System for Fortran Programs", *Software-Practice and Experience* 6, 473-486(1976).

[2] S.I. Feldman, "The Programming Language EFL", Bell Laboratories Computing Science Technical Report #78, 1979.

[3] B.G. Ryder, "The PFORT Verifier", *Software-Practice and Experience* 4, 359-377(1974).

[4] B.W. Kernighan, "RATFOR - A Preprocessor for a Rational Fortran", *Software-Practice and Experience* 5, 395-406(1975).

[5] Gordon Sande, "Program Execution Profiles", p325, **Computer Science and Statistics 8th Annual Symposium on the Interface,** Health Sciences Computer Facility, UCLA, Los Angeles.

[6] Brenda S. Baker, "An Algorithm for Structuring Flowgraphs", *JACM* 24, 98-120(1977).

COMPUTER GRAPHICS SOFTWARE

James E. George

Mesa Graphics

Los Alamos, NM
Integrated Software Systems Corporation
San Diego, CA

Today's renewed popularity of computer graphics is largely due to decreasing costs and increasing capabilities in hardware and software, as well as a growing awareness of the utility of computer graphics to many applications. This is evident in the expanding computer graphics industry and the activities of ACM's Special Interest Group on Computer Graphics (SIGGRAPH).

A pocket on the cover of these Proceedings contains a computer-generated color microfiche representing what is possible TODAY, not TOMORROW. All original material was produced on standard hardware with a variety of graphics software. Many of these software packages (DISSPLA, GINO-F, etc.) are compared in a recent SIGGRAPH Newsletter (Computer Graphics 12, Nos. 1 and 2, 1978).

Generally, modern graphics software is portable (i.e. can be moved to various machines and is available on a wide range of machines) and device independent (i.e. can drive any graphics device --- even the one you have). Additionally, these packages are usually distributed as a subroutine library and thus the user must be able to program to utilitize them; DISSPLA and GINO-F are good examples.

Newer products, such as TELLAGRAF and MAPPER, provide the naive computer user with a graphics capability for examining his data or creating professional quality illustrations. In the long run, these user-oriented packages will be heavily utilized for business applications and technical illustrations.

called programs, which perform well-defined tasks. In computational quantum chemistry it is generally agreed to distinguish between, for example, integral evaluation programs, Hartree-Fock programs, integral transformation programs, molecular property programs, etc. In most cases there exists more than one version for each of these programs. Some of the versions differ in the algorithm chosen to solve the mathematical problem. Other versions may differ in the computer code implementation chosen to optimize for different physical problems. These different versions of a program may be best suited for different applications. Each of these programs processes and generates lists of data. In some cases these data lists may become extremely large and data handling may become a severe problem that needs special attention. In the past, most of the program systems have been developed in different groups, independently and in parallel. They have been designed for a variety of different computer hardware and system software and have very often been optimized by making extensive use of certain system- dependent features. In all cases, different structures have been used to pass information and data between programs.

The enormous progress in computer technology, in particular in recent years, has led to the development of increasingly larger and faster computer systems, as well as of powerful "mini" computers, both with a strongly decreased cost-to-performance ratio. Simultaneously the research in computer science has explored new ways to use these large and powerful computer resources more intelligently and more economically. Both developments have had very strong feedback on computational molecular physics. The increase in raw computer speed has made it possible to study molecular systems in better approximations and with higher accuracy, and to study more complex molecular systems. The availability of large, directly-addressable main storage and of randomly-accessible, large-capacity external storage has allowed the implementation of well-known algorithms, which were not feasible before, and has led to the development of new physical models and to solution of the related mathematical problems. As a consequence, physical models, mathematical algorithms, and computer implementations have become outdated faster and faster. Most groups have not been able to keep up with this fast development in computational quantum chemistry. It is therefore impossible to have incorporated in an individual program system all the major new developments over a longer period. The availability of reasonably-priced and powerful "mini" computers has created wide interest among researchers who are not experts in the field of quantum chemical calculations in order to supplement their theoretical or experimental work. To serve these demands, a variety of

EXTERNAL STANDARD DATA STRUCTURES FOR COMPUTATION IN CHEMISTRY

G.H.F. Diercksen and W.P.Kraemer
Max Planck Institut fur Physik und Astrophysik
Fohringer Ring 6
D8000 Munchen

West Germany

Abstract

External Standard Data Structures and I/O interface service functions will be described as a programming tool to pass data between different programs. The objective for the design of external standard data structures has been to guarantee a maximum mutual independence of the individual programs. The objective for the design of the I/O interface has been to guarantee a maximum independence of the user programs from the system I/O functions. The external data structures and I/O service functions described have been in use for many years in the MUNICH Molecular Program System. They have been found flexible, open-ended, and easy and convenient to use and implement in a higher-level programming language.

1. Introduction

Over the past decade, a number of large program systems for computation in chemistry have been developed. Each of these program systems involves the continuous efforts of a number of scientists and needs many, many years of work for development. It therefore constitutes a large investment for the contributing authors and for the sponsoring agencies.

When computers became generally available about twenty years ago and work in the field actually started, no or only little experience was available about the problems faced in computational molecular physics and in computer science. The development of large program systems in molecular physics has led to a set of generally accepted standard procedures and computer implementations for solving the mathematical problems inherent in the physical models to be treated.

Most of these large program systems can be separated into logically-independent modules, traditionally programs are necessary that can be run on the different "mini" computer systems and can be freely combined to serve the different purposes.

In view of these developments, and due to the high investments in computer programs, it has become strongly advisable to tackle the problem of program portability and linking. Flexible program linking can be easily achieved by defining standard data structures for communication and data transfer between different programs. Because of the large amount of data to be passed between some programs, these have to be stored on external standard data lists resident on some large capacity storage device other than directly addressable main storage. External standard data lists guarantee the independence of different programs within a program system. They allow each individual scientist to combine different programs according to his own needs, i.e. according to the problem to be studied and to the computer resources available.

2. Requirements

External standard data lists are a basic design feature of the MUNICH Molecular Program (MMP) System. They were implemented first in 1970 and have been used with great success since then. The external standard data lists have undergone a number of major modifications since being introduced, in particular during the early program design phase. The following discussion is based on our present experience.

External standard data lists must be convenient and easy to use in order to be widely accepted as a programming tool. In particular, all explicit data handling, like writing, reading, searching, positioning, checking, copying, and compressing must be performed by utility routines. This has the advantage that all device- or system-dependent I/O coding is restricted to the set of utility programs, which greatly increases the portability of the actual programs. [See Figure 1].

We start the discussion of the detailed structure of the external standard data lists by specifying the following requirements:

R1..External standard data lists have to contain all data and control information uniquely defining all data stored and the path and method(s) by which these data have been generated; that is, external standard data lists have to be self-defining.

R2..External standard data lists have to be identifiable INDEPENDENT of their actual position on the external device (data set, file, etc.).

R3..External standard data lists have to be open-ended.

R4..External standard data lists have to be structured into multi-level logical units of data. The logical units of data on the first level must be identifiable independent of their actual position, those on the other levels must be identifiable by position; that is, external standard data lists have to be modular.

R5..External standard data lists have to allow data compression without explicit knowledge of the content.

R6..External standard data lists have to be copyable without knowledge of the content.

For convenience and easy reference, external standard data lists which fulfill the above requirements will be called Standard Data Interfaces (SDI's) and logical units of data identifiable independent of position will be called Standard Data Sections (SDS's).

Some additional remarks are necessary and useful to clarify the formal requirements specified. As a typical example, we consider the Standard Data Interface containing the results of a Hartree-Fock calculation. Primarily one thinks of the results of an analytical Hartree-Fock calculation as the eigenvalues and eigenvectors resulting from the diagonalization of the Hartree-Fock matrix. But it is perfectly clear that these eigenvalues and eigenvectors are only defined with respect to the Hamiltonian operator the Hartree-Fock matrix has been calculated for, and within the orbital space the wavefunction has been expanded in. Therefore all data defining the Hamiltonian operator and the orbital space must be stored together with the eigenvalues and eigenvectors of the Hartree-Fock calculation to form a (self-defining) Standard Data Interface. All of this information will be necessary if the wavefunction is to be analyzed or to be used in some further calculation.

On a Standard Data Interface containing the results of a Hartree-Fock calculation, various lists of data are stored which form logical units of data. Typical examples of these are nuclear, orbital, and electronic configuration parameters, eigenvalues and eigenvectors. It is necessary and convenient to be able to access the logically independent units of data separately and independently of their actual position on the Standard Data Interface. Each of these logical units of data is therefore stored as a separate Standard Data Section. Some of these Standard Data Sections contain data referring to more than one program variable. For example, the list of nuclear parameters will contain the nuclear coordinates and the nuclear charges. In the present example, both lists are stored in the same Standard Data Section as positional data lists.

This raises the question if individual Standard Data Sections, or trees of Standard Data Sections are to be preferred over positional data lists for storing the values of individual, but logically strongly-related variables. In general, experience has shown that the creation of too many Standard Data Sections may cause inconveniences in the data handling and that storing the values of different, logically-connected program variables as positional data lists within the same Standard Data Section is preferable and cannot be completely avoided. In particular, trees of Standard Data Sections increase the inconvenience in data searching and positioning without offering any advantages over the single-level Standard Data Interface for the purpose of passing information between different programs.

Standard Data Interfaces may contain extremely long data lists, in particular symbolic and numerical integral and matrix element lists. In such cases it may be necessary to compress the data lists in a convenient way to minimize the data transfer and the external storage request. Very elaborate data compression techniques have been developed for different purposes and are used at present in most large program systems. To restrict the data communication between programs to the level of protocols and I/O utility programs to insure optimum mutual program independence, these packing procedures have to be incorporated into the I/O utility programs and have to be reduced for practical reasons to a minimum number of versions.

3. Structure

The specified requirements for the structure of Standard Data Interfaces can be easily met by sequential data lists and the extensive use of labels. Direct-access data structures (not direct-access devices) have never been seriously considered for Standard Data Interfaces, because direct-access data structures are very highly device-dependent. Moreover, direct-access data structures offer no advantages over sequential data structures for Standard Data Interfaces that warrant this consideration in this context. (To avoid any misunderstanding, it will be recalled that the primary purpose of Standard Data Interfaces is the data transfer between programs.)

Standard Data Interfaces are completely determined by the definition of their logical structure, their labels, and their data contents. For any actual implementation, these definitions have to be rigorously described in appropriate protocols. Because this paper is restricted to a discussion of the basic structure of Standard Data Interfaces rather than to a discussion of the technical details of the implementation, formulation and writing down of rigorous protocols has been avoided. We start with the definition of the logical structure of Standard Data Interfaces.

D1..A Standard Data Record (SDR) is defined as one logical string of alphanumeric data preceded by and including an SDR label (SDRL).

D2..A Standard Data Section (SDS) is defined as a sequence of any number of Standard Data Records preceded by and including an SDS Label (SDSL). The Standard Data Section Label itself is an SDR. An SDS is closed indirectly by the start of the next SDS, or by an end-of-file mark.

D3..A Standard Data Interface (SDI) is defined as a sequence of any number of Standard Data Sections, preceded by and including a SDI Label (SDIL). An SDI is closed indirectly by the start of the next SDI, or by an end-of-file mark.

We continue by describing the structure of the various labels and by listing the information that has to become part of the labels. All information that is optional for the correct functioning of the Standard Data Interface is marked accordingly. For all information finally specified in the appropriate label protocols, values must be supplied to guarantee the correct functioning of the Standard Data Interface features.

L1..A Standard Data Record Label (SDRL) consists of a fixed number of integer variables. The SDRL should include the following information:

SDR Flag
SDR #
SDR Type
SDR Data Length
SDR Length

L2..A Standard Data Section Label (SDSL) consists of one SDR identified by a unique and reserved SDR Flag. The SDS Label should include the following information:

SDS Name
SDS Extent #
SDS Date and Time (optional)
SDS Data ID

L3..A Standard Data Interface Label (SDIL) consists of one SDS identified by a unique and reserved SDS Name. The SDS Label may include any number of SDR's called label blocks. All of the label blocks have to be of identical structure and should include the following information:

SDI Name	
Installation ID	
Author ID	
Computer ID	
Program ID	(optional)
Program Release #	(optional)
Program Update #	(optional)
Program Generation Date	(optional)
Date and Time	(optional)
Data ID	
Maximum External Record Size	(optional)

The meaning of most of the label information is self-explanatory. Obviously, the meaning and range of the variables, and the reserved values have to be rigorously defined in the appropriate protocols. We concentrate here on the mandatory label information; that is most important for the correct functioning and the sensible use of available facilities of Standard Data Interfaces.

The SDR Flag variable is a key feature of the Standard Data Interface structure. The SDR flag is to be used to distinguish between SDR's containing label information and those containing actual problem data. The SDR flag is used in addition to mark the end of Standard Data Sections and the position of data lists in it. By use of the SDR flag, tree structures of Standard Data Sections may be constructed if this should be considered necessary at some time. By the sensible use of non-reserved values for the SDR flag, the user can build positional sublists of practically any complexity and any level of depth. In actual programming this flexibility has been found to be extremely useful in building logical data structures for defining checkpoints in Standard Data Sections. Because the SDR flag is used for marking the end of all data lists, it has not been found necessary nor useful to introduce trailer labels for Standard Data Sections and Interfaces.

The SDR Type variable has been introduced to allow a flexible definition of different SDR types, according to the type of the stored data. This flexibility is of particular importance for the definition of SDR types containing data stored in non-standard formats which have to be converted to standard data formats (encoded/decoded) according to special algorithms (e.g. data compression).

The SDR Data Length specifies the data length in units of the data type. The Standard Data Record length is calculated by the I/O utility program from the data length and type in some suitable units common to all SDR's.

In actual application programming, it has been found most useful to have a set of SDR Label positions reserved for the user. Such label positions are in particular convenient for storing additional checkpoint and restart information that is only of meaning within the generating program. Of course it may be questioned if any control of this kind over label positions should be given to the user. It is advocated here that all control over the label that does not destroy the integrity of the Standard Data Interface should be left to the user.

A SDS Extent # variable has been introduced to link and to distinguish between SDS's with identical names. It has been found necessary to introduce SDS extensions to be able to store data on a Standard Data Interface after the associated SDS has been closed by another SDS. For example, such situations may arise, if expectation values have to be calculated from the same set of wavefunctions and stored on the same Standard Data Interface for parameters that have previously not been specified.

The SDS Name is used to identify the individual SDS's and therefore has to be unique. It is directly used by the application program to refer to the individual SDS's.

The SDI Label may contain any number of label blocks (SDILBK). The first label block in the list is called the active label block and its information refers to the present SDI. The other label blocks in the list are called concatenated label blocks. The information in each of the concatenated label blocks refers to one of the SDI's used in constructing the present SDI. This SDI Label structure allows a unique description of the history of all data stored on the present SDI. The SDI Name of the active label block is used to identify the individual SDI's and therefore has to be unique.

4. I/O Interface

It has been pointed out previously that all SDI handling has to be performed by calls to an appropriate I/O interface. This restricts all actual I/O statements which are generally strongly dependent on the computing environment to the I/O interface and thus facilitates the maintenance and the adoption of the SDI handling to different computing environments. This guarantees a maximum mutual independence and portability of programs.

The following interface functions have been found necessary and sufficient for all purposes of SDI handling. All administration of the lower level SDS positional data lists has been left completely to the individual programs.

The I/O interface service functions may be subdivided into three groups, according to the "structures" they act on, namely SDI, SDS, and SDR functions. These I/O interface service functions are summarized in the following list.

Generate an SDI label block	(GENSDI)
Locate an SDI	(LOCSDI)
Read next SDI label block	(RDXSDI)
Copy an SDI label block	(COPSDI)
Generate an SDS label	(GENSDS)
Locate an SDS	(LOCSDS)
Skip to next SDS	(NXTSDS)
Copy an SDS	(COPSDS)
Write an SDR	(WRTSDR)
Read an SDR	(RDSDR)
Copy an SDR	(COPSDR)

These I/O Interface service functions and their arguments have been compiled in the following table. The mnemonics are expected to be self-explanatory from the context.

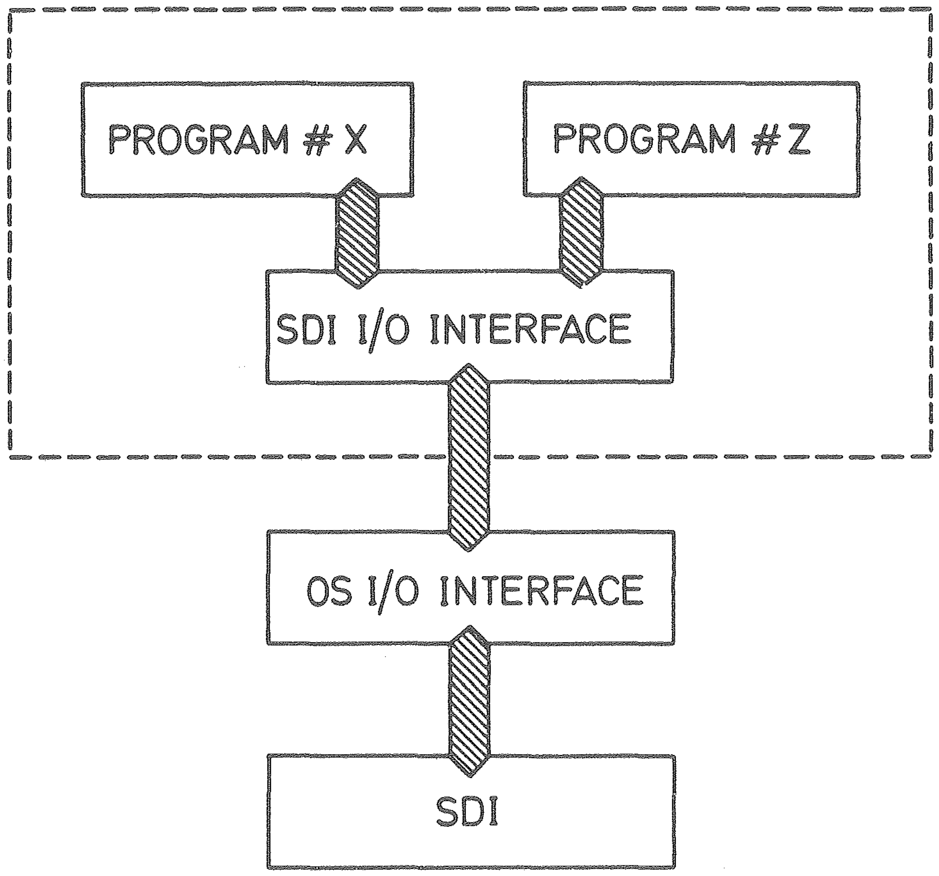
Function	Argument List
GENSDI	FILENO, SDINAM, SDILBK,...
LOCSDI	FILENO, SDINAM, SDILBK, RTCODE
RDLSDI	FILENO, SDINAM, SDILBK, RTCODE
COPSDI	FILEFR, FILETO, SDINAM, SDILBK, RTCODE
GENSDS	FILENO, SDSNAM, SDSEXT
LOCSDS	FILENO, SDSNAM, SDSEXT, RTCODE
NXTSDS	FILENO, SDSNAM, SDSEXT, RTCODE
COPSDS	FILEFR, FILETO, SDSNAM, SDSEXT, RTCODE, BUFFER, LBUFFER
WRTSDR	FILENO, SDRFLG, SDRNO, SDRTYP, ULABEL, DATA, LDATA
RDSDR	FILENO, SDRFLG, SDRNO, SDRTYP, ULABEL, DATA, LDATA
COPSDR	FILEFR, BUFFER, LBUFFER, FILETO, RTCODE

In addition a general utility function is needed that allows one to analyze and convert SDI's. This utility must include functions to list the contents of the SDI selectively in appropriate different formats and to convert SDI's from unformatted to formatted data structure and back, allowing for example data interchange between different computing installations.

It has been found necessary to include into the I/O interface three file handling functions, namely end-of-file, rewind file, and skip to end-of-file.

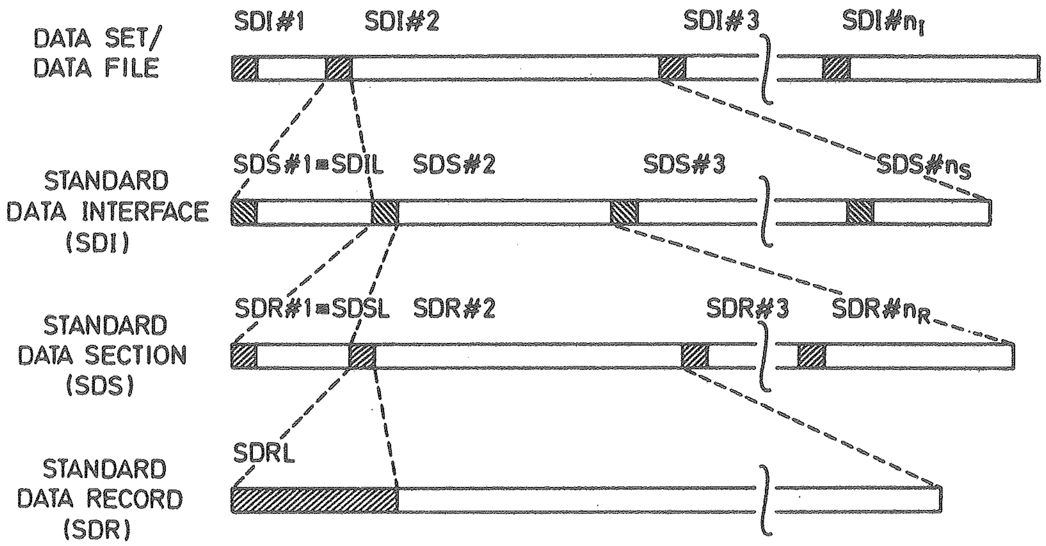
5. Summary

The logical structure and function of the Standard Data Interfaces and the service functions of the related I/O interface have been described. The Standard Data Interfaces and the I/O interface have been implemented and used in the MUNICH Molecular Program System for a number of years. The details have been defined in the necessary protocol and are part of the program documentation. They have been found easy to implement in a higher level language (FORTRAN), flexible and open-ended with respect to modifications, and easy and convenient to use. They guarantee complete independence between individual programs and increase the programs' portability. The Standard Data Interfaces described are structured and contain the necessary information to serve as a Data Information System. For easy and convenient data analysis, the Standard Data Interfaces should be converted to and handled by a suitable and commonly-available Data Base Management System.



SDI = STANDARD DATA INTERFACE
OS = OPERATING SYSTEM

FIG.1 DATA PATH BETWEEN PROGRAMS



SDIL = SDI LABEL, ETC.

FIG.2 STANDARD DATA INTERFACE STRUCTURE

PORTABLE SOFTWARE
WITH EMPHASIS ON CRYSTALLOGRAPHY

James M. Stewart

Department of Chemistry
Fellow of the Computer Science Center
University of Maryland,
College Park, MD 20742

I am riding on a limited express,

One of the crack trains of the nation.

Hurtling across the prairie into the blue
haze and dark air

Go fifteen all-steel coaches holding

a thousand people.

(All the coaches shall be scrap and rust
and all the men and women laughing
in the diners and sleepers shall pass to ashes.)

I ask a man in the smoker where he
is going, and he answers "Omaha".

"Limited" by Carl Sandberg

This quotation is an attempt to put into perspective the philosophical discussions of the morning sessions.

The subject I wish to address now, in addition to philosophy is the nitty-gritty of our experience with and our proposed methods of preparing transportable software. Mind you, not so much of the nitty-gritty that I get into an altercation of the kind that Diercksen ended in. He established that, if you reveal too much about what you've done in computing, people immediately are able to get to you. Even knowing that, I will not try to conceal all the details of what we have accomplished and that we hope to accomplish in the future.

The XRAY system is a set of programs that has been produced in a number of editions. The codes extend back to 1960. There was an XRAY63, XRAY70, XRAY73 and the latest is called XRAY76. These codes run on a variety of machines. They are written in what we call PIDGIN FORTRAN. PIDGIN FORTRAN used our experience in the frustration of moving from machine to machine to simulate PFORT. There were a number of us at the University of Washington in the early days who were involved first on the IBM 604, then the IBM 650 and finally, before we graduated and were dispersed, on the IBM 709. When all the effort had been applied it came to my attention that I didn't have access to a 709 anymore, but that we had written codes that were very specific to the 709. I discovered that probably, and it proved to be true, we would never get back in our lifetimes machine time savings corresponding to the time and care lavished on the machine language coding done on the IBM 709. We had created some really remarkable codes that had a half-life of a year or so that had taken about two years to get to work the first time.

Thus PIDGIN FORTRAN was evolved as we scrambled to recode our first efforts so they would run on our CDC, UNIVAC, HONEYWELL, IBM, ICL, DEC and other machines we found in our new surroundings.

Now with the announcement of FORTRAN 77 I have realized that we are on the threshold of another upheaval in coding. The changes which have been presented in the earlier papers today portend greater changes than those that occurred when we went from machine language to FORTRAN II. In fact, it is sufficiently different that we must decide how we are going to move to preserve our libraries of application programs in a manner that will give us continuity. The continuity that we gain should then conserve our programming effort and allow us to move forward to more demanding problems rather than being put back to square one every time a new generation of computers and computer software come to pass.

To accomplish this goal in crystallographic programming we have defined a new system, which we call XTAL. The coding for this system is being carried out in RATMAC which is a FORTRAN based preprocessor.

We agree very strongly with the previous presentations at this meetings that this is the way that we, as working scientists can preserve our software. We believe that this method of coding can give us a library of checked-out programs that will tend to remain checked-out.

Now choosing a preprocessor, I think, is a kind of social problem. It has all kinds of ramifications. Everyone who once has chosen or written his own preprocessor will naturally favor that one. The previous speaker pointed out there were 60 or more in existence. I didn't realize there were so many. We stumbled into RATFOR by Kernighan and Plauger because of the book Software Tools, and very quickly came to favor it.

Bob Munn, who works, as I do, at the University of Maryland got a copy of the Kernighan and Plauger book and brought it to my attention. (Those of you who believe in other preprocessors will enjoy punching out on him after this presentation.) I found Software Tools to be a lovely thing: it is really a fine preprocessor for the kinds of things we wish to accomplish in writing XTAL. The most important feature that it has is the MACRO function. Bob combined the macro function with the RATFOR preprocessor and made the whole a one pass preprocessor RATMAC.

Now I must confess that I can look at structured programming and I can take it or leave it. I do not believe, as I mentioned during a previous presentation, that structured programming helps the programmer. It may help improve the quality of programs: it may help make the FORTRAN optimizer look good, it may make it easier to debug or to correct. But it puts constraints on the programmer that force him to work harder than before to get it right the first time. As a matter of fact the first time Jim Holden and I tried to write a line counting routine we spent an inordinate amount of time because we couldn't believe that "you can't go back". It was just illogical after 20 years of FORTRAN programming. We finally realized that you sometimes have to make the same statement twice in a program and that that was an overhead of structured programming to be lived with.

As we knuckled under to the additional constraints, however, we found that many of our old codes became more compact as the structured discipline became clearer to us.

We discovered in our old codes things which were really trash, and when analyzed from the "top-down" point of view it puts the discipline on one to produce the result in a compact and flowing way.

What really turned us on, however, was the MACRO processor. This is a thing that you can live with if portability is your goal. This is because we can set up MACROS which contain all the kinds of statements we know cause difficulty in moving from machine to machine.

Either by running codes through PFORT or by the experience of moving codes from machine to machine we know that there is a subset of FORTRAN for every version which is not transportable to other machines. The use of MACROS for these "controversial" FORTRAN statements allows pushing them back one level. The problems of READ and WRITE, how many bits in a word, how many words in an I/O buffer length, can all be specified in MACROS. We then write our RATMAC programs to use the MACROS and adapt just the MACROS at each new installation. We have also created a set of subroutines which are structured on these MACROS and carry out the function of a sub-monitor of any operating system. We call these subroutines the NUCLEUS of the XTAL system. These are "primitives" which handle I/O, word packing, bit manipulation, etc, etc.

If the MACROS are properly tuned we have at once general transportable FORTRAN code and machine specific, operating system tailored code!!!

For example on the debate that occurred after Dr. Diercksen's presentation: you tune the MACROS to give 508 words per binary read-write if you're running on CDC, to 1024 on certain IBM systems, or whatever will be most efficient at a given shop. You can either specify in the MACROS that the writer will be FORTRAN READS or WRITES if you set the MACROS that way, or you can invoke an "executive request" on the given machine and avoid the FORTRAN library completely where that is advantageous. The MACROS give us that kind of flexibility.

We have, as an underlying principle in XTAL defined a crystallographic data base. The methodology is very similar to that presented here earlier. The data base must contain those physical quantities which crystallographers really need to use in the solution, refinement, and publication of single crystal structures. I believe the nature, contents and structure of data bases is the point about which there is the most hope for agreement. It should be possible to agree on the structure and contents of a data base. Furthermore if the work is done carefully, the data base should be open-ended enough that it can be expanded in the future to take care of oversights in the present.

The crystallographers met at the University of California at La Jolla earlier this year and tried to see what could be done about the point now raised in this conference. The greatest agreement was reached on what quantities are needed in a crystallographic data base and on a fairly universal way of creating it. We are now writing a set of programs in RATMAC for creating such a data base for crystallographic data processing. The working programs are written in RATMAC and draw on MACROS and the nucleus subroutines to read, 'massage' and write the data base.

When we wrote in PIDGIN FORTRAN, we did what I saw on a slide show in a previous presentation. We put in comments which said, in effect, "This is a machine specific FORTRAN statement" or "This is machine specific as all hell; use it at your own risk". Then we might supply in the comments some hints as to what you might do on your HITACHI or TELEFUNKEN and left it at that. This presents a real problem since every week or so we had to find these special comments and add new ones that said "360 level H compiler can't cope with this statement" or that sort of thing. We obviously wanted to get away from that by turning to RATMAC.

The next part of this talk is to say something about RATMAC since it is clear from what has gone before that not everyone is familiar with Kernighan and Plauger or Munn's work in setting up RATMAC for us to use in developing XTAL.

RATMAC comes from "RATional fortran with MACros". And it is the son/daughter of RATFOR and MACRO. The loop structures that are used in RATFOR are very similar to the ones described earlier today. Computer scientists are always searching for new forms. They have tenure considerations, too. You heard a chemist ask "why can't we just have FORTRAN IV and be done with it? What's all this FORTRAN 77 nonsense?"

This feeling tends to be very strong among those of us who do applications programming.

RATMAC nee RATFOR as a structured language uses brackets { } as its means of defining blocks of code; thus one says

```
FOR (I = 1; I.LE.100; I = I + 1)
{
<statement block>
}
```

The range of the FOR statement is defined by the open and close curly brackets. Similar to FORTRAN you write

```
DO J = 1, 100
{
<statements of DO loop>
}
```

We have prepared in addition to the excellent

Software Tools by Kernighan and Plauger a RATMAC primer TR-804 of the Computer Science Center of the University of Maryland.

What the preprocessor does is to change the statement blocks into the FORTRAN code by additionally including the appropriate GO TO's and CONTINUE's with their statement numbers which may be passed to the local FORTRAN compiler.

One particularly attractive feature of the preprocessor is the "free form". That is, unlike FORTRAN, you simply type anywhere on the line, and comments separated by a special character can be placed right on the line with the statement.

Another feature of RATMAC which is useful is the fact that it recognizes digraphs. These allow the extension of special characters to machines which do not allow certain special characters. The curly brackets required to define statement blocks do not exist on an 026 keypunch. I still work in the horse and buggy era with a keypunch. I don't have a fancy terminal like Munn has. His is even hard-wired to the 1108 concentrator! What he has provided for us dark agers is a digraph $(is seen as § while $) is seen as †. So I can use my keypunch and he can use his super terminal and RATMAC copes just fine.

The MACRO's are the useful software tool as far as portability is concerned. They allow the replacement of a simple string with a much different FORTRAN statement. A MACRO may, furthermore, have up to nine arguments.

```
MACRO:(MXBTWD:,60) # on CDC
MACRO:(MXBTWD:,32) # on IBM or VAX
MACRO:(MXBTWD:,36) # on UNIVAC
```

This allows simple substitution in the RATMAC code where the bits-per-word is important.

A more powerful example of MACRO use is in some function such as word packing. The RATMAC statement

```
INTPAK:(I,W,10,5)
```

means move the low-order 5 bits of integer I into bits 14, 13, 12, 11, 10 of real W. The MACRO: INTPAK: looks like this on CDC:

```
MACRO:(INTPAK:, [$2 = SHIFT(SHIFT(MASK($4),$4) $#
      .AND.$1,$3).OR.(.NOT.SHIFT(MASK($4),$3+$4).AND.$2)])#
```

while on UNIVAC it would look like:

```
MACRO:(INTPAK:,[KIK = 36-$3-$4; FLD(KIK,$4,$2) = $1])#
```

These examples are just intended to hint at the power of this preprocessor and to show how it gives one the power to do those things which at once generalize the code at the programming level, and then make it specific when it is passed to the local FORTRAN compiler.

There are a good number of built-in MACROS that are described in Software Tools that give one many excellent features to use in forming general transportable code. The following example taken from the XTAL system shows what a RATMAC program looks like in our interpretation of the language. I hope it's well enough commented so that one may get the flavor of the method. Remember that each string that ends with a colon (:) invokes a MACRO, none of which are shown here.

```
SYSTEMHEADER:(AA08)                            #
                                               # LINE OUTPUTTER AND PAGINATOR
SUBROUTINE  AA08(NSP,HEADER,LGH,KEY,PR)# NSP=SPACES BEFORE FIRST LINEOUT
SYSCOM:                                        # KEY=1 OP HEAD CONDIT./BUFF NOW
INTEGER K,NSP,LEN,KEY,LGH,PR                   # KEY=2 HEAD AND BUFFER NOW
CHARACTER: HEADER(1)                           # KEY=3 OP HEADER ONLY NOW
REAL FMT(2)                                    # PR = REQUESTED OUTPUT PRIORITY
DATA FMT/340313.,380313./                      # HALL, STEWART SEPT. 1978
                                               #
IF(PR.LE.OTPRMX)                               # IS PRIORITY ACCEPTABLE
$( #
LEN = MIN0(LGH,IOCHR:)                         # SET LINE LENGTH
                                               #
IF((KEY.EQ.1).AND.(LINRM.GT.0)) K=2            # SET NO. OF LINES TO BE OUTPUT
ELSE IF((KEY.EQ.2).AND.(LEN.GT.0)) K=2 # SET NO. OF LINES TO BE OUTPUT
ELSE K=1                                       # SET NO. OF LINES TO BE OUTPUT
                                               #
IF((LINCT-NSP-K.LT.0).OR.                      # TEST IF SUFFICIENT LINES LEFT
   (LINCT.LT.LINRM))                           # OR EXCEEDS PRESET LINEROOM CNT
$( #
LINCT = MXLNPG:                                # RESET LINE COUNT
NEXTPAGE:                                      # SKIP TO TOP OF NEXT PAGE
MPAGE     =MPAGE+1  ;NPAGE     =NPAGE+1        # INCR. CURRENT/TOTAL PAGE COUNTS
BFOTFP(1)=MPAGE     ;BFOTFP(2)=NPAGE           # STORE IN OUTPUT BUFFER
NCODEFLD:(BFOTFP,1,BFTITL,FMT,2)               # PUT PAGE COUNTS IN TITLE LINE
LINEOUT:(PR    ,BFTITL,IOCHR:)                 # PRINT THE PAGE HEADING
LINEFMT:                                       # FORMAT FOR LINEOUT:
LINEOUT:(PR    ,BLNKWD,1)                      # PRINT A BLANK LINE
LINCT = LINCT-2                                # DECREMENT LINE COUNTER
IF(LEN.GT.0)                                   # TEST IF HEADER TO BE PRINTED
$( #
LINCT = LINCT-1                                # DECREMENT LINE COUNT
LINEOUT:(PR,HEADER,LEN)                        # PRINT THE HEADER INFORMATION
$) #
IF(KEY.LE.2) LINCT = LINCT-1                   # ANTICIPATE BFOTLN LINE OUTPUT
$) #
                                               #
ELSE                                           # IF NOT A NEW PAGE
$( #
LINCT = LINCT-NSP-K                            # DECREMENT LINE COUNTER
FOR(K=1; K.LE.NSP; K=K+1)                      # GENERATE SPECIFIED BLANK LINES
LINEOUT:(PR,BLNKWD,1)                          # PRINT A BLANK LINE
IF((LINRM.GT.0).OR.(KEY.GE.2))                 # TEST IF HEADER TO BE FORCED
$( #
IF(LEN.GT.0) #
LINEOUT:(PR,HEADER,LEN)                        # PRINT THE HEADER INFORMATION
ELSE LINCT = LINCT + 1 #
$) #
IF(KEY.EQ.1) LINRM = 0 #                         RESET LINRM COUNTER
ELSE LINRM = MAX0(LINRM-K-NSP,0) #               RESET LINRM COUNTER
$) #
                                               #
IF(KEY.LE.2)                          # TEST IF LINE BUFFER OUTPUT
$( #
LINEOUT:(PR,BFOTLN,IOCHR:)            # PRINT THE OUTPUT CHAR BUFFER
MOVEBYTE:(BLNKWD,1,BFOTLN,1,MXCHLN:,1) # BLANK OUTPUT BUFFER
$) #                                  #
$) #                                  #
RETURN                                #
END                                   #
SYSTEMHEADER:(DR04)                   #
SUBROUTINE DR04(ARG,T1,T2,RESULT)     #
# SUBROUTINE DR04 MAKES A FOUR POINT INTERPOLATION OF SCATTERING FACTORS
# FROM TABLES. ARG IS SIN THETA OVER LAMBDA, RESULT IS F(J) BASED ON
# XRAY76 PROGRAM RDIN OF 5 OCT 1968.  THE TABLE T1 MUST EXTEND 3 ENTRIES
# BEYOND THE GREATEST VALUE OF ARGUMENT USED. (CAVEAT:  CALLING PROGRAM#
# MUST ASSURE THIS CONDITION)         #
# NO PRESUMPTION ON INTERVAL STEPS OF TABLES
#                                     #
# 22 JUNE 1978  J.M.STEWART AND R.DOHERTY
REAL ARG,RESULT,T1(50),T2(50),DIF(5),SMALL #
INTEGER K,I,J                         #
DATA SMALL/0.0001/                    #
K = 1                                 #
WHILE(ARG.GT.T1(K)) K=K+1             # SEARCH FORWARD UNTIL ARGUMENT
                                      # IS LOCATED IN SIN THETA OVER
                                      # LAMBDA TABLE.  AT THIS POINT
                                      # T1(K) IS VALUE GREATER THAN OR
                                      # EQUAL TO ARG, SO BACK OFF ON K
IF(ABS(ARG-T1(K))>SMALL)              # AVOID CLOSE ENCOUNTERS OF THE
$( #
                                      # FIRST KIND
K = K-1                               # THE ARGUMENT IS IN THE INTERVAL
                                      # T1(K) TO T1(K+1)
FOR(I=1; I<=3; I=I+1)                 #
$( #
J = K+I                               #
DIF(I)=(T2(K)*(T1(J)-ARG)-T2(J)*     $#
(T1(K)-ARG))/(T1(J)-T1(K))            #
$) #
DIF(4)=(DIF(1)*(T1(K+2)-ARG)-DIF(2)* $#
(T1(K+1)-ARG))/(T1(K+2)-T1(K+1))      #
DIF(5)=(DIF(1)*(T1(K+3)-ARG)-DIF(3)* $#
(T1(K+1)-ARG))/(T1(K+3)-T1(K+1))      #
RESULT=(DIF(4)*(T1(K+3)-ARG)-DIF(5)* $#
(T1(K+2)-ARG))/(T1(K+3)-T1(K+2))      #
$) #
ELSE RESULT = T2(K)                   #
RETURN                                #
END                                   #
```

In summary I wish to say that both RATMAC and XTAL are quite portable. We supply RATMAC and XTAL in a magnetic tape. It can be bootstrapped and then tuned to the local computer.

It's portable. It's optimizable. It's small. RATMAC is under 9K words. The XTAL system is shooting for 7K words plus data arrays which will be problem size dependent. (Speaking in an overlap sense, not a virtual memory sense!)

All in all, I'm enthusiastic about the method, and I hope others will see merit in it, too.

For those of you who are not convinced about these matter, I will leave you with the following refrain from Pope:

Vice is a creature of such horrid mien,
as needs be hated, needs be seen.
But seen too oft,
familiar with her face,
we first endure,
then pity,
then embrace!

REPORT OF WG1

PORTABILITY CONSIDERATIONS AND STANDARDS

There was little disagreement among the participants that FORTRAN is now, and will be for quite some time, the major language in which chemical software is written. The primary concern of Working Group 1 was the shifting standards of the FORTRAN language. Without a stable world-wide language base, portability becomes not simply difficult, it becomes impossible. Therefore, the group unanimously agreed that the NRCC should sponsor a member of the ANSI X3J3 FORTRAN Standards Committee. This person should be an effective lobbyist, willing to spend a significant amount of time protecting the interests of computational chemists. Among these interests is the strict preservation of upward compatibility in the language, including common current practices not included in the 1966 Standard. There is considerable concern that the 1977 Standard has violated this principle.

A subcommittee was set up to evaluate other areas of concern and their report is attached. In addition, two other reports were submitted and are included.

Concerning preparation of portable programs, two areas were addressed: tools to prepare and maintain portable code, and the use of standard libraries.

The group recommends that the NRCC support and distribute a macro preprocessor to handle machine dependencies. The facilities offered by preprocessor macros for the definition of machine-dependent constants should also be available via SUBROUTINE or FUNCTION calls. The selection of the actual preprocessor was left to the NRCC. [Editors' note: Working Group 6 has made a specific recommendation to the NRCC on this topic.]

The group also recommended that the NRCC acquire portable software maintenance tools, although there was considerable concern that such tools would be used to alter and distribute "unauthorized" versions of programs.

Finally, many computational chemists are unaware of many of the conversion and certification aids available to assist in producing portable code. To assist the general chemistry community in this area, it is recommended that the

NRCC prepare a short annotated list of software tools currently in use, and indicate where they may be obtained. Appropriate places to publish may be Chemical and Engineering News and Physics Today.

With respect to the use of standard libraries, it is recommended that the NRCC support and encourage the use of LINPACK, EISPACK, and the Basic Linear Algebra Subroutines (BLAS). The use of generic names that do not specify precision is also recommended. This means that S and D prefix letters in the BLAS would be changed to R (for Real).

The need for standard routines providing direct-access usage is recognized. A subcommittee to develop a series of standard calls was established, and their report will be included in the Conference Proceedings.

WG1 Chairman
Stephen Elbert

SUBCOMMITTEE REPORT 1

The purpose of standards is to enhance portability and unify common usage. The effect of the revised 1977 ANSI FORTRAN Standard (and future planned enhancements) is to continually change the nature of FORTRAN 66. Working chemists need a stable language in which changes are upward compatible. We would like the NRCC to support the continuing of FORTRAN 66, and support the position that future revisions be upward compatible from FORTRAN 66. Upward compatible enhancements that we would like added are:

1) End-of-file checking on READ statements

2) Uniform direct-access of files

3) Quotes to define Hollerith data, as well as nnH... character counting

4) ENCODE/DECODE in some form

In addition, we wish to reiterate that the Hollerith data type should be kept. The NRCC should contact the American National Standards Committee (X3J3) and make this view known. This simple language (FORTRAN 66) should be kept simple.

Subcommittee on FORTRAN of WG1
[report accepted by entire WG1]
Lawrence Andrews and Norman Schryer

POSITION PAPER ON PROGRAMMING LANGUAGES

We suggest that the NRCC, as a representative of the chemical computing community, monitor attempts to standardize or define programming languages. We have specific interests to be protected in this field, in common with some other VERY large users. For our purposes, a usable programming language MUST have the following standard features:

1) Full support of double precision, complex, and double precision complex data

2) Variable dimensions in procedure arguments

3) Separately compiled procedures

4) Direct-access I/O

5) Efficient processing of simple indexed loops

In addition to the above mandatory features, we would like to see such things as:

1) Structured code

2) Structured data

3) Packed data

4) Dynamic storage

It is also very important that we do not lose ground in this area. We strongly recommend that any changes to the FORTRAN standard be upward compatible. The proposed FORTRAN 77 standard is a major problem in this respect. NRCC could point out, both to the ANSI Standards Committee X3J3 and to other major users such as DOE and the National Laboratories, the millions of dollars and the man-years that will be required to convert existing major programs to FORTRAN 77.

Lynn TenEyck

POSITION PAPER ON FORTRAN

The universal availability of FORTRAN makes it likely that this language will remain the language of choice for chemical computing for a long time. While upward compatibility must be maintained as the language evolves, modern programming techniques require certain features not envisioned in the original definition of FORTRAN. A list and evaluation of some possible FORTRAN extensions follows.

ESSENTIAL

1) Upward compatibility with the 1966 FORTRAN ANSI Standard

2) Data structures and arrays of structures

3) Character string variable type and associated operators (move and compare, at least). Facilities for conversion between character strings and other types would also be helpful (e.g. ENCODE/DECODE).

4) Bit manipulation primitives

5) Dynamic allocation of central memory

DESIRABLE

1) Control structures

2) Declaration of precision in terms of digits of accuracy

3) Vector, matrix zero, move, multiply primitives

UNDESIRABLE

1) Multiple ENTRY

2) Extended range of a DO

George N. Reeke, Jr.

REPORT OF WG2

CODING AND DOCUMENTATION STANDARDS

The discussions of Working Group 2 led to the following conclusions and recommendations.

1. NRCC should acquire, use, promote, and evaluate the FORTRAN preprocessors RATFOR/RATMAC and SFTRAN3. All the participants in this meeting familiar with these systems strongly recommend them as valuable tools for the production of high-quality, portable scientific software.

2. Editors, document formatters, and word processors are also valuable but, as these are frequently provided with the local system, there is less need for NRCC to become involved in providing standard tools for the chemical community. (The group did not discuss other software maintenance tools such as update systems.)

3. As much of the documentation as possible should be machine readable. Use of the full ASCII character set, particularly lower case letters, is to be encouraged, although this may lead to portability problems, particularly where CDC machines are involved. It is strongly recommended that an external, hard-copy description of the character set used accompany distribution tapes. This might also be included in comments at the beginning of the listing.

4. Documentation continues to be inadequate, partly because professional recognition commensurate with the amount of effort required for good documentation is rare. "You can't put a program write-up on your vita."

   Formal, detailed documentation standards or guidelines would do little to alleviate the problem, and might actually aggravate it in some cases.

   However, good documentation should certainly be encouraged. We recommend that NRCC refuse to distribute programs that do not have at least minimal documentation including:

   (a) what does it do
   (b) description of input

(c) description of output
(d) an example
(e) restrictions and limitations of problem scope and size
(f) computer type and resources required

5. It is recommended that NRCC form a committee to specify primitives for bit and character manipulation. This committee might well find that satisfactory specifications have already been made by other professional groups, particularly the "Purdue Workshop" of the Process Control Society.

6. Use of the matrix software collections EISPACK and LINPACK and the related vector software "Basic Linear Algebra Subprograms" (BLAS) is recommended. These are described in more detail in the paper by Moler in these workshop proceedings. The source code for all three collections is available through the mathematical software distribution service of IMSL, International Mathematical and Statistical Libraries, in Houston. EISPACK and LINPACK are also available to institutions with Department of Energy connections through the National Energy Software Center at Argonne.

7. NRCC should publish a list of useful software available from other sources. An NRCC Newsletter article on the software tools available with the UNIX operating system would also be valuable.

8. NRCC should represent the interests of chemists to the FORTRAN standards committee.

9. The documentation, distribution, and maintenance of modified software was discussed at length, but no firm conclusions were reached. Several people felt that the computing chemistry community has been lax in reporting modifications of software to the original authors. Others felt strongly that the original author, or a properly delegated representative, should be the sole distributor of modifications and documentation, but not everyone agreed.

WG2 Chairman
Cleve Moler

REPORT OF WG3

HARDWARE GUIDELINES FOR PROGRAM PORTABILITY

The discussions of Working Group 3 concerned hardware requirements for quantum chemistry and crystallographic computations. In particular, an attempt was made to specify a practical "minimal program environment" for FORTRAN program portability. For example: how much main memory should a programmer assume he has to work with? How much disk space and what type of access method is needed? What level of precision is appropriate for his problem?

Answers to these questions will largely determine the practical portability of apparently portable programs (e.g., those that have passed a PFORT test).

Many syntactically-portable programs are not portable in practice because of their lavish and inflexible use of resources. Such practices as fixing dimensions of arrays at their maximum values rather than using dynamic dimensioning, ignoring the possibility of exponent underflow or overflow on machines having limited exponent ranges, and wasteful use of disk storage severely, and usually needlessly, limit the utility of many programs.

Ideally, programs should adjust to their environment, and there are techniques existing to facilitate this. Probably the most important is run-time dynamic storage allocation, which allows a program to adjust its memory requirements on the basis of information supplied at run time rather than compile time. Such programs tend to handle both small and large problems efficiently. Similarly, disk space requirements can be minimized by using various, usually problem-dependent, data compression methods.

Separate hardware recommendations are given below for quantum chemistry and crystallography because (1) the two disciplines have requirements disparate from others in the field, and (2) much small molecule work in crystallography is done on private 16-bit minicomputers while essentially all ab initio quantum calculations are carried out on 32-bit or larger machines (operating largely in a timesharing mode during prime time, and in a batch mode at other times).

The working group felt that its recommended "minimums" would not be unduly restrictive or cause inefficiency, PROVIDED one started with the constraints clearly in mind. It was recognized that for algorithms which cannot be efficiently implemented under these restrictions, a lesser degree of portability has to be accepted. In most cases, however, several alternative implementations are possible: the preferred one uses the least resources, and thus stands the best chance of being portable.

Due to a shortage of time, the committee did not consider the impact of special purpose hardware (such as array processors, vectorized functional units, and multiprocessor configurations) on program portability.

MINIMAL HARDWARE CONFIGURATIONS FOR PROGRAM PORTABILITY

1. DIRECTLY-ADDRESSABLE REAL MEMORY:

QC - Programs should run in less than 32K working precision words (256K bytes on byte-oriented machines). 24K would be even more desirable if the program is to be used in a timesharing environment and if reasonable turnaround is expected. When more memory is available programs should be able to use it effectively, either to increase the efficiency of the calculation, or to handle larger problems.

CR - 65K bytes (or 32K 16-bit words) for small molecule work (100 atoms or less). For large molecules, 256K bytes should be adequate.

2. MEMORY ORGANIZATION:

a) Virtual memory - for portability, virtual memory should equal real memory. This is in contrast to the usual rule of thumb "working set memory size equals real memory size" that applies for efficient program execution. To facilitate transporting to non-virtual environments, a segment or overlay loader approach to memory management is strongly recommended even if a segment loader is not available (e.g., the VAX 11/780). Alternatively, a job step form of memory management should be used.

b) Multilevel memory - a one-level memory organization is preferred, since only CDC uses the two-level, large core-small core (LCM-SCM) organization. Present CDC FORTRAN compilers require explicit allocation of variables to LCM which presents problems with respect to both portability and dynamic storage allocation. However, in many cases one can avoid explicit random-access references to LCM and use it almost as effectively as a zero-access time I/O device, which is an intrinsically more portable form of use.

3. FLOATING POINT WORD LENGTH:

QC - A 32-bit floating-point format is completely unacceptable for most ab initio computations. Double precision should be standard for real variables for all IBM-like hardware. The 48-bit floating point of Harris computers is adequate -- indeed, perhaps optimum -- while the 36-bit single precision of DEC-10/20, Univac, etc., is usable only with extreme care. Semiempirical and empirical models can usually be computed quite satisfactorily with 32-bit single precision floating-point arithmetic.
CR - 32-bit floating point is generally adequate.

4. FLOATING POINT EXPONENT RANGE:

QC - Algorithms should be designed for an exponent-of-ten range of approximately +/-37, i.e., an 8-bit exponent field, since this old obsolete IBM format is still in wide use. The newest machine of consequence to use this format is the VAX 11/780. Exponent overflow and underflow are seldom problems on CDC hardware, and only occasionally cause problems on IBM systems. They continue to be a problem on other short word length machines, however.
CR - A similar recommendation applies, but in practice exponent problems seldom arise.

5. ROUNDING VS. NO ROUNDING FOR FLOATING POINT OPERATIONS:

60-64 bit - Rounding preferred but not of much importance in practice due to the high precision being used.
32-36 bit - Rounding highly desirable but not generally available.

6. INTEGER WORD LENGTH:

QC - 32 bits is optimum (determined by IBM-like architectures). Integers greater than 32 bits should not be used on machines with longer word lengths (CDC, Harris, Univac, DEC-10/20). A 16-bit integer, though feasible, would be extremely wasteful of memory on long word length machines and is therefore also not recommended. Further, most FORTRAN compilers for the larger 16-bit minicomputers now support a long integer format (INTEGER*4). However, if the compiler supports a short integer which is adequate for some purposes, and if storage is at a premium, then it may be used. This may detract from the portability of the code.

CR - 16-bit signed is generally acceptable on smaller systems. A 32-bit integer is preferred for large molecule work.

7. AMOUNT OF DISK SPACE AVAILABLE AS LOCAL FILE SPACE FOR AN EXECUTING JOB:

QC - 20-30 Mbytes minimum should be available. For large scale CI calculations, one should assume that 50-100 Mbytes will be available. Programs should not assume any particular form of file organization and/or layout on disk for either sequential or random-access file types. FORTRAN sequential file I/O is highly portable, but direct access I/O is sufficiently non-standard that it still should be isolated to facilitate conversion. Disk storage requirements can generally be considerably reduced by using some form of data compression on large files (e.g., molecular integrals or formula tape). Ideally, all such compression should be localized in the access routines and be transparent to the user.

CR - 1-2 Mbytes without an on-line data base. May be floppy based. Up to several hundred Mbytes with a full data base.

8. MAGNETIC TAPE:

QC - Assume in general that tape files are first copied to disk before being used. That is, tapes should not be considered as being on-line to the program.

CR - On private systems one slow drive is apt to be available, but its on-line use as an extension of local storage is apt to be awkward and non-portable.

WG3 Chairman
Stanley Hagstrom

REPORT OF WG4

QUANTUM CHEMISTRY SOFTWARE BASE

The initial integrated software system for quantum chemistry developed by the NRCC may not be optimal, since in the beginning compromises must be made. This natural tendency is aggravated by the rapid development of new methods and codes in such areas as multiconfiguration self-consistent-field (MCSCF) theory, configuration interaction (CI) techniques, and the coupled cluster method. However, the expected benefits of such a system, both to the computational chemistry community as a research tool and to the staff of the NRCC as a prototype chemistry software system, far outweigh the negative impacts of the compromises.

The quantum chemistry software system currently in use falls into two separate, but indistinct, classes: (1) semi-empirical systems (CNDO, INDO, MINDO, Extended Huckel, etc.), which are based on various integral approximations and which assume a restricted form of the wavefunction, and (2) ab initio systems (MELD, ALCHEMY, GAUSS78, etc.) which compute all of the integrals and allow a general form for the wavefunction. Although the programs in these two systems often overlap, and duplication could be minimized if they were made 'plug compatible', initially it may be best to maintain the historical separation of the two systems. The reasons for this include not only manpower limitations, but also the limitations of the semi-empirical methods in using the more general forms of the electronic wavefunction available in the ab initio systems. Since the semi-empirical systems are usually 'complete' packages, at least as far as their range of validity is concerned, we will concentrate here on the specification of the ab initio system.

Modern ab initio quantum chemistry software systems generally consist of two subsystems of program modules:

(1) the MAINLINE programs, which are used to calculate the electronic wavefunction and which are usually executed in a linked sequence, and

(2) the AUXILIARY programs, which manipulate the electronic wavefunctions to calculate properties, to plot orbitals, etc. These programs are often run on a stand-alone basis.

Although we considered primarily the MAINLINE programs, we wish to stress that widespread application will be found only for a complete system consisting of both the MAINLINE and AUXILIARY subsystems, with proper communication links.

The MAINLINE Subsystem

The first aspect of the MAINLINE subsystem considered was the user input/output. This is the part of the system that users come into contact with daily, and clearly it should be designed to minimize the effort involved in the preparation and debugging of data decks.

The user input should be free-form to allow the data set to be conveniently constructed at a terminal. This can be accomplished in (at least) four ways, by

(1) using the LIST-DIRECTED READ option,

(2) using the NAMELIST option,

(3) developing an interactive program which queries the user for the parameters necessary to construct the data set, and

(4) special-purpose scanner build into the code.

It is felt that the last option, the development of an interactive data file constructor, offers the best means of freeing the users from the more tedious aspects of data set preparation. It is not intended, however, for the data set constructor to define any of the parameters of the calculation other than the normal defaults.

For the user output two different schemes were considered: (1) one file containing all of the output, and (2) two files, one containing a summary of the output and another containing all of the output. In the first scheme, each line of output is coded with a priority level, with the highest priorities being assigned to the most important output. The user may then use a text editor to obtain a listing at the terminal of only those parts of the output file of interest: for normal operation this may consist of only the highest priority output (equivalent to the summary output in the second scheme). However, in case of a program abort the user can, by specifying a lower priority, obtain additional output. In fact, the entire output file can be retrieved in this way. The entire output file is also printed on microfiche. This output should be in 72-column format.

In the second scheme, two output files are generated:

* a summary file, in 72-column format, which contains the dayfile (job log file) and a summary of the results of the calculation; and

* a full output file in 132-column format on microfiche, which contains all of the output of the calculation(s).

In this way the user can view the summary file at a terminal to determine the essential results of a calculation, while receiving the full results at a later date on microfiche. This method has the disadvantage that, in the case of a program abort, only the summary output file may be available for debugging purposes.

Let us now consider the programs involved in the MAINLINE subsystem, specifying the characteristics considered to be the most desirable in the various modules. Existing programs whose characteristics closely match those suggested, are listed at the end of the section.

INTEGRALS Programs

Function: To compute the one- and two-electron energy integrals over the basis functions.

Characteristics should include:

* Handling (s,p,d,f) basis functions, combining the components to obtain integrals over the usual d- and f- functions.

* Built-in standard basis sets (exponents and contraction coefficients).

* Allowance for general contraction of the basis functions.

* Geometry input in both cartesian and 'bond length-bond angle' coordinates.

* Capability of calculating integrals over effective core potentials.

There is no integral program which combines all of the above features; however, selections of all of these features are found in the HONDO, BIGGMOLI and the MELD programs.

SCF Programs

Function: To compute the optimum orbitals (and configuration coefficients) given a particular electronic configuration (or a set of such configurations).

There are three different types of self-consistent field (SCF) programs to be considered: Hartree-Fock (HF), generalized valence bond (GVB), and multiconfiguration self-consistent-field (MCSCF).

HARTREE-FOCK Programs

There are three general types of Hartree-Fock programs:

(1) those using the integrals in supermatrix format,

(2) those with loops driven by the integral labels, and

(3) those with loops driven by the integrals themselves.

Which program is optimal depends on the parameters (degree of symmetry, number of basis functions, etc.) of the calculation. Initially only one type of program [type(1) or (2)] should be chosen; later it may become important to add other types to the system.

Characteristics should include:

* Handling both closed and open shell cases, with a general spin-coupling in the latter case.

* Use of non-orthogonal orbitals for open shell singlets with orbitals of the same symmetry.

* Handling unrestricted Hartree-Fock calculations, with the calculation of <S**2>. This will undoubtedly be a separate program.

The Hartree-Fock programs in the MELD and ALCHEMY systems and the GVB(ONE,TWO) programs satisfy the above criteria (except for the use of non-orthogonal orbitals for open shell singlet states).

GENERALIZED VALENCE BOND Programs

Characteristics should include:

* Allowing both perfect-pairing and more general spin-couplings.

* Assembling all Hamiltonians in one pass on the integral file.

* Allowing multiple correlating functions for each electron pair.

The GVB(ONE,TWO) and SOGVB programs satisfy many of the above criteria. Unfortunately, they are not currently available from the authors.

MULTICONFIGURATION SELF-CONSISTENT-FIELD Programs

Two major techniques are currently being used to solve the MCSCF equations: a new technique based on the generalized Brillouin Theorem and the conventional Hamiltonian technique. The techniques based on the Brillouin theorem should be implemented first, as they appear to avoid many of the problems encountered in the Hamiltonian techniques. A program based on the Hamiltonian approach can be added later.

Characteristics should include:

* Allowing a general configuration list, with the capability of generating the configurations internally.

* Unique partitioning of the 'doubly occupied' orbitals into core and valence orbitals.

* Calculation of the density matrix.

The MCSCF program in ALCHEMY and the ALIS program essentially satisfy the above criteria.

INTEGRAL TRANSFORMATION Programs

Function: To transform the one- and two-electron integrals over basis functions to the corresponding integrals over molecular orbitals.

In its optimum form, the integral transformation program requires N**5 mathematical operations. Within this class two general subclasses can be identified:

(1) those with N**2 memory requirements, and

(2) those with N**3 memory requirements.

Again, the optimum algorithm to use in a particular application depends on the calculational parameters, such as the number of basis functions and truncation of the virtual orbital space. Eventually, both types of programs should be made available.

Characteristics should include:

* Allowing all doubly occupied orbitals in the CI calculation to be transformed away.

* Allowing for the calculation of only those types of integrals occurring in the first-order CI calculations.

The transformation programs written by Elbert and by Raffenetti appear to be satisfactory, although neither takes advantage of the simplications which occur when transforming the integrals for first-order CI calculations. Elbert is planning to remedy this.

CONFIGURATION INTERACTION Programs

Function: To construct and diagonalize the Hamiltonian matrix over a specified set of configurations.

Two general types of CI programs are currently in use. Type (a) constructs the Hamiltonian matrix from the configuration list, type (b) from a formula tape. Type (a) programs are more efficient for one-time calculations (constructing the formula tape can be both a time- and storage-consuming process); and should be implemented first. Type (b) programs are most efficient if a calculation is going to be repeated many times, say at different geometries.

Characteristics should include:

* Handling a general configuration list.
* Providing for A(k) and/or B(k) selection.
* Arranging the configuration list so that configurations can be deleted from the end of the list (useful for extrapolation).

For the diagonalization step, either in-core (GIVENS, etc.) or out-of-core (SHAVITT, DAVIDSON) methods can be used. The decision on which method to use can be made within the program.

The MELD and CALTECH CI programs are type (a) programs which satisfy most of the above criteria, while the CI program in ALCHEMY is of type (b).

In addition to the above general CI programs, there are specialized, and highly-efficient, CI programs such as the direct CI programs of Roos and Siegbahn, and the self-consistent electron pairs program of Meyer, Dykstra, and Schaefer. After a general CI program has been implemented, these programs should be added to the system as soon as possible.

PERTURBATION THEORY Programs

Function: to calculate the many-body perturbation theory energy correction to some finite order.

Typically, the programs in use are based on spin-unrestricted Hartree-Fock wavefunctions, and calculate the energy correction to second, third, or fourth order. The second- and third-order energy corrections can always be calculated fully, but in those cases of fourth order, it is sometimes necessary to leave out the terms requiring O(N**7) arithmetic operations, where N is the number of basis functions. Presently, the Battelle Columbus programs and Gaussian 78 implement these procedures. The Battelle programs go beyond fourth order, and could be made publically available. The Gaussian 78 package is organized so that only a partial transformation of the two-electron integrals need be done. Also, the program produces the third-order energy in the first iteration of a direct CI calculation.

COUPLED CLUSTER Programs

The coupled cluster method is based on a molecular formulation of techniques first used in nuclear physics. Calculations are usually restricted to double substitutions, and are referred to as CCD. The method employs a wavefunction that allows for all double substitutions, including those simultaneous substitutions that are omitted in a CI with double substitutions. As in the perturbation procedures, CCD calculations are carried out in the spin-unrestricted framework for open-shell molecules. This method is currently implemented in Gaussian 78 and in the Battelle Columbus program, where single excitations are also included. The technique is iterative in nature, and requires about the same amount of effort as a direct CI calculation. The final CCD energy contrasts with the CI energy by being size-consistent but not variational.

WG4 Chairmen
Thom Dunning and Stephen Binkley

REPORT OF WG5

QUANTUM CHEMISTRY DATA INTERFACE

The questions addressed by Working Group 5 were:

1. What are the modules in a quantum electronic structure system?
2. How must these modules be connected to achieve a general system?
3. Where does input to each module come from? How many output files should it have?
4. How can we decide on details of program coupling so that individual modules can be plugged together without rewriting the programs?
5. Of all parts of an electronic structure calculation, assumptions about the organization of the two-electron integral tape have potentially the greatest impact on the program structures of individual modules. Is there any way to standardize the structure of the two-electron integral files on the standard data interfaces to avoid unresolvable problems?

1. Quantum Chemistry Program Modules

The following list of modules was determined to be those into which an electronic structure system could be divided so that standardization of communication protocols is a reasonable possibility.

a) System-wide Input Module

b) Integral Generation

c) SCF (RHF, UHF, MCSCF, GVB)

d) Transformation Program

e) Post-Hartree-Fock (CI, CC, MBPT, RSPT, SCEPA)

f) Properties, including graphics

2. Order of Module Execution

Standard interfaces between modules must provide for the execution of programs in normal order:

Input->Integrals->SCF->Transformation->Post-HF->Properties

as well as loop modes in which basis functions are optimized:

$$\begin{tikzcd}
\text{Input} \arrow[r] & \text{Integrals} \arrow[r] & \text{SCF} \arrow[r] \arrow[l, bend left] & \text{Transformation} \arrow[r] & \text{Post-HF} \arrow[r] \arrow[l, bend right] \arrow[ll, bend right] \arrow[lll, bend right] & \text{Properties}
\end{tikzcd}$$

3. Module Input/Output

Standard interfaces between modules will consist of several structured files passed from previously-executed modules to the next. In addition, "card" or "line" input specific to the module currently executing will be allowed. A single structured output file will contain the results of the module executed and be used for input to the next module. A normal printed output is allowed. An arbitrary number of temporary scratch files may be used by the module.

4. Specification of the Standard Data Interfaces between Modules

The specification of a standard data interface between program modules to achieve module insertability (plug-compatibility) is extremely detailed and must be specified and implemented on several real programs before a useful universal standard can be proposed. For example, an electronic structure computing system might contain the following information on the SCF data interface:

1) basis function type (GTO, STO, elliptic STO, etc.)
2) nuclear coordinates
3) nuclear charges
4) center names
5) orbital centers
6) orbital exponents
7) contraction matrix
8) tolerances
9) number of d functions

10) statistics about: number of nonzero integrals
    number of discarded integrals
    CPU times for integral computation
11) type of SCF method used (RHF, UHF, GVB, MCSCF)
12) eigenvectors
13) eigenvalues
14) total energy
15) electronic configuration
16) orbital energies
17) convergence criteria and method of computation

.... etc. ....

Is this all of the information which should appear at the end of an SCF calculation? Is it too much information? What specific formats should these records occupy?

Answers to these detailed questions cannot be established by a two-hour discussion. Therefore the following proposal was adopted.

A working group must be established by the NRCC to specify both the structure, service functions and contents of the standard data interfaces (logical and physical) between the modules identified in item 1 such that the modules are capable of functioning in the orders prescribed in item 2.

The final recommendations of the working group are to be submitted for publication in the International Journal of Quantum Chemistry, and other relevant journals, no later than one year after the working group is convened. It is anticipated that the working group consist of authors of major quantum chemistry systems such as GAUSSIAN 70, ALIS, MUNICH, etc.

Work on the specifications of standard data interfaces is expected to proceed in the following manner:

(3 weeks)	1. The working group chairman will solicit suggestions for Standard Data Interfaces (SDI's) from all chemists interested in electronic structure computations.
(1 week)	2. Specifications of the MUNICH system's Standard Data Interfaces and others are distributed to group members.
(2 months)	3. Each group member modifies the best SDI specifications to meet his program needs. No implementation of an SDI is attempted at this point.

(3 days)	4. A working group meeting is convened and a trial standard is agreed upon.
(3 months)	5. Group members implement the trial standard in their program systems.
(4 days)	6. A working group meeting is convened and a proposed implementable standard is agreed upon. A draft of a publication standard is prepared.
(2 weeks)	7. The chairman revises the draft and submits it for publication. All group members are co-authors.

5. A Specific Recommendation for the Standard Data Interface Working Group

Currently, two-electron integrals (ij/kl) are stored on files in many different formats. Possible classifications include:

(1) Order-unspecified labelled integrals (ij/kl) are stored on the file with a label IJKL specifying the integral. Zero (or approximately zero) integrals are not stored.

(2) Ordered integrals (no packing or compression, hence no labels need to be stored)

(a) canonically ordered integrals--integrals are in the order i>=j, k>=l, ij>=kl

(b) canonically ordered in symmetry blocks

(c) exchange (triplet) ordered integrals--i>=j>=k>=l, with 1, 2, or 3 integrals stored for a given IJKL location.

(d) distribution ordered (used in two-electron transformations) i>=j, k>=l. Hence for a given ij, all kl required for the transformation are available.

(3) Ordered Packed Integrals. Same as (2) except integral files have zeros removed by any of a number of packing (compression) schemes such as skip counts, labelling, etc.

Except for the two-electron integral file, assumptions about the particular structure of files processed by electronic structure modules have little impact on the design of the modules. However, many efficient modules assume that two-electron integrals will be presented in a special order. Modification of these programs to accept unordered integrals is impractical because the algorithm used is dependent for efficiency upon ordered integrals.

After considerable discussion took place weighing the benefits and potential inefficiencies possible, WG5 recommends that the SDI Working Group consider requiring that each module submitted to NRCC for inclusion in the electronic structure system be able to accept as input unordered, labelled two-electron integrals. Programs which require ordered integrals must supply a sort from unordered format to ordered format as part of the module. Each module may also have additional capabilities for accepting appropriately ordered integrals and omitting the sort.

Most working group members have agreed to provide and maintain the ability to process unordered, labelled two-electron integrals between the modules specified in item 1 in their own programs if it is adopted as part of the SDI.

WG5 Chairman
George Purvis

COMBINED REPORTS OF WG6 AND WG7

SMALL AND LARGE MOLECULE CRYSTALLOGRAPHY

This report is from the crystallographers of the conference and is the consensus of Working Groups 6 and 7. The report is concerned with portability and standardization of crystallographic programs, data bases, and computer graphics. The report is divided into a preamble which states the purpose of all this and a main body consisting of recommendations, to both the NRCC and the crystallographic community, as to how it may be accomplished. The body of the reports follows this outline:

I. Comments on FORTRAN standardization and portability.

II. Adoption of the RATMAC programming language.

III. Crystallographic data bases.

   A. The Binary Data File.
   B. The IUCr formatted file.

IV. Adoption of standardized programming conventions.

V. Standardization for crystallographic computer graphics.

VI. The NRCC to serve as a clearing house for distribution of crystallographic programs.

VII. NRCC sponsored workshops to promote standardization in programming and data structures.

PREAMBLE

All crystallographic laboratories in the United States have, or have access to, extensive libraries of computer programs which implement the existing methods used in x-ray diffraction and crystal structure analysis. Why, then, do we need to define software and data format standards now, when we have lived without them for twenty-five years? The answers are:

1. To facilitate exchange of programs among laboratories using a wide variety of processors.

2. To enable programs written by different people for different purposes to access a common, standard data file.

3. To avoid the waste of scientific manpower which has occurred in the past whenever an institution or company has decided to change its mainframe computer.

4. To minimize the dislocations which will occur if the currently used FORTRAN compilers are phased out in favor of later versions.

5. To insure that as new methods in crystallography are invented and developed, they are programmed in languages that are as portable and machine-independent as possible. Methods which can be immediately recognized are (a) less empirical treatments of extinction and thermal diffuse scattering; (b) more powerful direct methods, especially applicable to large molecules; (c) more sophisticated charge-density analysis methods; (d) more sophisticated thermal motion analysis to deal with segmented and anharmonic motion, both for small and large molecules; (e) improved non-linear least squares methods for refinement and (f) production of device-independent software for molecular graphics.

RECOMMENDATIONS

I. Comments on FORTRAN standardization and portability.

We strongly recommend that any changes to the FORTRAN standard be upward compatible. The proposed FORTRAN 77 standard is a major problem in this respect. NRCC should point out, both to the ANSI Standards Committee X3J3 and to such other major users as DOE and the National Laboratories, the millions of dollars and the man-years that will be required to convert existing major programs to FORTRAN 77. In addition, the NRCC should sponsor a member of the ANSI FORTRAN Standards Committee. In order to enhance the portability of all FORTRAN programs, NRCC should encourage programmers to use PFORT as a screening tool. All FORTRAN codes supported and distributed by NRCC should be accompanied by a PFORT verification report (Ryder, B.G., Software-Practice & Experience, Vol.4, 359-377 (1974)).

II. RATMAC Programming Language.

In order to proceed with the development of portable crystallographic software, a path must be chosen. We feel that the use of a preprocessor language will ease adaptation to different processors and dialects of FORTRAN, and will aid in the generation of rational, understandable code. We recommend the use of RATMAC (Munn and Stewart, Technical Report TR-675, University of Maryland Computer Sciences Center), a macro-enhanced version of RATFOR ("Software Tools," B.W. Kernighan and P. J. Plauger, Addison-Wesley, 1976) for crystallographic software development. We have focused on RATMAC for several major reasons:

1. RATMAC is in the public domain and can be distributed without licensing fees.

2. RATMAC has been developed and will be maintained by a group that is primarily concerned with crystallographic software portability.

3. RATMAC has a full set of capabilities as a structured programing language.

4. RATMAC allows for machine-specific dependencies to be handled through macros.

5. There already exists a core set of RATMAC routines for handling the recommended binary data file for crystallographic applications.

We recommend that NRCC distribute RATMAC and its associated documentation. This documentation is to include a user manual, an installation guide and a set of macros designed to achieve machine- and operating-system independence.

NRCC should publicize in Journals and Newsletters the availability of these materials.

III. Data Base Structure.

A. Binary Data File.

Whenever possible crystallographic software should be written to use a common data structure. This will greatly improve program portability. The Binary Data File designed by Stewart et al. for the XTAL80 system is reasonably compact, self-documenting, open-ended and comes with portable subroutines to read, write, edit, and maintain the system. This data structure should be used by as many programs as possible whether or not they are part of the XTAL80 system. NRCC should encourage this practice by distributing the appropriate software and documentation.

B. The IUCr Formatted File.

The International Union of Crystallography at the 1978 General Assembly formed a working group to define an international standard for archival storage and exchange of crystallographic data. The effort is necessary to remove human errors from the chain which transmits data from the researcher to the archival data bases. The results of the deliberations of the IUCr's working group (see Appendix) are to be presented to the 1981 General Assembly and if they are adopted, publication of crystal structures in the IUCr's journals will require presentation of all data in this format upon an acceptable medium. The NRCC must remain aware of the progress of these deliberations and should encourage programmers to provide tools to read and write the IUCr Standard Data File for easy data interchange.

NRCC should maintain communication between disciplines which are producing related chemical structural data bases to ensure that their formats are as easily interchangeable as possible.

IV. Standardized Programming Conventions.

For greatest ease of coding and use, it is recommended that sets of related routines be coded according to standardized programming conventions to form integrated program systems. Such systems offer the advantage of standard data structures, a consistent control structure, and availability of well-tested subroutines for standard funtions.

In order to demonstrate the efficacy and promote the acceptance of standard programs and data structures the NRCC should financially support the development of a pilot system of crystallographic programs to match the requirements of both small and large computers. The standardization and coding conventions of the XTAL80 system should be adopted for this purpose. In certain areas different algorithms will be necessary for the two types of machines. By and large such algorithms exist and should be incorporated into an integrated system centered around the recommended binary data file structure.

for all comers to criticize. An appropriate significant problem is multiple isomorphous replacement phasing of macromolecular structures. The theoretical aspects of the problem are well understood, the problem is of great importance to a significant community of users, and there is no completely satisfactory program which solves this problem.

The structure of the workshop should be as follows:

1. Participation should be limited to no more than ten scientists including specifically Professors Stewart and Munn.

2. The workshop should be held at the NRCC, which has a VAX-11/780 and one other major computer system. The other computer system presents difficulties with respect to portability. The code would be developed and debugged on the VAX, which has good tools for program development, and portability verified on the second system.

The schedule for the workshop would be:

1. The morning of the first day would be devoted to a RATMAC and XTAL80 tutorial. The remainder of the day would be devoted to analysis of various methods of MIR phasing and phase refinement, and selection of those to be incorporated in the new program.

2. The second day would be devoted to analysis and criticism of existing programs and methods, and a selection of specifications

3. The third day would be devoted to overall design and documentation. Some documentation details will have to be deferred.

4. The fourth, fifth and sixth days would be devoted to coding, debugging, and testing the code on the VAX.

5. The seventh day would be devoted to completion of documentation and preparation of a note describing the program.

6. Portability (or lack thereof) can be verified by a subgroup in two days.

This requires the following support from NRCC:

1. Transportation and expenses for visiting scientists.

To encourage this the NRCC should:

a. sponsor prototype workshops designed to:

   1) acquaint interested parties with the methodology of the XTAL80 system
   2) design programs
   3) write documentation
   4) write actual programs

b. support the distribution of updated versions of the University of Maryland XTAL80 primer and reports.

V. Standardization of Computer Graphics.

To insure portability of computer graphics programs there is a need to establish programming standards and develop a standard data base for use with interactive graphics. This need is underscored by the proliferation of various graphics devices, many of which have quite different implementation software.

To promote device independence, NRCC should encourage users to follow the proposed graphics standard by SIGGRAPH (Computer Graphics 13 #3) and should sponsor a presentation of the SIGGRAPH standards at an ACA meeting.

VI. NRCC Clearing House for Distribution Of Crystallographic Programs.

The NRCC should exercise leadership in encouraging the adoption of the programing and data file structure standards mentioned above. This will best be done by the NRCC acting as a clearing house for all programs but providing support only for programs which meet the above standards. In addition, programs developed under NRCC sponsorship or funding must meet the software standards in order to be distributed.

VII. NRCC Sponsored Workshops.

A. Macromolecular Workshop

The best possible demonstration of the power and flexibility of a program system is an actual portable implementation which can be widely distributed and tested by all interested parties. We propose that the specific virtues and portability of RATMAC and the Binary Data File be demonstrated by holding a workshop to produce working code

2. Secretarial assistance with documentation.

3. Free access to the VAX with 9-track tape, at least five constantly available video terminals and a line printer.

4. Access to the second computer system, on the scale of several hours equivalent VAX time.

5. Distribution and publicity for the resulting software and documentation.

The macromolecular special interests at this meeting regard this as a very high priority. The proposed workshop should be held in late October or early November.

B. Small Computer Workshop

A workshop for small computer (32K 16-bit words) users, along the lines outlined for the large molecule workshop, should be held in the spring of 1980. The workshop should be held at NRCC, using the VAX 11/780 in the RSX-11 compatibility mode. Not more than 12 scientists in the field, including Professors Stewart and Munn, should be present. The workshop should be approximately one week in length. Tutorials to familiarize the participants with RATMAC and the XTAL80 Binary Data File should be held during the first day of the week. A day should then be devoted to the development of machine-specific macros.

The remainder of the workshop will be spent developing generally-useful crystallographic codes. One such code would be a full-matrix least-squares routine. It is suggested that Dr. R. A. Sparks and Dr. A. C. Larson should participate in the development of this routine. A second code would be the adaptation of the search-match routine for the Powder Diffraction File to small machines; this work would be extremely useful and would have immediate applicability in several hundred laboratories. Dr. G. G. Johnson Jr., who designed the existing system, should take part in this work along with one or two others. The development work on these routines is not mutually exclusive and could be done concurrently. Together the two routines would represent a very significant step forward in the small computer field in crystallography.

C. Graphics Workshop

NRCC should sponsor a crystallographic graphics workshop devoted to both small and large molecule graphics programs. The objectives of such a workshop would be:

a. To review existing graphics software systems of various types,

b. To thoroughly acquaint the participants with the SIGGRAPH Standard,

c. To develop standard data structures and methods for handling the special types of data needed for interactive graphics.

Attendance should be limited to active programmers in the field and should be at a location where there are suitable graphics facilities.

WG6 and WG7 Chairmen
Eric Gabe and Stephen Freer

A ROSTER OF SOFTWARE TOOLS

by

Nelson H.F. Beebe
Departments of Physics and Chemistry
University of Utah
Salt Lake City, UT 84112

Tel: (801) 581-5254

During the course of the conference, a number of participants requested that a roster of software tools be developed for inclusion in the Conference Proceedings, the NRCC Newsletter, and possibly others. The preliminary review of the proceedings by the participants resulted in a number of suggestions which have been incorporated in this roster. Errors and omissions remain the responsibility of the author.

For each of the software tools discussed here, I have tried to include information about what they do and why, where the programs can be ordered, what they cost, and whether or not there are restrictions as to their use. For some, I have also taken the liberty of inserting some personal evaluations. Although this certainly may introduce personal biases, I feel that the comments may nevertheless prove useful to readers considering acquisition of some of these tools.

It is worth noting here that an organized effort is getting underway for the development of portable software tools, in the form of the TOOLPACK project. TOOLPACK is only a few months old, and probably will not have produced significant amounts of software before at least a couple of years have passed. Another development is the recent announcement (Comm. ACM 22, (No. 9), 545 (1979)) of a U.S. National Bureau of Standards effort to compile information about the use and availability of software tools. A third item of interest is the NBS contract announcement (NB79SBCA0128) for a FORTRAN 77 Analyzer, similar in some respects to the PFORT Verifier, but with enhanced capabilities.

Some of the following tools are noted as being available from the author of this roster. They will be provided on a master software tools tape which is being prepared, and will be available for unrestricted distribution for the cost of mailing a tape. Arrangements can be made for transfer of some of the smaller tools via intercomputer dialup through a file-transfer program on the DECSYSTEM-20, and also through the high-speed ARPA network.

FLECS

FLECS (FORTRAN Language with Extended Control Structures) is a preprocessor developed by Terry Beyer at the University of Oregon. It offers decision structures IF..., UNLESS..., WHEN...ELSE..., CONDITIONAL and SELECT (similar to PASCAL CASE statement), and looping structures DO, WHILE, UNTIL, REPEAT WHILE, and REPEAT UNTIL. It also has internal procedures. Statement formatting is rather rigid. FLECS has been distributed to over 300 sites, and is in the public domain. It may be freely modified and redistributed without restriction. Implementations for more than twenty different computing systems are available. It is available for a cost of $225 (USA) or $250 (non-USA) from

Computing Center
University of Oregon
Eugene, OR 97403

Tel: (503) 686-4394

The original author is no longer available for consultation on FLECS, due to its popularity.

RATFOR

The RATFOR preprocessor, developed by Brian W. Kernighan and Peter J. Plauger, and a number of software tools written in RATFOR are thoroughly discussed in their book "Software Tools" (Addison-Wesley, 1976), which should be required reading for everyone who is engaged in scientific software development. The RATFOR preprocessor and all the associated software tools are available on magnetic tape from the publisher. However, this is no longer the recommended source for these materials. Since their publication, a number of improvements have been made to the preprocessor and the accompanying tools. Notable among these is the macro extension to RATFOR, called RATMAC, developed by Robert J. Munn and James M. Stewart at the University of Maryland, and described elsewhere in the

Conference Proceedings. A Software Tools Users Group has been formed, and a Software Tools Communications newsletter is now being produced. The editor is

Debbie Scherer
Computer Science and
Applied Mathematics Department
Lawrence Berkeley Laboratory
University of California
Berkeley, CA 94720

Tel: (415) 486-5881

and one can be added to the mailing list simply by writing to her. The Software Tools Users Group is developing a master tape containing all of the tools, and this will be available for a modest handling charge.

SFTRAN/3

SFTRAN/3 is a structured FORTRAN preprocessor developed by Charles L. Lawson and John A. Flynn at the Jet Propulsion Laboratory of the California Institute of Technology in Pasadena. It offers decision structures IF...THEN...ELSE and DO CASE, and looping structures DO FOR, DO WHILE, DO UNTIL, DO FOREVER, and DO BLOCK. There are CYCLE and EXIT statements for skipping to the next loop cycle or jumping out of a control structure. There is also a procedure facility. Of all the structured FORTRAN preprocessors that I have seen, SFTRAN appears to give the clearest code.

Machine-readable documentation and source code are available from C.L. Lawson at JPL or from N.H.F. Beebe at the University of Utah. By early spring, 1980, we hope to have a formal distribution service established with versions for a number of different machines.

SPARKS

SPARKS is a structured FORTRAN preprocessor developed by Ellis Horowitz and Sartaj Sahni to illustrate algorithms in their books "Fundamentals of Data Structures" (Computer Science Press, 1976) and "Fundamentals of Computer Algorithms" (Computer Science Press, 1978). It is in the public domain, and may be obtained from the authors for a charge of $20 by writing to

Dr. Ellis Horowitz
SPARKS Users Group
Computer Science, Powell Hall
University of Southern California
Los Angeles, CA 90007

The PFORT Verifier

The PFORT Verifier is a program which can be used to automatically verify that other FORTRAN software adheres to a subset of 1966 ANSI FORTRAN called Portable FORTRAN. Portable FORTRAN has been described by one of its authors, Barbara G. Ryder, in an article "The PFORT Verifier", Software -- Practice and Experience 4, 359-377 (1974).

The Verifier not only carries out syntax checking of individual program units, but also checks inter-routine communication through argument lists and COMMON blocks, something which few FORTRAN compilers do. It is available from two sources:

Quantum Chemistry Program Exchange
Room 204
Department of Chemistry
Indiana University
Bloomington, IN 47401

Tel: (812) 337-4784

Ms. Irma Biren
Computing Information Services
Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

Tel: (201) 582-3000

From either source, the charge is $30 which covers the cost of a 600 ft magnetic tape, documentation, and postage. Bell Laboratories has recently placed the PFORT Verifier in the public domain, and requires only that recipients retain the copyright notice in the code. Users are free to distribute their copies of the Verifier.

The PFORT Verifier is itself written in PFORT, and installation requires only the writing of two short subroutines for the packing and unpacking of characters, plus possibly changing a couple of assignment statements defining I/O units in the MAIN program. The Installation Guide recommends rewriting subroutine MAPCHR to use a table lookup, rather than a linear search; on both IBM and DEC machines, I have found that this makes no more than a two percent difference in execution time, and the exercise is probably not worthwhile. It is, however, worthwhile to increase the dimension of array DSA(*) in COMMON /CTABL/ from 5000 to about 10000, and change the assignment statement defining LDSA in the main program. If the local loader balks at COMMON blocks of different lengths, then dimension changes must be made in the 37 other routines which reference /CTABL/. If this change is not made, a program unit longer than about 300 lines tends to result in a table overflow which causes the Verifier to skip further processing on that unit.

DAVE

DAVE is a FORTRAN tool developed at the University of Colorado in Boulder by Leon Osterweil and colleagues. Its purpose is to perform global data flow analysis of FORTRAN programs, checking for use of variables before initialization. There is apparently a considerable overlap with some of the functions of the PFORT Verifier. A survey article by L.D. Fosdick and L.J. Osterweil about data flow analysis and DAVE has been published in Computing Surveys (Vol. 8, No. 3, pp. 305-330, September 1976).

DAVE is in the public domain, and can be ordered from its authors at

Department of Computer Science
Campus Box 430
University of Colorado at Boulder
Boulder, CO 80309

Tel: (303) 492-0111

An order form and questionnaire will be sent on request. The cost is $100, which covers a magnetic tape, documentation, and postage. The distributors tailor a version of DAVE to the recipient's computer, since it uses extensive bit-field packing to save its elaborate data structures in minimal storage. In addition, FORTRAN random-access I/O facilities are required. This may require the user to simulate the CDC OPENMS/READMS/WRITMS/CLOSMS random-access I/O routines.

DAVE is a large program, and normally runs as four separate job steps. On the CDC 6400, these require memory sizes of 40K, 114K, 121K, and 135K octal words, respectively. The User's Manual is about 250 pages long, and contains detailed information on how to use DAVE, and on how it works. This should greatly facilitate user modifications to the system. DAVE has been distributed to more than 100 installations, and recipients are free to modify and distribute it further.

DAVE by and large expects the input FORTRAN code to adhere to 1966 ANSI FORTRAN. However, it does accept a few extensions, among them the PROGRAM and IMPLICIT statements and array initialization by array name in DATA statements. This should ease the processing of quantum chemistry software which frequently makes heavy use of these extensions, and is in sharp contrast to the PFORT Verifier which abandons processing of a program unit when it finds an unrecognizable statement.

STRUCT

STRUCT is a utility which promises to be of great use in cleaning up existing FORTRAN code. From a program unit written in ordinary FORTRAN, it attempts to produce a structured FORTRAN equivalent, wherever this is possible. Of course, it is quite easy to write GO TO statements in such a way that no human, and no program either, could ever unravel the flow of control, so the usefulness of STRUCT may be in inverse proportion to the complexity of the input FORTRAN code.

STRUCT was developed by Brenda S. Baker at Bell Laboratories and has been described by its author in some detail in the literature (Conference Record of the Third ACM Symposium on Principles of Programming Languages, 113-126 (1976)). STRUCT is written in the language C. C is a high-level language intended for systems programming. It was designed by Dennis Ritchie and is used for about 95% of the programming in the UNIX operating system. UNIX is one of only two operating systems which have been successfully ported to machines of drastically different architecture. Another Bell Laboratories scientist, Steve Johnson, has developed a portable compiler for C. This compiler produces an output pseudo-code which can then be assembled or interpreted into machine code for the host computer.

UNIX system (Release 7 or later), but academic institutions can obtain the entire UNIX system from Bell Laboratories for a very modest cost. UNIX is most widely used on the larger models of the ubiquitous PDP-11 computer, and readers at universities will probably be able to find on-campus computers which may already be using UNIX.

Basic Linear Algebra Subroutines (The BLAS)

The BLAS were developed as part of the LINPACK project (see below), and FORTRAN versions of some of the BLAS are available on the LINPACK distribution tape. They have been described in the following articles:

C.L. Lawson, R.J. Hanson, D.R. Kincaid and F.T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," Assoc. Comput. Mach. Trans. on Math. Software, 5, 308-323 (1979). (Also available as Sandia Laboratory Report SAND 77-0898, Albuquerque, 1977.)

J.J. Dongarra and A.R. Hinds, "Unrolling Loops in FORTRAN", Software -- Practice and Experience 9, 219-226 (1976).

The BLAS consist of 38 FORTRAN-callable subroutines for common operations of numerical linear algebra. These include:

* copy a vector
* swap two vectors
* dot product of two vectors
* constant times a vector
* constant times a vector plus a vector
* set up Givens rotation
* apply Givens rotation
* set up modified Givens rotation
* apply modified Givens rotation
* 2-norm (Euclidean length)
* index of element with maximum absolute value

The complete set of FORTRAN BLAS, plus assembly language versions for the IBM 360/370, CDC 6000/7000/Cyber, and UNIVAC 1108 machines, as well as an extensive test program, are available from

International Mathematical and Statistical
Libraries, Inc.
Sixth Floor, GNB Building
7500 Bellaire Boulevard
Houston, TX 77036

Tel: (713) 772-1927

The distribution charge of $45 covers the postage charges and a magnetic tape containing documentation and software.

Assembly language versions of the BLAS for the DECSYSTEM-10 and -20 computers with hardware double precision are available on the Utah Software Tools tape.

PORT Library Framework

Bell Laboratories has developed a large portable mathematical subroutine library, called the PORT Library, which is being marketed for a sizable fee. A special nucleus of subroutines, called the PORT Library Framework, was developed for the parametrization of the environment and for run-time allocation of working storage from a large array in COMMON. This nucleus is available from

International Mathematical and Statistical
Libraries, Inc.
Sixth Floor, GNB Building
7500 Bellaire Boulevard
Houston, TX 77036

Tel: (713) 772-1927

The distribution charge of $45 covers the postage charges and a magnetic tape containing documentation and software. The material is all in the public domain and passes the PFORT Verifier, as required for publication in the ACM Transactions on Mathematical Software (TOMS).

The PORT Library has been described by P.A. Fox, A.D. Hall, and N.L. Schryer in an article "The PORT Mathematical Subroutine Library", TOMS 4, 104-126, 177-188 (1978). Machine parameters are included in comment statements for the following computing systems:

* Burroughs 1700/5700/6700/7700
* CDC 6000/7000/Cyber
* CRAY 1
* Data General Eclipse S/200

* DEC-10 (KA and KI processors)
* DEC PDP-11 (both 16-bit and 32-bit INTEGER support)
* HARRIS Slash 6/Slash 7
* Honeywell 600/6000
* IBM 360/370
* Interdata 8/32
* SEL Systems 85/86
* UNIVAC 1100 (FTN V Compiler)
* XEROX SIGMA 5/7/9

Support for other machines is easily added. P.A. Fox has recently made available to the author constants for the DEC VAX 11/780 which have been used for installing the PORT Framework on the NRCC VAX computer.

EISPACK/2

EISPACK/2 is the second release of the Eigenvalue System Package developed by the National Activity to Test Software (NATS). It is based heavily on Algol procedures developed during the 1960's and published in the journal Numerische Mathematik. These were finally collected by J.H. Wilkinson and C. Reinsch in their book "Handbook for Automatic Computation, Vol. II. Linear Algebra" (Springer-Verlag, 1971). The principal references for EISPACK/2 are:

B.T. Smith, J.M. Boyle, J.J. Dongarra, B.S. Garbow, Y. Ikebe, V.C. Klema and C.B. Moler, "Matrix Eigensystem Routines -- EISPACK Guide", Springer Lecture Notes in Computer Science, Vol. 6, 2nd edition, 1976.

B.S. Garbow, J.J. Dongarra, C.B. Moler and B.T. Smith, "Matrix Eigensystem Routines -- EISPACK Guide Extension", Springer Lecture Notes In Computer Science, Vol. 51, 1977.

EISPACK/2 is available from two sources:

International Mathematical and Statistical
Libraries, Inc.
Sixth Floor, GNB Building
7500 Bellaire Boulevard
Houston, TX 77036

Tel: (713) 772-1927

National Energy Software Center
Argonne National Laboratory
9700 South Cass Avenue
Argonne, IL 60439

Tel: (312) 972-2000

The charge from IMSL is $75. Principal Investigators holding Department of Energy grants can obtain software from NESC without charge. EISPACK/2 is in the public domain, and there are no restrictions on its distribution. Some 22 routines contain assignment statements setting the machine precision or the machine radix. Since these are different for each computer, special versions must be ordered, or one must laboriously make the changes by hand. The double precision versions require double precision complex arithmetic (COMPLEX*16).

I have prepared both single and double precision versions which do not require double precision complex arithmetic, and which obtain their machine constants from functions in the PORT Library Framework. I have also made a few minor corrections in order to make both single and double precision versions pass the PFORT Verifier. Both versions are available on the Utah Software Tools tape.

LINPACK

LINPACK is a collection of routines for solving in-core systems of linear equations, as well as for computing matrix inverses and the singular value decomposition. It is perfectly portable, and contains no machine-dependent constants whatever. It is available from the same sources as EISPACK/2 for the same costs. The principal reference is

J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart, "LINPACK Users' Guide", SIAM Publications, (33 South 17th St, Philadelphia, PA 19103, Tel: (215) 564-2929), 1979.

The publishers maintain a mailing list of purchasers, and automatically send out error corrections for the LINPACK Users' Guide.

FORSTA

The Harwell Subroutine Library contains a subroutine OE02A which can be used to automatically insert statement frequency counters in FORTRAN programs. I have modified and extended this to also optionally insert timing traps at entry and exit. The resulting utility, FORSTA (for FORTRAN STAtistics), can be quite useful in tracking down just which parts of a program are used most heavily and are therefore candidates for closer scrutiny with a view to code optimization. The statistics, which appear in both tabular and histogram form, can also assist in detecting sections of code which are either not executed at all, or are not entered the expected number of times. The statistics can be output at any time to a user-specified file, so that dumps can be taken at predetermined times to avoid losing all the information in the event of the job aborting. FORSTA is available on the Utah Software Tools tape.

EDIT

EDIT, for want of a better name, is a program which cleans up FORTRAN programs. Features include:

* indenting of loops
* reassignment of statement numbers in ascending order
* blocking of comments by delimiter lines to increase visibility
* conversion of quote-delimited strings, "...", to Hollerith strings, nnH...
* conversion of FORTRAN II I/O statements to FORTRAN IV
* changing precision of functions and constants (single, double, and quadruple precision)

EDIT has been used on several hundred thousand lines of FORTRAN code, including several large quantum chemistry systems such as IBMOL, MOLECULE, POLYATOM, HONDO, and BIGGMOLI. Programs like EDIT have come to be called "pretty-printers", and such utilities for BASIC, PL/1, PASCAL, LISP, REDUCE, and ALGOL 60 have appeared in the computing science literature. Another FORTRAN pretty-printer named TIDY is apparently available for CDC machines, at least, as part of the CDC Users Group Library, VIM Doc L4. I have not yet seen specific documentation about TIDY, and cannot comment on its facilities or portability.

Macro Processors

Assembly language programmers have for many years routinely used macro facilities to assist in the usually rather tedious job of writing assembly code. A macro is essentially a template for automatic production of similar sections of code, usually with minor substitutions of names, and sometimes with conditional selection of alternate code sequences. This is a frequent requirement in assembly language programming, but macro facilities can also be useful in higher-level languages. A simple example occurs in dealing with symmetric matrices stored in lower-triangular packed form. The index of the (I,J) element is obtained from the expression (I*(I-1))/2 + J, where I must not be less than J. If I is less than J, then I and J must be interchanged in the expression. In principle, one could write

```
      IJ = (MAX0(I,J)*(MAX0(I,J)-1))/2 + MIN0(I,J)
```

to do the index computation in one line, but the code produced by most FORTRAN compilers for this is usually far from ideal, so instead one usually sees the sequence

```
      IF (I - J) 10,20,20
   10 IJ = (J*(J-1))/2 + I
      GO TO 30
   20 IJ = (I*(I-1))/2 + J
   30 ...
```

This is tedious and error prone, and also badly clutters the code. Most macro processors would permit a macro, say PINDEX, to be defined, allowing a simple invocation in the form

```
      IJ = PINDEX(I,J)
```

which would actually result in generation of the IF and GO TO statements necessary to compute the index in-line.

Macro expansion of course requires at least one pass through the program text, and there are thus at least four different stages at which the macro processing could be applied.

The first of these could be in a text editor during the actual creation of the source code, assuming it is done on-line. This would offer certain conveniences for writing the code, but once the text was finally saved on a file, references to the macros would be replaced by their expansions, and one would then be back with the cluttered code situation.

The second place a macro processor could be used is as an integral part of a software maintenance system. This would have the advantage that the macro processing could be language independent, and indeed, might even be used to operate on input data, rather than program text.

The third place is as an independent processor invoked prior to compilation. This too could be language independent.

The fourth alternative is integration into the language compiler, so that each statement is scanned for macro substitutions before being analyzed for compilation. This has been the traditional approach with macro processors embedded in assemblers, and a compile-time macro facility is actually included as part of the language definition for PL/1. This approach has the disadvantage that the macro facilities are usually restricted by the syntax of the language in which they are embedded.

In practice, I/O often accounts for a major portion of the execution time in text processing systems, and the first, second, and fourth alternatives reduce the cost by doing the macro processing at the same time as other required processing is going on. The third alternative incurs the additional overhead of an extra pass over the code, but is the best candidate for a portable implementation. At some installations, it might be acceptable to also integrate such a stand-alone macro processor into an editor, software maintenance system, language preprocessor, or compiler, so that it could be made available without incurring possibly unacceptable additional overhead of a separate job step.

There are a great many macro processors around, for they are usually simple enough that they can be implemented in the order of 500 to 2000 lines of code, making them good candidates for student projects or "quick and dirty" weekend implementations. Most are designed with particular host languages in mind, and few are portable, although some have been designed expressly to increase portability by acting as bootstrap languages and preprocessors.

Macro processors fall into essentially two distinct categories. The simplest are those that are basically substitutional; macro invocations are usually flagged by special characters, as for example the percent sign is used in PL/1's macro facilities (%IF, %DO, ...). Each macro invocation usually contains one or more arguments which are used to replace symbolic parameters. Thus the PINDEX macro noted above might be invoked in one place as PINDEX(I,J) and in another as PINDEX(MU,NU), with MU and NU appearing in the same places that I and J do in the macro expansion.

The second category is the pattern-matching macro processor which is more general and powerful, and usually also slower. Pattern-matching processors usually contain the substitutional facilities of the simpler processors as well. As an example of their use, suppose that one had a FORTRAN program written using expressions involving COMPLEX variables, and that double precision complex arithmetic had been determined to be necessary, but was not supported by the local FORTRAN compiler. A pattern-matching macro processor could be used to recognize arithmetic expressions and turn them into code sequences which used real arithmetic. Thus the statements

```
      COMPLEX A,B,C,D
      A = CEXP(D)/(B*C)
```

might be automatically converted by the pattern matcher to the sequence

```
      DOUBLE PRECISION A(2),B(2),C(2),D(2),TEMP1(2),TEMP2(2)
      CALL DCEXP (TEMP1,D)
      CALL DCMULT (TEMP2,B,C)
      CALL DCDIV (A,TEMP1,TEMP2)
```

An extensive discussion of macro processors may be found in the books

"Software Portability", Ed. P.J. Brown, Cambridge University Press (1977).

P.J. Brown, "Macro Processors and Techniques for Portable Software", Wiley (1974).

W.M. Waite, "Implementing Software for Non-Numeric Applications", Prentice-Hall (1973).

Brown and Waite both discuss a pattern-matching macro processor called STAGE2 which was developed by Waite, and a listing of about 3700 lines of code for the entire STAGE2 system is contained in an appendix of Waite's book. About 1/3 of this code is in Portable FORTRAN, and the remaining 2/3 is bootstrapped from the FORTRAN code portion.

A simpler substitution macro processor called M6 was developed by M.D. McIlroy and W.S. Brown of Bell Laboratories and is included as part of the ALTRAN symbolic and algebraic manipulation language (see below). M6 consists of about 600 lines of Portable FORTRAN and has been implemented on many different computers. M6 influenced the macro facilities of D.M. Ritchie's language C used for the UNIX system, and these were adopted by Kernighan and Plauger and implemented in RATFOR in their book Software Tools. This macro processor in turn was substantially improved and extended by R.J. Munn and J.M. Stewart at the University of Maryland. Their RATMAC preprocessor combines the macro processor with RATFOR and was mentioned in the RATFOR section of this roster.

For computational chemists, the macro processors likely to be most easily available and maintainable are the M6 processor and the RATMAC preprocessor. Neither offers pattern-matching facilities, but then neither are these required as often as simple substitutional facilities. Programming for portability can be greatly aided by a macro processor which can take care of the code sections which require knowledge of details of the implementation and host computer.

MATLAB

MATLAB is an interactive matrix laboratory developed by Cleve Moler at the University of New Mexico. It incorporates routines from both EISPACK/2 and LINPACK and is written entirely in FORTRAN for wide portability. MATLAB provides the user with convenient facilities for on-line computations using scalars, vectors, and matrices. It can be used like a simple desk calculator, but is actually much more powerful. It has its own input language which permits users to program computations much as they would program in FORTRAN or PASCAL. It differs substantially from these in that variables are defined only when they are introduced at run-time, and they can be redefined in type as well as in value at any time. In this respect, MATLAB resembles the language APL, but requires neither special symbols nor cryptic notations familiar only to those schooled in the language.

A convenient facility of MATLAB is the ability to store and retrieve data and programs from external files, so that it is quite easy to produce matrices in a FORTRAN program, and then save them on a file for use with MATLAB. MATLAB also has commands for saving and restoring its entire environment, so that one can go home and come back the next day, picking up exactly where one finished the day before.

MATLAB can be obtained for a modest handling charge from its author:

Cleve B. Moler
Department of Mathematics and Statistics
University of New Mexico
Albuquerque, NM 87106

Recipients are free to modify and further distribute MATLAB if they wish to do so. MATLAB is rather easy to install. At the University of Utah, about one day's work was required to get it operational in a form where it could be used conveniently and interface to the DECSYSTEM-20 file system by file name, rather than by FORTRAN unit number. A copy of MATLAB with the DECSYSTEM-20 interface is also available on the Utah Software Tools tape.

Module Cross-Reference Utilities

When working with large software systems, or even small ones which are unfamiliar, it can be enormously useful to have available "calls" and "called by" cross-reference lists. These can of course be made up by hand, but this is both tedious and error-prone, and the labor can be prohibitive with systems having hundreds of subroutines. Some people make a practice of including in comment statements for each routine a list of called routines, and it may then be possible to make selective "calls" listings. The "called-by" listings are not as easily obtained, and it is inadvisable to list in comments in a given routine the names of all those routines which call it, since such lists are bound to rapidly get out of date.

Linkage editors and loaders have all the information necessary to build "calls" and "called by" cross-reference lists, but I not yet seen an operating system in which these listings are available in a convenient form. The compiled code is the logical place to build the tables from, since the listings can be automatically regenerated each time the load library is updated, and one can thus always be assured of their correctness.

I therefore developed a FORTRAN program on the CDC 6400 which works with the SCOPE, KRONOS, and NOS operating systems for all CDC 6000, 7000, and Cyber models, and constructs these listings from a library file. The listings are formatted in neat tables, using only 65 columns, with the intention that a "C" can easily be inserted in Column 1, so that they can become part of the main program documentation. This program, MAP, reads object code files which are naturally extremely machine-dependent, and the code was written rather machine-dependently. However, apart from the extraction of symbols from loader control blocks in the object code, the rest of the program concerned with sorting and printing the cross-reference lists could have been done without knowledge of the underlying machine.

When I moved back to an IBM operating system, I rewrote MAP from scratch, and had to use some assembly language, since IBM FORTRAN does not provide access to members and directories of partitioned data sets in which IBM load libraries are stored. The symbol sorting and listings were arranged to be more-or-less independent of the underlying system, and the resulting utility has been named PDSMAP.

Since then, I have moved on to a DEC-20, and have modified PDSMAP to become LIBREF, which should work on both the TOPS-10 and TOPS-20 operating systems for the DEC-10 and 20 computers. Although MAP, PDSMAP, and LIBREF are all machine-dependent, their output listings are identical, and I have found them absolutely indispensable in my own software development. MAP, PDSMAP, and LIBREF are available from the author on the Utah Software Tools tape.

DOCUMENT

DOCUMENT is a machine-independent FORTRAN document formatter, intended for the maintenance of software documentation which is kept in machine-readable form, along with its associated software. Output of DOCUMENT is intended to be printed on an ordinary line printer, preferably one having lower-case letters.

In practice, it is useful to distribute DOCUMENT along with other software, and to include both the raw input documentation files, and a printer-ready copy of the output of DOCUMENT. In this way, neat documentation is available immediately, and as soon as the software has been installed (which usually requires modifications of some sort), the raw document can be updated to reflect the local changes. Features of DOCUMENT include

* variable line widths and page sizes selectable at run time

* optional right-margin justification by blank padding (just like this report)

* automatic page numbering with one or two header and/or footer lines

* bibliography formatting

* easy subscripts and superscripts (printed one line above or below)

* backspacing for overstriking (frequently not available on ASCII line printers, however)

* underlining

* automatic centering of text

* easy-to-type mathematical text (sums, integrals, derivatives, products, and limits), although limited by the inflexibility of most line printers

DOCUMENT is available from the author on the Utah Software Tools tape.

FMTFOR

FMTFOR is a utility program which processes FORTRAN programs, outputting variable names in specification statements in alphabetical order in neat columns. Although this may seem like a "frill", I have come to regard it as one of my most useful tools. The file use-count statistics on our DECSYSTEM-20 show that it has been used on average about 220 times per month at our small installation.

It was originally written to clean up a large quantum chemical two-electron integral subroutine which has type declarations and COMMON and EQUIVALENCE statements stretching over five pages of listing. A switch can be set to permit variables in COMMON to be ordered alphabetically as well as being tabulated; this of course can only be done when COMMON blocks of the same name are identical everywhere, and when storage alignment requirements do not dictate otherwise. FMTFOR is written in PASCAL, but as time permits, a FORTRAN version will be made and included as a new feature of EDIT.

FMTFOR is available from the author on the Utah Software Tools tape.

The Pascal User's Group

All of the software tools in the preceding sections are oriented towards FORTRAN software, and most are written in Portable FORTRAN. PASCAL is a language which is rapidly gaining in popularity, and for which an ANSI committee is developing a Standard. It has strongly influenced subsequent languages, particularly the U.S. Department of Defense language, ADA, which was developed to replace FORTRAN, COBOL, PL/1, JOVIAL, assorted assembly languages, and a few other high-level languages for defense applications.

For several reasons which I will not go into here, PASCAL is not well-suited to the development of large-scale scientific programs. That situation may change in the future if the standardization efforts succeed in removing the detriments which currently exist, but ADA is already in a position for consideration, although it has yet to be widely implemented. However, PASCAL is becoming more and more widely available. There are at least two FORTRAN compilers which have been written in PASCAL, and there is also a computer chip designed to execute P-Code, which is a pseudo-machine code produced by several PASCAL compilers. One of the PASCAL-coded FORTRAN compilers was developed as a joint project of the Los Alamos and Lawrence Livermore National Laboratories. It is exciting to note that the same compiler is used to produce highly-optimized pseudo-code which is translated to run on the Cray computer at Los Alamos and the S-1 at Lawrence Livermore, machines which have rather different architectures.

A large and active PASCAL User's Group (PUG) exists, and publishes the PASCAL News, a lengthy newsletter (often 100 pages or more) approximately four times a year. Membership in PUG costs $6 per year, and membership application forms can be obtained from

PASCAL User's Group
c/o Andy Mickel
University Computer Center, 227 EX
208 SE Union Street
University of Minnesota
Minneapolis, MN 55455

The reason for mentioning PUG in this report is that the PASCAL News is generally filled with listings of lots of useful software tools. Many of these are oriented towards PASCAL applications, but some could also be useful for FORTRAN programmers. Transcription of a language like PASCAL into FORTRAN can be very tedious, but provided that PASCAL's powerful data structures and recursive procedure invocation are not too heavily used, it will often be straightforward to recode PASCAL algorithms into SFTRAN/3, FLECS, or RATMAC, all of which have PASCAL-like control structures, and most importantly, procedures.

Symbolic and Algebraic Manipulation Languages

This section deviates from most of the previous ones in that one particular software tool will not be discussed. Many readers in the chemistry community may be unfamiliar with languages for symbolic and algebraic manipulation (SAM), and in some cases might benefit from the availability of such a language.

Briefly, SAM languages permit one to write programs in procedural languages resembling Algol, PL/1, or PASCAL, in which statements like A := B*C result not in replacement of A with the value obtained by multiplying the numeric values of B and C, but rather in assigning to A an expression obtained from the product of the expressions represented by B and C. Thus, if B contained the expression "(X+Y)" and C the expression "(X**2 + 3)", A would become "(X**3 + Y*X**2 + 3*X + 3*Y)". SAM languages permit rather general expression substitutions and evaluations, and some, such as REDUCE, even allow the user to output expressions as FORTRAN code. Most contain facilities for symbolic differentiation, and some now have rather powerful symbolic integration facilities.

More information about the subject can be obtained by examining the proceedings of the Association for Computing Machinery Conferences on Symbolic and Algebraic Manipulation and from the SIGSAM Bulletin published quarterly by the ACM, and now in its fourteenth year. A survey article by J.H. Griesmer published in the SIGSAM Bulletin (Vol. 10, No. 2, pp. 30-32, May 1976) may be useful. There are unfortunately no textbooks devoted to the subject of symbolic and algebraic manipulation languages, although at least one is in preparation.

No one has any difficulty in understanding that computers are useful for numeric computations, but many people seem to find it hard to imagine using a computer for symbolic operations like integration and differentiation. An example from quantum chemistry may help to illustrate the useful of SAM languages. In ab initio computations, one of the main organizational problems involves the efficient use of the vast numbers of two-electron integrals which are stored, often in random order, on external media. The trick is to pick up each two-electron integral in turn, and then to add its contribution everywhere it belongs. In some theories, the two-electron integrals enter in rather involved formulas, and the programming is complicated by the fact that storage economization usually dictates that only one of a group of equivalent integrals is ever stored, so that each integral generally must be reused under the guise of several different names. A SAM language can be quite useful for automatically making index permutations and other substitutions to allow expressions to be produced in a form suitable for programming, and even for producing the FORTRAN code itself. Although this requires little of the power of some SAM languages, it is nevertheless a very useful application. Another potentially valuable application of SAM languages would be the production of FORTRAN code for special cases derived from general case formulas, as is often desirable in Gaussian molecular integral programs.

The names of several SAM languages will now be mentioned to at least make them known to readers, who may find that they already have one or more of them available locally, but have never actually used them. The languages are given in alphabetical order. I have had personal contact only with FORMAC and REDUCE, and am therefore not in a position to make recommendations of one SAM language over another. However, few of these systems are portable, so in many cases, the field of choice will be drastically narrowed once the make of the local computer has been specified. The assistance of Martin A. Griss of the Utah Symbolic and Algebraic Computation Group in preparing this section is gratefully acknowledged.

ALTRAN was developed at Bell Laboratories and is among the most portable of the SAM languages, since it is based on FORTRAN and a portable macro processor, M6. Its development has been frozen, but the processor is still available from Bell Laboratories.

CAMAL was developed at Cambridge University by J. Fitch and others, and has contributed significantly to the field of symbolic integration. Its base language is BCPL which is available on several different computers.

FORMAC is one of the earliest SAM languages and was originally FORTRAN based; a FORTRAN version is available from the U.S. Naval Postgraduate School in Monterey, California. The current principle version of FORMAC is PL/1 based, and is available through the IBM SHARE organization. A new release of FORMAC 73 was announced by K.A. Bahr in the SIGSAM Bulletin (Vol. 12, No. 1, p. 6, February 1978); Bahr has been one of the most active supporters of FORMAC in recent years.

MACSYMA and MATHLAB, its precursor, have been developed at MIT's Project MAC. MACSYMA is claimed to be the most powerful SAM language available, but is unfortunately also one of the least portable. It is based on a variant of LISP known as MACLISP, and is available on DEC-10 and -20 computers, and also on the Honeywell MULTICS system.

Most SAM languages require a large amount of resources to maintain symbolic expressions in memory and manipulate them. A recent interesting development is muMATH-79, which is intended to run on a microprocessor and therefore be widely available at a very low cost, so that it can even be used at the elementary school level. Its developer, David Stoutemeyer of the University of Hawaii, is a 1979-80 ACM National Lecturer, and has described muMATH-79 in two recent articles (BYTE Magazine, Vol. 4, No. 8, pp. 176-192, and SIGSAM Bulletin, Vol. 13, No. 3, pp. 8-24, August 1979). The first article also contains a discussion of several other SAM languages.

REDUCE is a LISP based language developed by a physicist turned computer scientist, Anthony C. Hearn of the University of Utah. In order to make REDUCE more portable, Hearn and colleagues have defined a Standard LISP, and developed compilers for this for a variety of machines. A compiler is even available which translates Standard LISP into FORTRAN, making it relatively easy to bootstrap onto a new machine. REDUCE is operational at more than 175 computing installations around the world, and runs on Burroughs, CDC, CEMA, DEC, Fujitsu, Hitachi, IBM, Honeywell, ITEL, Siemens, Telefunken, and UNIVAC computers. It is available from the Computer Science Department of the University of Utah for $100.

SAC-1 is a SAM system developed by G.E. Collins and colleagues at the University of Wisconsin. Unlike most other SAM systems, SAC-1 is not an independent language, but is instead used via FORTRAN subroutine calls. This means that it is not interactive, which is often an important requirement of SAM computations. However, its FORTRAN base makes it quite portable.

SCHOONSHIP is a SAM language developed in Holland. It has been widely used in Europe, and is available for IBM computers only.

SCRATCHPAD is a large SAM system developed by IBM. It has not yet been released as a program product to users of IBM equipment, but is very powerful and has received a lot of discussion in the SAM literature.

SYMBAL was developed by Max Engeli at the Technical University of Zurich in Switzerland. It remains one of the few SAM languages available on CDC computers.

TEX and METAFONT

TEX, for Tau Epsilon Chi, and rhyming with "blecchhh", according to its author, Donald E. Knuth of Stanford University, is a typesetting system for technical text. In the past, authors of scientific articles and books usually prepared their manuscripts in typewritten form and sent them to a journal or publisher where they were refereed or reviewed. Resulting modifications of the manuscript often required its retyping, before it was sent back for typesetting. This meant that it had to be typed again, this time into a typesetting system by personnel unfamiliar with the contents of the material, and particularly if it contained mathematical text, usually resulted in many errors being introduced. Consequently, galley proofs of the typeset material were returned to the authors for correction before final publication. With complicated or lengthy manuscripts, more than one round of this could be required.

This situation is about to change, thanks to the exciting development of Knuth's TEX. TEX differs from most current typesetting systems is that it is written in a high-level language (SAIL, and presently being recoded into PASCAL) and produces an output device-independent file containing the formatted text. A device-dependent driver program then converts the file into a form in which it can be presented to a particular typesetting system. Device drivers for Varian 9511, Versatec 3200, Alphatype CRS, Xerox XGP, and Xerox Dover raster scan printers already exist, and many more are expected to be developed.

Knuth is currently working on METAFONT, a system for font design to be used with TEX. A TEX Users Group, to be called TUG, is being formed to coordinate the use and dissemination of TEX.

The American Mathematical Society has already made a firm commitment to move its entire journal and monograph typesetting operation to the TEX system, and expects to complete this by mid-1981. They expect this to significantly reduce production costs, which in turn may allow a reduction in journal subscription prices. Great efforts are being made to encapsulate journal style differences in macros, so that manuscripts need not be hand-modified if the target journal changes.

TEX is going to be widely distributed essentially at cost, so that it should not be long before authors will be in a position to produce their own galley proofs, right from the start, and then to provide their manuscripts in machine-readable form to their publishers. For information on TUG, write to

TUG
American Mathematical Society
P.O. Box 6248
Providence, RI 02940

The TEX manual can be ordered with a prepayment of $4.40 per copy for individuals, or $8.80 per copy for organizations, from

American Mathematical Society
P.O. Box 1571, Annex Station
Providence, RI 02940

A book containing both the TEX and METAFONT User Manuals is being published jointly by the AMS and Digital Press. Information on the availability of TEX and METAFONT may be obtained by writing to

Dr. Luis Trabb Pardo
Computer Science Department
Stanford University
Stanford, CA 94305

Finally, a detailed discussion of the motivation and background for TEX and METAFONT may be found in Knuth's article "Mathematical Typography", published in the March 1979 Bulletin of the American Mathematical Society.

YACC

The peculiar name YACC stands for "Yet Another Compiler Compiler". It is a program developed by Steve Johnson at Bell Laboratories which, given a language grammar, automatically constructs another program to parse that language. Probably only those readers who have studied compiler writing, or have written command parsers themselves, will appreciate the utility of this. It is a very difficult job to construct by hand a CORRECT parser for an even moderately-complicated grammar. Once lexical analysis and parsing are complete, it is a much easier job to write a code generator for a specific machine, although if one is going to do a good job, or worry about optimization, that too can become complicated.

Many readers might inquire just why a tool such as YACC is included in a roster of software tools which are intended to make scientific software development easier. The response I would give is that a tool such as this might profitably be used to construct a generalized quantum chemistry or crystallographic input routine which could be powerful, correct, and of wide utility. It could also help standardize the multitude of input formats required by such programs.

With a tool such as YACC, one could permit arithmetic expressions as input values, simply by augmenting the input language grammar. In fact, the input language could be made to look like a compiler language with procedure and function calls, conditional statements, and looping constructs, if this proved to be useful.

YACC is distributed as part of the Bell Laboratories UNIX system, with availability as noted in the discussion of STRUCT.

PROPOSAL FOR A STANDARD SET OF PRIMITIVES FOR MACHINE-INDEPENDENT BIT MANIPULATION IN FORTRAN

by

Nelson H.F. Beebe
Departments of Physics and Chemistry
University of Utah
Salt Lake City, UT 84112

Tel: (801) 581-5254

INTRODUCTION

The 1979 NRCC Conference on Software Standards in Chemistry held at the University of Utah resulted in considerable discussion about standard ways of implementing bit manipulation in FORTRAN programs. A previous proposal by the author described in the "Programmer's Guide to Portable Software" [BEEB79a] met with the criticism that the routine names began with the letters BIT, even though some were INTEGER functions. Many programmers apparently prefer to use FORTRAN's default typing conventions based on the initial letter of variable names, even if this reduces mnemonic significance and does not provide for other than REAL and INTEGER data types, unless an IMPLICIT statement is supported by the host compiler. General agreement could apparently be reached if the routine name prefix was changed to IBT, but it should be noted that this still requires variable name typing for those routines which are LOGICAL functions.

Several participants recalled that the Instrument Society of America (ISA), which establishes industrial standards of various sorts, and acts as an advisory body to the American National Standards Institute (ANSI), perhaps had established standard bit primitives for FORTRAN programming. This is indeed the case, and has been published as ISA Standard S61.1 "Industrial Computer System FORTRAN Procedures for Executive Functions, Process Input/Output, and Bit Manipulation".

I feel strongly that ISA S61.1 is not an acceptable standard for the needs of chemistry-related programming. There are several objections which may be noted.

First, names chosen for the bit primitives (IOR, IAND, NOT, IEOR, ISHFT, BTEST, IBSET, and IBCLR) and for obtaining the date and time of day (DATE and TIME) are very likely to conflict with names already implemented by various manufacturers in their extensions to the FORTRAN language. This is not a trivial problem, because some compilers expand intrinsic functions in-line, and overriding them by user-provided routines may not be possible, or may require the use of a special compiler option, or additional declaration statements in the routine which refers to them. The order and interpretation of arguments may differ from machine to machine, and could cause unexpected results which might be difficult to trace.

Second, there is no consistent naming convention for the routines. The use of a mnemonic prefix such as BIT or IBT has proved invaluable in large program systems in that it allows groups of related routines to be readily identified, and also distinguishes the names from local names.

Third, the bits are numbered from right to left starting with zero. There is no agreement among computer manufacturers about whether numbering should be 0123..., 123..., ...321, or ...3210, and the precedence of FORTRAN array subscripts following the order 123... strongly suggests that a left-to-right 123... numbering convention should be adhered to. This is particularly important for the routines which test and set bits, for which an extension to bit arrays contained in more than one word of storage is rather useful.

Fourth, ISA S61.1 does not prescribe a complete interpretation of the arguments. Nothing is said about what the primitive will do when presented with an out-of-range bit number. This is an unsatisfactory situation in a standard, and is bound to cause portability problems.

An alternative set of standard bit manipulation routines is presented in this proposal, and, I believe, remedies all of the above criticisms.

Because the internal format of FORTRAN data types, and also of characters, differs from machine to machine, and indeed, even from compiler to compiler at a single installation, it should be clear that the use of bit primitives to access data immediately involves assumptions which will defeat the goal of portability of chemical software.

It is nevertheless possible to implement certain types of bit manipulation in a machine-independent way, as is illustrated, for example, in the report "Integral File Data Compression" [BEEB79b]. In particular, those applications which require only bit strings can be implemented completely portably, once the number of bits in an integer storage unit is available. I propose for this purpose that the PORT Library Framework [FOX78a, FOX78b] be adopted as a standard facility for accessing machine- and operating-system parameters. This software is in the public domain, and at present, supports various operating systems for computers manufactured by:

* Burroughs (1700/5700/6700/7700 series)
* CDC(6000/7000/Cyber series)
* Cray 1
* Data General (Eclipse S/200)
* DEC (10/20/PDP-11/VAX)
* Harris (Slash 6 and Slash 7)
* Honeywell (600/6000 series)
* IBM (360/370 series)
* Interdata (8/32)
* SEL (System 85/86)
* UNIVAC (1100 series)
* Xerox (Sigma 5/7/9 series)

Parameters for others are very straightforward to add.

The PORT functions I1MACH(5) and I1MACH(9), returning respectively the number of bits in an integer storage unit, and the largest positive integer, are sufficient to allow the data compression routines in [BEEB79b] to convert quantum chemistry integral and matrix element files containing REAL, DOUBLE PRECISION, and INTEGER data into compressed bit strings in which insignificant leading and trailing bits have been removed, offering a data reduction by up to a factor of five or six. Other applications which can be supported by these two PORT functions together with the bit primitives include:

* Manipulation of bit representations of Slater determinants and excitation operators in configuration interaction, perturbation theory, coupled cluster, and propagator calculations, including conversion to and from human-readable integer representations.

* Bit mapping used to represent patterns of non-zero elements in sparse matrices.

* Patterns of dots in a plot to be output on a dot-matrix plotter.

* Compressed LOGICAL arrays using only one bit per element, instead of an entire word.

In those higher-level primitives which deal with variable-length bit strings, rather than single-word bit patterns, the strings are defined in terms of three variables. These are the name of the INTEGER array containing the string, a starting position (numbering 1,2,3,... from the left), and the number of bits to be considered. Thus, an argument sequence STRING,LOC,LEN represents bits LOC, LOC+1, LOC+2, ..., LOC+LEN-1 stored in the array STRING(*). It is an error condition if LOC or LEN is less than 1, and the action to be taken in such a case will be expressly defined for each primitive. In some cases, two strings of the same length are present in the argument list, and the length parameter of the first will then be omitted.

BASIC PRIMITIVES

INTEGER FUNCTION IBTAND (K,L)

Return the logical AND of K and L. This has 1-bits where both K and L have 1-bits, and 0-bits elsewhere.

INTEGER FUNCTION IBTCOM (K)

Return the logical complement of the bit pattern stored in K. The complement is formed by inverting all bits.

LOGICAL FUNCTION IBTEST (STRING,NBIT)

Test bit number NBIT in STRING(*), and return .TRUE. if it is a 1-bit, and .FALSE. if it is a 0-bit. If NBIT is less than 1, return .FALSE..

SUBROUTINE IBTOFF (STRING,NBIT)

Set bit number NBIT in STRING(*) to 0. If NBIT < 1, no bit will be set.

* Implementation of the standard set of character primitives on a given machine.

* Masking operations.

* Construction of byte strings for control of devices such as plotters.

Other functions in the PORT Library Framework which provide such parameters as the machine precision, the base of integer and floating-point numbers, and floating-point exponents may largely obviate the need for bit manipulations which require knowledge of the internal format of such data.

THE BIT PRIMITIVES

A description of the complete set of bit primitives which are proposed here follows. An experienced assembly language programmer should be able to implement them in an afternoon's work on any existing computer presently used for chemical computations. An agency such as the Quantum Chemistry Program Exchange or the NRCC could act as a source of implementations of these routines for a variety of host computers and operating systems, thereby increasing their availability and hopefully encouraging their widespread adoption. It has proven useful at our computing installation to install these in the system FORTRAN library, so that they are automatically available to all FORTRAN programmers. If they are not made convenient to use, many less-motivated programmers will not take the trouble to use them, and the present undesirable situation of machine-dependencies permeating chemical software will only continue.

In all of the bit primitive routines, K, L, and NBIT represent INTEGER variables; STRING(*) is an INTEGER array. Many compilers will generate correct code if STRING is an INTEGER scalar variable, but this use does not conform to Portable FORTRAN. Portability dictates that the arguments to these functions should be restricted to INTEGER data types; use with arguments of REAL, LOGICAL, COMPLEX, DOUBLE PRECISION, or Hollerith (character) data types may introduce machine-dependence, and should be strongly discouraged.

It is considered good programming practice to treat FUNCTION arguments as read-only values, in order to avoid side effects. This convention is adhered to in the definition of all of the FUNCTION primitives.

SUBROUTINE IBTON (STRING,NBIT)

Set bit number NBIT in STRING(*) to 1. If NBIT < 1, no bit will be set.

INTEGER FUNCTION IBTOR (K,L)

Return the logical OR of the bit patterns present in K and L. 1-bits are returned in positions in which either K or L, or both, have a 1-bit.

INTEGER FUNCTION IBTROL (K,NBIT)

Return the bit pattern represented by K after rotating it left by ¶NBIT¶ bits. If NBIT is negative, ignore its sign.

INTEGER FUNCTION IBTROR (K,NBIT)

Return the bit pattern represented by K after rotating it right by ¶NBIT¶ bits. If NBIT is negative, ignore its sign.

INTEGER FUNCTION IBTROT (K,NBIT)

Return the bit pattern represented by K after rotating left (NBIT > 0) or right (NBIT < 0) by NBIT bits.

INTEGER FUNCTION IBTSHF (K,NBIT)

Return the bit pattern represented by K after performing a logical shift left (NBIT > 0) or right (NBIT < 0) by NBIT bits. Bit positions vacated are filled by 0-bits. 1-bits shifted out of the word are lost.

INTEGER FUNCTION IBTSHR (K,NBIT)

Return the bit pattern represented by K after performing a logical shift by ¶NBIT¶ bits to the right.

INTEGER FUNCTION IBTSHL (K,NBIT)

Return the bit pattern represented by K after performing a logical shift by ¶NBIT¶ bits to the left.

INTEGER FUNCTION IBTSUM (K)

Return the number of 1-bits in the bit pattern represented by K.

INTEGER FUNCTION IBTXOR (K,L)

Return the exclusive OR of the bit patterns represented by K and L. The result has 1-bits in positions where K and L have different bits, and 0-bits where K and L have identical bits.

HIGHER-LEVEL PRIMITIVES

SUBROUTINE IBTGET (BYTE,STRING,LOC,LENGTH)

Return in the INTEGER variable BYTE a right-adjusted bit string containing LENGTH bits extracted starting at position LOC in the STRING(*). Any leading bits in BYTE are set to zero. If LENGTH is larger than the word size, only the first I1MACH(5) bits of the selected substring will be returned in BYTE. If either LENGTH or LOC is less than 1, 0 will be returned in BYTE.

SUBROUTINE IBTPUT (BYTE,STRING,LOC,LENGTH)

Store the right-most LENGTH bits of BYTE in the STRING(*), starting at bit position LOC. If either LENGTH or LOC is less than 1, no bits will be stored. If LENGTH is larger than the wordsize, only the first I1MACH(5) bits of BYTE will be stored at the designated position.

INTEGER FUNCTION IBTCMP (STRNGA,LOCA,STRNGB,LOCB,LENGTH)

Given two bit strings define by STRNGA,LOCA,LENGTH and STRNGB,LOCB,LENGTH, compare LENGTH bits of the two substrings, treating the substrings as UNSIGNED binary integers. Return -1, 0, or +1 for the conditions A < B, A = B, or A > B, respectively. If any of LOCA, LOCB, or LENGTH is less than 1, return 0.

SUBROUTINE IBTMOV (TARGET,LOCTAR,SOURCE,LOCSRC,LENGTH)

Given two bit strings defined by TARGET,LOCTAR,LENGTH and SOURCE,LOCSRC,LENGTH, move LENGTH bits from SOURCE into TARGET. If any of LOCTAR, LOCSRC, or LENGTH is less than 1, no bits are moved. Bits must be moved in order from left to right, equivalent to one bit at a time, in order to provide consistent behavior in case TARGET and SOURCE overlap.

SUBROUTINE IBTSWP (STRNGA,LOCA,STRNGB,LOCB,LENGTH)

Given two bit strings defined by STRNGA,LOCA,LENGTH and STRNGB,LOCB,LENGTH, swap the bit strings in memory. If any of LOCA, LOCB, or LENGTH is less than 1, no swap is performed.

CONCLUDING REMARKS

It may be of some cause for concern that the routines containing STRING(*) as an argument do not also contain its dimension, in order that an out-of-bounds storage reference can be avoided in the event of NBIT being too large. We have elected NOT to include this extra argument, because the same criticism may be raised for any FORTRAN array reference for which the compiler does not perform subscript range checking. Even when this feature is available, it is usually suppressed in production programs for reasons of run-time efficiency.

I believe that the above routines provide a satisfactory set of bit primitives whose performance, apart from word-length variations, is the same on all machines, for both valid and invalid arguments. Arithmetic shift routines have been intentionally excluded, because their behavior is inherently connected with the host number representation, and their use would then unnecessarily compromise portability.

Careful programming utilizing I1MACH(5) to provide the number of bits in an INTEGER storage unit can eliminate all word-length dependencies from the executable code, and even from most of the dimensional limitations in the case of the array STRING(*) if its storage is suitably allocated at the top level of a program system. Just as it is becoming more widely understood that fixed dimensions for potential variable-sized arrays of standard data types should be restricted to the MAIN program, so should the dimension of a bit string disguised as an INTEGER array, since the required amount of storage is certain to vary from one computer to another.

Applications will no doubt arise in which it will be convenient to permit Boolean operations to be extended to variable-length bit strings beginning at arbitrary offsets in data words. This presents sufficient extra complications, that a set of higher-level bit routines based on the basic primitives described above would be desirable. Development of such a collection will be left for later work.

REFERENCES

BEEB79a Beebe, N.H.F., "Programmer's Guide to Portable Software", Proceedings of the NRCC Conference of Software Standards in Chemistry (1979).

BEEB79b Beebe, N.H.F., "Integral File Data Compression", Proceedings of the NRCC Conference of Software Standards in Chemistry (1979).

FOX78a Fox, P.A., Hall, A.D., and Schryer, N.L., "The PORT Mathematical Subroutine Library", ACM Transactions on Mathematical Software 4, 109-126 (1978).

FOX78b Fox, P.A., Hall, A.D., and Schryer, N.L., "Algorithm 528. Framework for a Portable Library", ACM Transactions on Mathematical Software 4, 177-188 (1978).

ISA76 "Standard S61.1. Industrial Computer System FORTRAN Procedures for Executive Functions, Process Input/Output, and Bit Manipulation", Instrument Society of America, 400 Stanwix Street, Pittsburgh, PA 15222, February (1976).

PROPOSAL FOR A STANDARD SET OF PRIMITIVES FOR MACHINE-INDEPENDENT CHARACTER MANIPULATION IN FORTRAN

by

Nelson H.F. Beebe
Departments of Physics and Chemistry
University of Utah
Salt Lake City, UT 84112

Tel: (801) 581-5254

INTRODUCTION

At the 1979 NRCC Conference on Software Standards in Chemistry held at the University of Utah, the author proposed that a standard set of primitives for character manipulation in chemical software written in the FORTRAN language be adopted, and set forth a selection of routines which were felt to serve the purpose. The suggested routines followed a uniform naming convention of a mnemonic prefix CHR, to which some opposition was voiced by several participants who felt that since many of the primitives were INTEGER functions, their names should conform to the FORTRAN default by which variable names beginning with the letters I through N are typed as INTEGERs. Although I personally feel that good names are more important than the question of whether an explicit type statement must be inserted, a compromise prefix KAR was suggested and accepted by the working group which considered the issue.

The discussions at the Conference and subsequent communications showed the desirability of adding a few additional primitives to facilitate programming and increase the general utility and efficiency of the character primitives. Several participants also suggested that standards may already exist in the literature for FORTRAN character primitives, and that such standards should be seriously considered for adoption, if they existed. Unlike the case of bit primitives, no such standard seems to have been proposed.

BACKGROUND

Before describing the proposed primitives, some background information is useful. FORTRAN has never offered satisfactory support of character data. Indeed, some compilers extant until the mid-1960's did not even have Hollerith data items or A FORMAT descriptors, or LOGICAL variables, for that matter. When limited character support became widely available in FORTRAN, it was restricted to Hollerith string constants of the form 8HCHEMISTRY, together with the A FORMAT item. Hollerith constants were permitted by the 1966 ANSI FORTRAN Standard to occur only in DATA and FORMAT statements, and as subroutine arguments in CALL statements (but not in FUNCTION references, although no compiler that I am aware of enforces this restriction). No CHARACTER data type was introduced, and characters were forced to masquerade in the guise of other data types.

Coding Hollerith strings is somewhat tedious and error-prone, because of the necessity of counting characters. Consequently, many manufacturers permitted character constants to be surrounded by delimiter characters, for example, "CHEMISTRY", but again, no general agreement was reached about what the delimiter characters ought to be. Single and double quotes are most common, but asterisks and not-equal signs have also been used. When string delimiters are used, the question arises as to how the delimiter character itself is to be represented in a string constant. Usually, the doubled-delimiter approach, "O""MALLEY" for the string O"MALLEY, has been adhered to, although CDC's use of the asterisk as a string delimiter simply prohibited its appearance as a string character. As a result of these variations, only the Hollerith string can be relied upon for portability, and automated means of converting between the different string conventions in FORTRAN source programs are available at some installations.

The 1966 implementation of support for character data is just about the worst possible. The Hollerith form is certainly undesirable. Even worse is the convention for internal storage of character strings. These must always be stored left-justified in a computer word, and right-padded with blanks if the number of characters specified does not fill an integral number of machine words. The number of characters which fit in a word ranges from 1 to 10 on existing computers [BEEB79], and the left-justification means that even if one arranges to store only one character per word for word-length independence, the character will be occupying the most-significant bit positions and probably the sign bit as well. This means that even comparison of characters for equality can result in an arithmetic overflow condition on those machines where comparisons are implemented by subtraction. It also means that accessing the numerical value of a character cannot be done portably, for division by a power of two to effect a right shift of the bit pattern will fail if the sign position is occupied by a 1-bit.

Another problem is that depending upon the FORTRAN type of the variable in which characters are stored, different results may be obtained on different machines. For example, character storage in LOGICAL variables is impossible on those machines which implement LOGICAL scalars and arrays as bit strings, and on most others, the 1966 Standard's prohibition of the use of the relational operators .EQ., .NE., .LT., etc. between LOGICAL variables would prevent character comparisons. Floating-point types are also unsuitable, because mantissa normalization which may occur in assignments or in expression evaluation usually will scramble the bits, destroying the characters stored in the word. This leaves INTEGER variables and arrays as the only possible repository of character data, and even this may fail. On the IBM 7030 Stretch computer, for example, integers are represented internally as floating-point numbers, and unless assembly-language coding is resorted to, it is very inconvenient just to get character data correctly in and out of variables on that machine.

The 1977 FORTRAN Standard has made an attempt to remedy these difficulties by the introduction of a CHARACTER data type, but is still not going to offer a complete solution.

First of all, the Hollerith data type is dropped from the 1977 Standard. This means that a very large body of existing FORTRAN software which uses character data, even in an at-present widely portable fashion, may require extensive changes to run with a FORTRAN 77 compiler, unless manufacturers can be pressed to continue support of character data stored in Hollerith constants and variables. The 1977 standard prohibits all storage equivalencing, either via COMMON and EQUIVALENCE statements, or by FUNCTION or SUBROUTINE argument associations, between CHARACTER data and all other FORTRAN data types. This was necessary to enable FORTRAN 77 to support variable-length character strings, so that declarations of the form

```
SUBROUTINE A (B,C)
CHARACTER B*(*),C(*)*(*)
```

could be permitted, allowing CHARACTER variables to inherit both a size and an array length from a calling program. This forces a compiler to generate code to pass to a called routine the address of a string descriptor containing size and dimension information, as well the actual address of the character data.

Second, standardized library support of character data in the form of useful utility routines is non-existent in the 1977 Standard, apart from the ICHAR and CHAR functions for converting between INTEGER and CHARACTER form.

Third, null character strings, that is, strings of zero length, are not permitted. Null strings are in fact quite useful, and indeed, even necessary in some applications. In particular, a null string cannot be simulated by any string of non-zero length.

Fourth, the 1977 Standard does not specify the character set to be used. The fact that many manufacturers employ their private versions of character sets, each with its own special character repertoire and collating sequence, only continues to perpetrate additional machine dependence upon FORTRAN users.

CRITERIA FOR SATISFACTORY SUPPORT OF CHARACTER DATA

The reaction of some people on reading the above criticisms, or having experienced them personally, will no doubt be to reject FORTRAN completely as a language in which any kind of character manipulations are to be done. There is certainly some validity to this view. However, as one Conference participant remarked, there is really no choice in the matter, for FORTRAN 66 is the only "(almost) machine-independent high-level 'assembly' language" that we have for scientific computation.

FORTRAN is available on essentially all medium- and large-scale computers in the world today, and also on many microcomputers as well. It has been in existence for nearly twenty-five years, and is one of the two or three still-existing original high-level programming languages. It is widely understood by scientists and engineers the world over.

A widely-implemented ANSI and ISO Standard has been in existence for fourteen years, and in fact, FORTRAN was probably the first language to be so standardized.

An enormous amount of FORTRAN software, representing a huge investment of money and programmer years, already exists, and sophisticated and extensive scientific subroutine libraries such as IMSL, Harwell, Boeing, NAG, EISPACK, FUNPACK, and LINPACK are widely available.

FORTRAN's lack of structured control statements, but unfortunately not its limited variety of data types, can be largely avoided by programming in a preprocessor language, such as RATFOR or SFTRAN3, which can then be translated into Portable FORTRAN.

Finally, and importantly, there exist automated tools such as the PFORT Verifier, which can be used to test FORTRAN software for adherence to Portable FORTRAN syntax, grammar, and usage.

In constructing a set of character primitives for widespread implementation on a variety of host machines, two goals must be kept in mind. First of all, the primitives should provide frequently-needed functions. Examples of these include packing and unpacking of characters, obtaining integer equivalents, comparing and moving strings, and letter case and character set conversions. Second, they should permit machine-independent implementation of programs which manipulate character data.

The second goal carries with it an important decision. This is that a standard character set must be adopted, or at least be available via function calls, in order that such operations as sorting by collating sequence, or the use of integer equivalents of characters for governing the flow of control in programs such as parsers and lexical analyzers, can be implemented in a fashion which will guarantee that the same results will be obtained, independent of the host computer.

There is fortunately at present an internationally-agreed-upon character set, known as ASCII (American National Code for Information Interchange), defined in ANSI Standard X3.4-1968 and revised in X3.4-1977. It has been adopted in Japan as the Japanese Industrial Standard Code for Information Interchange (JISCII) (1969), and by the International Standards Organization as ISO DR 1052 (1967). Unfortunately, at present the American "Big 3" computer manufacturers IBM, CDC, and UNIVAC do not provide wide support for ASCII, although both UNIVAC and CDC are evidently moving in that direction.

ASCII is a 7-bit code offering 2**7 or 128 different characters, made up of 32 standard control characters, followed by a space, then the special characters !"#$%&'()*+,-./, the digits 0-9, the special characters :;<=>?@, upper-case letters A-Z, special characters [®]©_°, lower-case letters a-z, special characters §¶†™, and finally, a DELete control character. With the exception of the DELete control character, the special characters following the letters may be replaced with national characters for those alphabets having more than 26 letters. Standardization work is going on at present to expand the code to 8 bits, and Cyrillic and Japanese Katakana characters have already been assigned to characters in the range 128-255 for use in the Soviet Union and Japan.

This proposal recommends the adoption of the ASCII character set as a standard one, and functions are defined allowing access to it even on those computers which do not yet use it. It is worth noting in passing that the new U.S. Department of Defense programming language, ADA [SIGP79], has specified that all character data shall be in the ASCII character set, independent of the host computer.

THE CHARACTER PRIMITIVES

The character primitives defined in the remainder of this proposal can all be implemented entirely in FORTRAN if a standard set of bit primitives is available. However, because of the differing storage order on some machines such as the PDP-11 and the DEC VAX 11/780, where characters are stored in reverse order, a FORTRAN implementation will in general not be portable, even if such parameters as the number of bits in a character, and the number of characters in an INTEGER storage unit, are available machine-independently through the PORT Library Framework [FOX78a, FOX78b]. However, an initial FORTRAN implementation in terms of bit primitives may nevertheless be useful as a bootstrapping process when software is to be installed on a new machine. All of the routines will be straightforward to implement in assembly language, and particularly for those machines which support character addressing in hardware, it may be an order of magnitude more efficient to do so.

Just as in the case of the proposed bit primitives, it is anticipated that bodies such as the Quantum Chemistry Program Exchange or the NRCC could act as a source of implementations of these primitives for a wide variety of host computers. Installations will also find that programmers are more easily encouraged to use the standard character primitives if they are conveniently available, preferably as part of the local system FORTRAN library.

In the following descriptions, all arguments are scalar INTEGER variables, except TEXT(*), which represents either a Hollerith constant, or character data packed with the maximum number of characters per word. Exceptions to this will be noted when necessary. Readers familiar with the programming languages PASCAL and PL/1 will note their influences on the design of these routines.

The character primitives will be divided into two classes -- basic routines, and higher-level routines. The latter can be implemented in FORTRAN in terms of the former, although on some systems with advanced hardware facilities, it may be desirable to define them directly in assembly language.

In developing any software system, a decision must always be made about how error conditions are to be handled. In a set of routines which are proposed for adoption as a Standard, it is clearly unacceptable to ignore errors, and it is equally unsatisfactory to define behavior under error conditions to be "undefined", for this simply means that the action to be taken is decided by the implementor.

Only two acceptable alternatives exist. Either an error flag can be returned, or predefined reasonable action can be taken when errors arise. The first of these places the burden of error handling on the user of the software, and frequently results in error conditions simply being ignored, or perhaps handled incorrectly. The second alternative simplifies programming on the part of the user by moving the error processing to a lower level, and also guarantees consistent error handling in all implementations. For this reason, the second of these has been adopted for the character primitives.

An axiom of good programming is that functions should not have side effects. In practical terms, this usually means that they should not modify their arguments, or variables globally accessible through COMMON storage or its equivalent. This convention has been adhered to in the definition of the FUNCTION character primitives.

In those primitives which deal with character strings, rather than single characters, the strings are defined in terms of three variables. These are the name of the INTEGER array containing the string, a starting position (numbering 1,2,3,... from the left), and the number of characters to be considered, counting from the starting position. Thus, an argument sequence TEXT,LOC,LEN represents characters LOC, LOC+1, LOC+2, ..., LOC+LEN-1 stored in the array TEXT(*). It is an error condition if either LOC or LEN is less than 1, and the action to be taken will be expressly defined for each primitive. In some cases,

two strings of the same length are present in the argument list, and the length parameter for the first will then be omitted. In most applications, the LOC parameter will point to the first character in the array; its presence is, however, necessary to allow access to strings which do not begin at a word boundary.

BASIC PRIMITIVES

SUBROUTINE KARGET (CHAR, TEXT, LOC)

Extract a single Hollerith character from the packed Hollerith string stored in the INTEGER array TEXT(*), taking it from position LOC (numbered 1,2,3,... from the left), and return it in the INTEGER variable CHAR in A1 format. If LOC < 1, return the value +0 in CHAR. This value is not equal to any A1 format character on any existing computer.

SUBROUTINE KARPUT (CHAR, TEXT, LOC)

Store a single Hollerith character, CHAR, into the packed Hollerith string stored in the INTEGER array TEXT(*), placing it in position LOC. If LOC < 1, no character is stored.

INTEGER FUNCTION KARORD (CHAR)

Return the ordinal position of the single Hollerith character CHAR in the installation-dependent character set. The bit pattern corresponding to the left-most character in CHAR is used to determine the ordinal value, and all trailing bits are ignored. No error condition is therefore possible.

INTEGER FUNCTION KARCHR (ORD)

Return the single Hollerith character corresponding to the INTEGER ordinal position ORD in the installation-dependent character set. The right-most bits in ORD are used to construct the Hollerith character, and all remaining bits of ORD are ignored, so that no error condition exists if ORD is out of the valid range.

INTEGER FUNCTION KARLC (CHAR)

Return the lower-case equivalent of the single Hollerith character CHAR. Only the letters "A" - "Z" will be affected by this function. If the host computer does not support lower-case letters, then KARLC must return CHAR as its value.

INTEGER FUNCTION KARUC (CHAR)

Return the upper-case equivalent of the single Hollerith character CHAR. Only the letters "a" - "z" will be affected by this function. If CHAR is not a lower-case letter, then the function value returned must be CHAR.

INTEGER FUNCTION KARASC (CHAR)

Return the ordinal position of the single Hollerith character CHAR in the ASCII character set. For ASCII machines, this function is identical to KARORD, but for non-ASCII machines will be different. If CHAR has no ASCII graphic equivalent, then the value -1 must be returned.

INTEGER FUNCTION KARLCL (ORD)

Return the local Hollerith character corresponding to the ASCII character having ordinal value ORD. If ORD corresponds to a lower-case letter, and the local character set supports only upper-case letters, then KARLCL must return the upper-case equivalent of ORD. KARLCL is the inverse of KARASC, since KARLCL(KARASC(CHAR)) = CHAR for characters for which equivalences can be defined. The right-most bits in ORD are used to construct the Hollerith character, and all remaining bits of ORD are ignored, so that no error condition exists if ORD is out of the valid range. If the ASCII character corresponding to ORD has no equivalent in the local character set, then KARLCL must return a value +0; this cannot represent an A1 format character on any existing computer.

HIGHER-LEVEL PRIMITIVES

SUBROUTINE KARMOV (TARGET,LOCTAR,SOURCE,LOCSRC,LENGTH)

Given two character strings defined by TARGET,LOCTAR,LENGTH and SOURCE,LOCSRC,LENGTH, move LENGTH characters from SOURCE(*) into TARGET(*). If any of LOCSRC, LOCTAR, or LENGTH is less than 1, then no characters are moved. The move must be performed in order from left to right in order to provide uniform behavior in the event of overlapping character strings.

SUBROUTINE KARSWP (TEXTA,LOCA,TEXTB,LOCB,LENGTH)

Given two character strings defined by TEXTA,LOCA,LENGTH and TEXTB,LOCB,LENGTH, swap them in memory. If any of LOCA, LOCB, or LENGTH is less than 1, no action is taken.

INTEGER FUNCTION KARCMP (TEXTA,LOCA,TEXTB,LOCB,LENGTH)

Given two packed character strings, TEXTA(*) and TEXTB(*), beginning at character positions LOCA in TEXTA(*) and LOCB in TEXTB(*), compare the next LENGTH characters of the two strings, and return -1, 0, or +1 according to A < B, A = B, or A > B, respectively. The collating sequence that MUST be used is that defined by the ASCII character set. The overhead for this is minimal, and often will require no more than an indexed register load, rather than a direct load, in an assembly language implementation. Thus

```
      I = KARCMP(1H ,1,1HA,1,1)
```

will ALWAYS give I = -1, even on a machine such as a CDC computer which uses an internal representation in which a space is greater than the letter A. Note that upper- and lower-case letters are NOT equivalent with this function. If any of LOCTAR, LOCSRC, or LENGTH is less than 1, then KARCMP = 0 on return. This result is chosen to comply with the interpretation that invalid strings are null strings, and all null strings are equivalent.

```
      INTEGER FUNCTION KARCM2 (TEXTA,LOCA,TEXTB,LOCB,LENGTH)
```

This function is similar to KARCMP, except that the character comparison is done without regard to letter case for the letters A-Z and a-z. It is provided because of its wide utility in, for example, testing responses entered from interactive terminals which support lower-case letters.

```
      INTEGER FUNCTION KARIDX (TEXTA,LOCA,LENA,
     #                         TEXTB,LOCB,LENB)
```

Given two packed character strings defined by the arguments TEXTA,LOCA,LENA and TEXTB,LOCB,LENB, search the first string for the first occurrence of the second string. If any of the arguments LOCA, LENA, LOCB, or LENB are invalid, or if the second string is not found in the first, return 0. Otherwise, return its index, or position, in TEXTA(*), counting from the first character stored in TEXTA(*). The result returned is thus always 0, or not less than LOCA. For example, KARIDX(5HHELLO,LOC,5,2HLO,1,2) returns the value 4 if LOC = 1,2,3, or 4, since the second string begins at the fourth position in the first string. If LOC is outside the range 1..4, then 0 is returned.

KARIDX is essentially equivalent to the PL/1 INDEX function. It is worth noting that at least two computers, the DEC VAX-11/780 and the UNIVAC 1160, have a single hardware instruction for performing the function of this routine.

```
      SUBROUTINE KARXLT (TEXT,LOCTXT,LENTXT,
     #                   OLD,LOCOLD, NEW,LOCNEW, LEN)
```

Given a string defined by TEXT,LOCTXT,LENTXT, translate characters according to the characters stored in each of the packed strings defined by OLD,LOCOLD,LEN and NEW,LOCNEW,LEN. Each character occurring in OLD(*) has a corresponding character in NEW(*). For example, the statement

```
      CALL KARXLT (TEXT,1,LENTXT, 2H<>,1, 2H(),1, 2)
```

will result in each occurrence of < in TEXT(*) being translated to (, and each > to).

The characters in OLD(*) should be unique. In case they are not, the translation MUST be according to the last occurrence of a duplicated character. That is, if OLD(*) contains 3HBAB and NEW(*) contains 3HXYZ, then A's in the substring will be translated to Y's, and B's to Z's. This restriction facilitates implementation of the translation with an internal lookup table constructed from the standard character set with changes according to substitutions of OLD characters with NEW characters performed in order from left to right.

If any of LOCTXT, LENTXT, LOCOLD, LOCNEW, or LEN is less than 1, return occurs immediately with no modification of TEXT(*).

```
      INTEGER FUNCTION KARVFY (TEXT,LOCTXT,LENTXT,
     #                         PATERN,LOCPAT,LENPAT)
```

Search the string defined by TEXT,LOCTXT,LENTXT for the first character which is NOT contained in the pattern string defined by PATERN,LOCPAT,LENPAT. If any of LOCTXT, LENTXT, LOCPAT, or LENPAT is less than 1, return 0. If all characters in the first string are found in PATERN(*), return 0. Otherwise return the index of the first mismatching character in TEXT(*), counting from the first character stored in TEXT(*). Thus KARVFY always returns a value 0, or one which is not less than LOCTXT. It is important to note that the index returned points to the first string, not to the pattern string.

KARVFY is based upon the PL/1 VERIFY function. In PASCAL, its would be implemented by testing a character to see if it belongs to the set of characters forming the pattern. For example, if the pattern contained letters, digits, and a space, then KARVFY would return the index of the first character in TEXT(*) which was not a letter, digit, or space. This is a rather convenient function to have for implementing command parsers.

```
      SUBROUTINE KARUPK (TARGET,SOURCE,LOCSRC,LENGTH)
```

Unpack LENGTH characters from SOURCE(*) into A1 FORMAT in TARGET(*), beginning at character position LOCSRC in SOURCE(*). If either LOCSRC or LENGTH is less than one, then no characters are unpacked.

SUBROUTINE KARPAK (TARGET,LOCTAR,SOURCE,LENGTH)

Pack LENGTH characters stored in A1 FORMAT in SOURCE(*) into TARGET(*), beginning at character position LOCTAR in TARGET(*). If either LOCTAR or LENGTH is less than one, no characters are packed.

REFERENCES

BEEB79 Beebe, N.H.F., "Programmer's Guide to Portable Software", Proceedings of the NRCC Conference of Software Standards in Chemistry (1979).

FOX78a Fox, P.A., Hall, A.D., and Schryer, N.L., "The PORT Mathematical Subroutine Library", ACM Transactions on Mathematical Software 4, 109-126 (1978).

FOX78b Fox, P.A., Hall, A.D., and Schryer, N.L., "Algorithm 528. Framework for a Portable Library", ACM Transactions on Mathematical Software 4, 177-188 (1978).

SIGP79 "Report on the ADA Language", ACM SIGPLAN Notices 14, (No. 6A, 6B, June) (1979).

OUTLINE FOR A MACHINE-INDEPENDENT RANDOM-ACCESS INPUT/OUTPUT INTERFACE

by

Members of the NRCC Conference on
Software Standards in Chemistry
Working Group Subcommittee
(in alphabetical order)

Nelson H.F. Beebe
J. Stephen Binkley
David J. Duchamp
Stephen T. Elbert
A. Douglas McLean
George D. Purvis, III
Richard C. Raffenetti

****
* Abstract *
****

A proposed set of FORTRAN-callable routines which implement a standard machine-independent interface for random-access I/O is presented. Although the internal structure of the interface routines will vary from machine to machine, the calling sequences will be constant, and devoid of machine-specific parameters. Actual implementations of the interface routines will be provided to the NRCC and the Quantum Chemistry Program Exchange by members of the Working Group Subcommittee for those host operating systems to which they have access. It is hoped that other members of the computational community will be similarly willing to provide implementations developed for different computer systems.

****
* Background *
****

The 1966 FORTRAN Standard did not include facilities for random-access input/output in the language. In retrospect, this has been one of the most serious shortcomings of the Standard, and also one in which manufacturers have offered the greatest diversity of implementations. There do not appear to be even two independent computer manufacturers who support an identical implementation of random-access I/O. A recurring barrier to quantum chemical program portability has been random-access I/O, and it is the consensus of the Subcommittee that a Standard Random-Access I/O Interface can indeed be defined and implemented without regard on the part of the user to the vagaries of the host operating systems.

A random-access storage device may be any physical device permitting direct retrieval of individual data records without an overhead which depends on the number of preceding records in the file. Examples of such devices are magnetic disks and drums, bubble memory, slow core, and film store. Inherently sequential devices such as reels of magnetic tape are not considered random-access devices, even though some manufacturers have provided facilities which simulate this (e.g. DECTAPE).

The authors of this report are primarily concerned with the specification of a random-access I/O interface which is suitable for quantum chemical applications. However, it should be evident that the design is general enough that the interface could certainly receive wide use in other areas of computational endeavor as well. Random-access I/O is essential in certain computations, and an example may help illustrate why this is so.

The primary application in quantum chemistry is the out-of-core sorting of a very large randomly-sparse matrix into row or column order. This matrix is most frequently a "two-electron integral" matrix, and is indexed by four integers (p,q¶r,s), (p,q¶ forming a row index and ¶r,s) a column index. Each index lies in the range 1..N, where N is the number of basis functions included in the calculation. This is typically of the order of 50, although problems with N = 100 to 150 may be tackled if resources permit. There are thus about N**4 numbers to sort, and few computers currently available permit an address space sufficiently large to allow direct memory access to individual elements of such an array. Even if such central memory sizes were widely available, storage economization possible due to the sparseness would usually discourage keeping the full matrix in memory.

At certain stages of a computation, it may be necessary to produce a new matrix with elements (a,b¶c,d), where a, b, c, and d represent linear combinations of the original basis functions. The individual elements (p,q¶r,s) are themselves sufficiently difficult to compute, so that it is not practical to generate the (a,b¶c,d) integrals directly in most cases. Instead, a four-index transformation from (p,q¶r,s) to (a,b¶c,d) is carried out. If only sequential I/O facilities are available, this very large matrix multiplication requires the I/O transfer of the order of N**5 elements, whereas with random-access I/O, only of the order of N**4 need be transferred between central memory and external storage.

If a sufficient number of I/O units are available, it is possible to use sequential sorting algorithms such as the polyphase sort to reduce the I/O transfer to the order of N**4, but this has not usually been exploited. Which sorting algorithm proves the fastest depends to a great extent on the randomness of the data to be sorted, and also on characteristics of file organization and device access times. However, there are cases in which the sorted data must be retrieved in a particular order which is not known until execution time, and in such cases, a random-access sort is essential in order to allow later selective retrieval of the data.

****************** * Design Issues * ******************

Two overriding factors have influenced the Subcommittee's design of the interface. The first of these is that it must be implementable on ALL systems which are currently in wide use for quantum chemistry computations, and that cognizance must therefore be taken of the diversity in machine architecture and file system design. The second of these is that support for two primary data types must be provided. The first is the usual FORTRAN (fullword) INTEGER type, and the second is what will be termed WORKING PRECISION. On those computers offering a large single-precision mantissa (perhaps 40 or more bits, or about 12 decimal figures), this may be implemented as FORTRAN REAL type. On those with a smaller single-precision mantissa, it will normally be DOUBLE PRECISION type. The use of both single and double precision in the same calculation is rare, and consequently WORKING PRECISION will in fact be a fixed type on a particular host computer. Quantum chemistry software is often widely shared, and any given program may be expected to run on perhaps dozens of host machines, sometimes in single precision, and sometimes in double precision. In order to reduce the modifications necessary when programs are moved from one machine to another, the interface routines distinguish only between INTEGER and WORKING PRECISION. We wish to avoid the SQRT/DSQRT/QSQRT problems that normally plague FORTRAN software due to FORTRAN's lack of generic function names.

A consideration which is related to the first of the above is the choice of names for the interface routines.

It is very important to avoid collisions with names already preempted by the host operating system, or the user, for that matter. Experience with large program libraries, such as the Harwell Subroutine Library, the NAG library, the IMSL library, PLOT76, LINPACK, and others, has amply demonstrated the desirability of systematic names, chosen usually from a leading two- or three-letter prefix denoting a general routine class, followed by letters indicating specific functions or perhaps representing some mnemonic phrase describing the routine. For example, in the IMSL library, ZRPOLY is a routine for finding zeroes of real polynomials, and the library includes about a dozen other routines with the prefix ZRP which are used internally. In PLOT76, VISNH is a routine in the VISibility class which establishes a New Horizon.

Prefixes such as RA (for Random Access) or DA (for Direct Access) immediately suggest themselves for our interface design. These have been rejected in favor of the prefix IR, in order that certain of the routines can be implemented as FORTRAN INTEGER FUNCTIONs without requiring the programmer to explicitly declare their types. This deviates from our policy of generic naming, but it was felt that the use of default variable typing by FORTRAN programmers was too well-established to go against at this late stage. A characteristic prefix is preferable to a suffix (as in CDC's OPENMS, WRITMS, READMS, and CLOSMS) in that file directory listings, loader map listings, and routine cross-reference maps will tend to keep the names of the interface routines close together, reinforcing their logically-close relationship.

In order to conceal unnecessary detail from the user, the interfaces will generally require common blocks in which to share information. For this reason, names of the form IRCBnn, where nn represents a two-digit string, will be reserved for common block names. Documentation for a particular implementation will make available to the user the lengths of any such common blocks, in order that dummy common blocks of the same name may be declared in the MAIN program if this is required by the host operating system. The actual contents of the common blocks are not relevant to the user, and for this reason, user documentation will not usually define them. A system installation document describing the details of a particular implementation should, however, be provided, even though its availability to users not involved in the system installation may be limited. In particular, a system implementor has the right to change the contents of any such internal common areas or interface routine program logic at any time, provided that the external appearance and operation of the interface is not changed.

The design of I/O systems in most modern programming languages and operating systems customarily introduces four distinct tasks: OPEN, READ, WRITE, and CLOSE.

The first of these provides for any initialization required, as well as establishing the logical connection between a file on a storage device and the file designator used within the program to refer to the file. This designator normally takes the form of a short string of characters, or a small integer. In the FORTRAN tradition, an integer in the range 1..99 will be assumed by the interface. For portability, programmers should normally use a restricted range of 1..20, and should ALWAYS represent the integer by a symbolic name, since some operating systems permanently associate specific device types with certain FORTRAN unit numbers. A FORTRAN unit number may be associated with one and only one file during the course of a job. This restriction is already enforced by many operating systems, and the Standard Interface will not depart from it. However, no checking will be done by the interface to ensure adherence to this requirement.

The OPEN function must be able to obtain the actual name of the file on the storage device from the operating system, if this name is required. The file name will not be required by the user's program, nor will it be made available to the user via the interface, because the representation of file names varies so drastically between host operating systems. This implies that some means of establishing a connection between a particular file and its unit number must be available outside the FORTRAN environment. In most operating systems, this facility is provided in the job control language. In at least one, the DEC TOPS-20 operating system, this is not possible if other than predefined file names are to be used. Such an implementation may then have to provide a means for the user to define the unit number <--> file name connection, but this will not be part of the Standard Interface. Because many operating systems require that the OPEN function provide information such as file size and record size, use of an explicit OPEN call shall be MANDATORY. The READ/WRITE routines will return an error indicator if the file is not open, and will not attempt an open internally, as FORTRAN normally does for sequential I/O. A file disposition of KEEP or DELETE must be provided in the OPEN call, by analogy with the 1977 FORTRAN Standard. Its primary purpose is to provide information to the operating system of what to do with the file if the job aborts before a CLOSE call (see below) can be issued.

The READ and WRITE routines pose a few problems. First is the question of how the records are to be identified. This will be discussed at some length in the next section. The second is what type of data may be transmitted. Since the interface operates via subroutine call, the generality of an I/O list on a sequential READ or WRITE statement is not available. After lengthy discussion, it was decided that each READ and WRITE routine shall receive ONE data array and a corresponding length indicator. Separate routines will be provided for INTEGER and WORKING PRECISION arrays, and the lengths will be measured in INTEGER and WORKING PRECISION storage units respectively. The length parameter will be recorded in the record identifier and a function, IRRECL, will be provided to extract it. On a READ operation, the array length will be returned. It need not be known when the READ is initiated, but it is the user's responsibility to ensure that the argument array has sufficient space to contain the returned data.

On some machines, I/O can proceed asynchronously as well as synchronously. With asynchronous I/O, the operating system returns control to the initiator of an I/O request before completion of the operation. The initiator may then carry on with other processing, but the user must ensure that the contents of the storage locations involved in the I/O transfer are not altered until the completion of the I/O operation. A WAIT function is provided which can be invoked when processing has reached a point where the data areas are required again. It will suspend the initiating process until the completion of the I/O transfer. A CHECK function is provided to test for completion without causing a wait. This is useful if multiple buffers are used, because one can arrange to loop, testing the status of each of the buffers in turn until one is finally found for which no I/O operation is in progress; this becomes the next buffer to use.

When the asynchronous I/O facility is available, clever programming can sometimes be used to efficiently overlap processing with I/O and reduce the overall residence time of the job. This is probably less important on multi-user machines, since waits on I/O usually do not waste CPU time, but simply make it available to other users. However, on single-user machines, which are becoming more and more common with the falling costs of computing systems, it is an important enough consideration to warrant inclusion in the Standard Interface. In fact, since synchronous I/O, which never returns control until completion, is a special case of asynchronous I/O, it has been decided to implement it in terms of the asynchronous READ, WRITE, and WAIT primitives. Thus, programmers who do not wish to take the trouble to implement asynchronous processing need not do so, but can nevertheless utilize Standard Interface routines without the burden of issuing calls to the WAIT routine themselves.

The last of the four basic I/O tasks is the CLOSE operation which has the responsibility of ensuring that any outstanding I/O operations are completed, including proper emptying of any buffers. In keeping with the 1977 FORTRAN Standard, the user may provide a disposition parameter to the CLOSE function, which indicates whether the file is to be kept or deleted. If the file is to be kept, it may be accessed again in the same job, after a new OPEN call is issued. If it is deleted, it will no longer be available to the job, and the host operating system may in fact release the file storage space for use by other jobs. This release of space may occur dynamically, which is desirable, or it may be delayed until the termination of the job. The latter will usually be the case in IBM implementations, for example. In either case, once a CLOSE with the delete option has been issued, the file contents are forever inaccessible to the program. A disposition parameter specified for the CLOSE operation OVERRIDES any disposition parameter provided in the OPEN operation referring to the same file. Like the OPEN operation, a CLOSE operation is mandatory, and an attempt to issue an OPEN for a file which is already open will return an error indicator. Since it is unlikely that the interface routines will have access to the operating system interrupt handler, it will normally not be possible for an interface routine to gain control after an interrupt to ensure that any open random I/O files are properly tidied up. This is one area where use of the local random-access I/O facilities might provide a greater degree of security, since many FORTRAN systems do provide post-interrupt cleanup operations. However, since any interrupt which occurs while a file is open and I/O is in progress potentially can compromise the integrity of the file, we do not regard this deficiency as a serious one.

An additional primitive which may be useful for both synchronous and asynchronous I/O is a FIND operation which initiates positioning of the storage device to the record which is intended to be retrieved next. The FIND operation usually will not transfer any data into an internal buffer, although some implementations may choose to do so. For other implementations, it may in fact be a null operation. It will generally be useful only when the issuing job is the principal user of the storage device. If this is not the case, the chances are that another user will cause repositioning for another I/O request, and the FIND operation may then even cause wasted repositionings. A primitive implementing the FIND operation is provided as part of the Standard Interface.

```
*************************
* Record Identification *
*************************
```

Records in a random-access file must be identified in some manner which will permit retrieval of a selected record without having to read through other records to find it. For example, in a personnel file, records might be uniquely identified by a character string, or key, consisting of an employee's social security number. Since the number of such keys is very large, it is usually necessary to either maintain a multi-level index, such as a B-tree, or else to use a hashing algorithm to reduce a large key to a small integer which is used as an index into a hash table which eventually locates the record address in a chained list. Either of these techniques is satisfactory and apply quite generally to a file with alphanumeric keys and variable-length records.

If records are of fixed length, and if the file space is allocated in a contiguous block on the storage device, then it is possible by a simple calculation to determine the device address of, say, the k-th record, and thereby retrieve it directly without any index search. This simple scheme is all that many manufacturers have provided in their FORTRAN implementations, and is the only method provided for in the 1977 Standard. It is unfortunately rather restrictive, both from the point of view of fixed-length records, and from the limitation to sequential integer keys.

Variable-length records are highly desirable for two reasons. First, in the quantum chemical application of random-access I/O to ordering of sparse matrices, it is highly undesirable to fill in the zeroes just to get rows of equal length. Second, even if the matrix is dense, data compression which eliminates non-significant bits from individual items will always produce variable-length data arrays. The use of data compression is of great practical importance, since experience has shown that reductions by factors of 4 to 6 are possible in the total data file length. In these large computations, total execution time may be proportional to the size of the data file, so costs can be reduced substantially. The designers of FORTRAN understood even in the middle 1950's the importance of letting the programmer work in terms of arbitrary-length logical records defined by the I/O list on the READ and WRITE statements. Any blocking of records to fixed length should occur inside the I/O system, completely transparently to the programmer.

Choice of a suitable record key which allows a high degree of machine independence is a matter which received intense debate in the Subcommittee. It was finally decided that the record key should be a quantity which is provided by the interface, rather than by the user. When a record is written to the file, a record identifier will be returned to the user in a two-element INTEGER array. It will contain information defining the record key, as well as the record length and data type. The format of this data stored in the record identifier is implementation-dependent, and functions are therefore provided to allow the programmer to obtain the length and type fields. The actual record key is not interpretable by the user, since it is embedded in the record identifier. A function is provided to allow testing two identifiers for equality. The key may be a simple integer, but it can also be a track and record address (e.g. IBM's TTR), a word address, or a byte address. The record length itself will in most implementations not be placed on the file by the interface, so it is the user's responsibility to save the record identifiers. It may often be convenient to reserve one or more records of the file for this purpose.

Although the internal format of a record identifier is implementation-dependent, an identifier consisting of two positive INTEGER zeroes shall be regarded as an invalid one, and may therefore be used, for example, as a null pointer in a linked list of record identifiers. An implementation will ensure that such a null record identifier is never returned to the user.

This method at times will be inconvenient, but it was felt that a more sophisticated indexing algorithm could always be constructed using the primitives that we have defined. It is our intention that these higher-level routines will be developed later, and could then be included in the Standard Interface. As a start in this direction, a set of higher-level routines has been defined. These are assigned the prefix KR and use record sequence numbers in place of record identifiers. These will be implemented in terms of the lower-level primitives, and are described in detail later.

```
*********************************
* Random-Access I/O Primitives *
*********************************
```

In this section, the primitives which implement the Standard Interface will be defined. All arguments and function values are of type INTEGER, with the exception of WPVEC(*), which is a vector of type WORKING PRECISION and length LENWP measured in WORKING PRECISION storage units. INTVEC(*) is a vector of type INTEGER and length LENINT measured in INTEGER storage units. It is permissible for LENINT and LENWP to take on zero or negative values; this will simply cause a NULL record to be read or written. A null record may or may not be recorded on the file, but it shall have a record identifier which does not correspond to any non-null record on the file. A negative length will be recorded as a zero length. A two-element INTEGER array will contain the record identifier. It will be called IDIN(*) and IDOUT(*) in the argument lists to emphasize whether it is an input parameter or an output parameter.

Error codes are uniform for all routines and all implementations, and consequently, no routine will be able to set every possible error code. Some implementations may not find it necessary to be able to issue each error code. A zero value for an error code indicates that no error condition exists. Negative error codes are not permitted. To assist in analysis of error codes, a primitive is provided which can write an informative message on a user-specified file.

Some implementations of the interface may require that certain flags be set prior to beginning execution. This can be done in a portable fashion in only two ways. Either a BLOCK DATA routine can be defined to initialize variables in common, or an explicit initialization routine can set common variables by direct assignment. The BLOCK DATA route is rejected because of the difficulties of getting the common blocks to be automatically loaded by the system loader or linker at run time, and because the limitation to one BLOCK DATA routine would require the user to know the contents of the interface common blocks. For this reason, a standard initialization routine, IRINIT, is provided which should be called only once, usually at the start of the user's MAIN program. Results are undefined if IRINIT is called more than once.

Initialization

Initialize the Standard Random-Access I/O Interface. This call should be made only once during a given job execution.

```
      CALL IRINIT
```

Open a Random-Access File

Open a file for random-access processing. This must be done before any operations are performed on the file. It is an error if the file is already open. However, since no error code causes the job to be aborted, the user has complete control over any fixup action to be taken. The record size and record type parameters are included because some implementations may be able to provide substantially-improved performance if this information is available to them. However, the values of recsize and rectype are only guidelines to the interface, and the interface may choose different values if this is appropriate. The recsize and rectype arguments are not modified, however. It is NOT an error to specify records of one type or size at OPEN, and then to actually read or write records of a different type or size.

```
      CALL IROPEN (fileid,filesize,extendsize,disposition,
     #             recsize,rectype,errorcode)
```

fileid..........	FORTRAN unit number in range 1..99.
filesize........	Estimated file size in WORKING PRECISION storage units.
extendsize......	Number of WORKING PRECISION storage units to extend the size of the file by if filesize is reached. The effect of this parameter may not be attainable in all implementations.
disposition.....	File disposition in case of termination before a CLOSE call can be issued. Specify 0 for KEEP, 1 for DELETE, or 2 for SCRATCH.
recsize.........	Estimated average record size in storage units of type determined by rectype. A value which is not in an acceptable range will cause this option to be ignored.

rectype.........	INTEGER flag defining type of storage units and record. 1 ==> WORKING-PRECISION variable-length records. 2 ==> WORKING-PRECISION fixed-length records. 3 ==> INTEGER variable-length records. 4 ==> INTEGER fixed-length records. Any value outside the range 1..4 will cause this option and the recsize option to be ignored.
errorcode.......	Output return code (see table below).

The disposition code deserves some elaboration. A file opened with KEEP specified must be on a device which permits the file to be retained after the job terminates. DELETE might often be specified in the OPEN with KEEP requested when a CLOSE is issued, so that an unsuccessful job would simply cause the file to be discarded. SCRATCH implies DELETE, and CANNOT be overridden by a disposition of KEEP on the CLOSE call. This restriction is enforced because many installations have certain storage devices reserved exclusively for use during job execution. At some installations, it may frequently be the case that significantly more file space is available on SCRATCH devices than on those permitting files to be retained after job termination.

Close a Random-Access File

Close a file after random-access processing. This call must be issued before job termination if the file is to be usable in a subsequent job. It should generally be issued as soon as random-access processing of the file is complete to ensure integrity of the file in the event that the job later terminates unexpectedly.

```
      CALL IRCLOS (fileid,disposition,errorcode)
```

Asynchronous Write of a Record

Write a record asynchronously. The data areas involved in the transfer must not be modified until a WAIT has been issued to ensure completion of the operation. Note that letters 3 and 4 (WR) indicate the operation (WRite), letter 5 indicates the data type (W - WORKING PRECISION, I - INTEGER), and letter 6 (A) indicates that the operation is Asynchronous. The record will be written at the end-of-information on the file, and the record identifier returned to the caller.

```
CALL IRWRWA (fileid,IDOUT,WPVEC,LENWP,errorcode)

CALL IRWRIA (fileid,IDOUT,INTVEC,LENINT,errorcode)

fileid..........FORTRAN unit number in range 1..99.
IDOUT(*)........2-word  INTEGER  record  identifier
                returned to the caller.
WPVEC(*)........WORKING-PRECISION array to be written.
LENWP...........Number of elements in WPVEC(*).
INTVEC(*).......INTEGER array to be written.
LENINT..........Number of elements in INTVEC(*).
errorcode.......INTEGER error  code returned to caller
                (see below).
```

Asynchronous Update of a Record

Rewrite an existing record asynchronously. The new record will be written in place of the record identified by IDIN(*) if space permits, or will be added at end-of-information. IDOUT(*) will be set to the address of the replacement record. The interface need not provide for garbage collection of vacant records, although it may do so. Vacant records can arise only when an update replaces a record by a larger one. If this is done frequently, large holes of unused space may arise in the file. Programmers who contemplate maintaining random-access files over an extended time period with frequent updating should consider developing a utility which can be used to copy the file to a new file with consequent deletion of vacant records.

```
CALL IRUPWA (fileid,IDIN,IDOUT,WPVEC,LENWP,errorcode)

CALL IRUPIA (fileid,IDIN,IDOUT,INTVEC,LENINT,errorcode)
```

Asynchronous Read of a Record

Read a record asynchronously. It is the caller's responsibility to ensure that the data array has sufficient space to contain the record. The length and type of the array may be determined from the functions IRRECL and IRTYPE described below.

```
CALL IRRDWA (fileid,IDIN,WPVEC,LENWP,errorcode)

CALL IRRDIA (fileid,IDIN,INTVEC,LENINT,errorcode)
```

Asynchronous Pre-positioning to a Record

Initiate positioning of the storage device to read the next desired record. A WAIT need not be issued for this operation to complete before starting a READ.

```
      CALL IRFIND (fileid,IDIN,errorcode)
```

Wait for Completion of Asynchronous Operation

When it becomes necessary to reuse a data area involved in an outstanding READ or WRITE operation, program execution must be suspended until the operation is complete. The WAIT function provides for this.

```
      CALL IRWAIT (fileid,IDIN,errorcode)
```

Check for Completion of Asynchronous Operation

When multiple transfer operations have been initiated, it may be desirable to test individual ones for completion without forcing a WAIT operation. The CHECK function provides for this. A zero value for the status code indicates that the operation has completed successfully. A non-zero value indicates that it is still in progress.

```
      CALL IRCHEK (fileid,IDIN,statuscode)
```

Issue Error Message

In order to provide the user with informative messages in the event of an error, the interface contains a standard routine which may be called after each I/O operation if desired. Its function is simply to use the error code recorded internally to print a descriptive message on a user-specified print file, denoted by printfileid. If the error code is 0, indicating no error condition exists, it will simply return immediately.

```
      CALL IRERMS (fileid,printfileid)
```

Obtaining the Record Length

The length of a record described by a record identifier IDIN(*) may be obtained through the function call

```
      LENGTH = IRRECL(fileid,IDIN)
```

The length is measured in storage units corresponding to the type of the data stored on the record.

Obtaining the Record Data Type

The type of data stored in a record described by a record identifier IDIN(*) may be obtained through the function call

```
      ITYPE = IRTYPE(fileid,IDIN)
```

The type code returned is 1 for INTEGER and 2 for WORKING PRECISION. These values were chosen so that they may conveniently be used in a CASE statement, or in a computed GO TO, if required. Note that the IRRECL and IRTYPE functions make it possible to copy the file without knowledge of the contents, apart from the maximum record lengths, provided that some method for storing the record identifiers has been established.

Obtaining the File Physical Blocksize

In order to permit user programs to optimize their output record sizes, a routine is provided to return the actual blocksize, in both working-precision and integer storage units, used on the file, if this is appropriate. For those devices for which a blocksize has no significance, zero values will be returned. The file must be open before invoking this function.

```
      CALL IRBLKL (fileid,LENWP,LENINT,errorcode)
```

Comparing Record Identifiers for Equality

It is sometimes desirable to be able to compare two record identifiers from the same file for equality. Unfortunately, on some machines, a comparison is implemented as an integer subtraction, which can result in an overflow condition if high-order bits are non-zero. To avoid this machine dependence, a function is provided to perform the test. It returns a .TRUE. or .FALSE. value, and consequently, must be explicitly typed by the user as a LOGICAL function.

```
      INTEGER ID1(2),ID2(2)
      LOGICAL EQUAL,IRCOMP
      EQUAL = IRCOMP(ID1,ID2)
```

Standard Error Codes

In order to allow a consistent and straightforward way of identifying error conditions, a standard set of error codes has been defined. These have been arranged such that a code of 0 indicates that no error condition exists, and values 1, 2, 3, ... are assigned to various error conditions. Negative error codes will not be used. Sequential error numbers make it straightforward to handle error processing with a CASE statement, or with a computed GO TO.

0.....No error condition exists.
1.....OPEN attempted on a file which is already open. Processing continues with the redundant open request ignored.
2.....CLOSE attempted on a file which is already closed. Processing continues with the redundant close request ignored.
3.....OPEN attempted on a non-existent file. When possible, an implementation will dynamically create a file to avoid issuing this message.
4.....Insufficient file space to write next record, or to open file with specified filesize parameter.
5.....I/O error detected on file.
6.....Invalid file unit number identifier. This must be an integer in the range 1..99.
7.....Invalid file record identifier. The identifier does not correspond to any existing record in the file.
8.....Invalid disposition code. The acceptable codes are 0 for KEEP, and 1 for DELETE, and 2 for SCRATCH.
9.....I/O operation attempted on file which is not open.
10....I/O operation still in progress on file. A WAIT must be issued to ensure completion of outstanding operations. When possible, an implementation will arrange to issue a WAIT internally to correct this condition if it occurs.
11....Insufficient internal table space available in interface to open a new file.

When any random-access I/O routine in the interface is entered, the error flag will be tested before performing the requested operation. If a serious error exists which prevents further processing in that implementation, return will occur immediately with further processing suppressed. The interface should try to avoid premature termination of a job through a STOP or CALL EXIT statement. In this way, the user is given an opportunity to take corrective action which might permit the job to be corrected and restarted later without a total loss of the processing time invested up to the point of error.

* Synchronous Random-Access I/O Routines *

Synchronous I/O READ and WRITE routines are defined analogous to the asynchronous versions simply by dropping the terminal letter A. It is not then necessary to use the WAIT or CHECK primitives, but it is not an error to do so. The routines are then as follows:

Synchronous Write of a Record

```
      CALL IRWRW (fileid,IDOUT,WPVEC,LENWP,errorcode)

      CALL IRWRI (fileid,IDOUT,INTVEC,LENINT,errorcode)
```

Synchronous Update of a Record

```
      CALL IRUPW (fileid,IDIN,IDOUT,WPVEC,LENWP,errorcode)

      CALL IRUPI (fileid,IDIN,IDOUT,INTVEC,LENINT,errorcode)
```

Synchronous Read of a Record

```
      CALL IRRDW (fileid,IDIN,WPVEC,LENWP,errorcode)

      CALL IRRDI (fileid,IDIN,INTVEC,LENINT,errorcode)
```

```
*******************************************
* Higher-Level Random-Access I/O Routines *
*******************************************
```

Many applications of direct-access I/O are most simply implemented if records are identified by record sequence numbers lying in some range 1..MAXREC, rather than by the more primitive record identifier. Sequence numbers are assigned beginning with 1 for the first record written, and incremented by unity for each record written thereafter. In some implementations, it may be possible to determine the location of a record directly from the sequence number, without having to store the record identifiers for all records in the file. The use of a record sequence number is already included in some manufacturers' implementations of random-access I/O in FORTRAN, and is also part of the 1977 FORTRAN Standard. For this reason, a set of higher-level routines has been defined. These differ from the basic primitives by the use of the prefix KR, rather than IR, and by the replacement of two-word record identifiers by single INTEGER sequence numbers. An implementation must implement these in terms of the primitives, and conceal the mapping of a record identifier to a record sequence number from the user. It is worth noting that this is not a completely general implementation, in that records will always be written in sequential order on the file, and the sequence numbers returned by the WRITE routines will always be an ascending sequence 1,2,3,... Counterparts of the IR routines are provided only for those which access record identifiers.

Asynchronous Write of a Record

Write a record asynchronously. The data areas involved in the transfer must not be modified until a WAIT has been issued to ensure completion of the operation. Note that letters 3 and 4 (WR) of the routine names indicate the operation (WRite), letter 5 indicates the data type (W - WORKING PRECISION, I - INTEGER), and letter 6 (A) indicates that the operation is Asynchronous. The record will be written at the end-of-information on the file, and the record sequence number will be returned to the caller.

```
CALL KRWRWA (fileid,NUMOUT,WPVEC,LENWP,errorcode)

CALL KRWRIA (fileid,NUMOUT,INTVEC,LENINT,errorcode)

fileid..........FORTRAN unit number in range 1..99.
NUMOUT..........INTEGER record sequence number
                returned to the caller.
WPVEC(*)........WORKING-PRECISION array to be written.
```

LENWP...........	Number of elements in WPVEC(*).
INTVEC(*).......	INTEGER array to be written.
LENINT..........	Number of elements in INTVEC(*).
errorcode.......	INTEGER error code returned to caller.

Asynchronous Update of a Record

Rewrite an existing record asynchronously. The new record will be written in place of the record identified by NUMIN if space permits, or will be added at end-of-information. NUMOUT will be set to the sequence number of the replacement record. The interface need not provide for garbage collection of vacant records, although it may do so. Vacant records can arise only when an update replaces a record by a larger one. If this is done frequently, large holes of unused space may arise in the file. Programmers who contemplate maintaining random-access files over an extended time period with frequent updating should consider developing a utility which can be used to copy the file to a new file with consequent deletion of vacant records.

```
      CALL KRUPWA (fileid,NUMIN,NUMOUT,WPVEC,LENWP,
     #             errorcode)

      CALL KRUPIA (fileid,NUMIN,NUMOUT,INTVEC,LENINT,
     #             errorcode)
```

Asynchronous Read of a Record

Read a record asynchronously. It is the caller's responsibility to ensure that the data array has sufficient space to contain the record.

```
      CALL KRRDWA (fileid,NUMIN,WPVEC,LENWP,errorcode)

      CALL KRRDIA (fileid,NUMIN,INTVEC,LENINT,errorcode)
```

Asynchronous Pre-positioning to a Record

Initiate positioning of the storage device to read the next desired record. A WAIT need not be issued for this operation to complete before starting a READ.

```
      CALL KRFIND (fileid,NUMIN,errorcode)
```

Wait for Completion of Asynchronous Operation

When it becomes necessary to reuse a data area involved in an outstanding READ or WRITE operation, program execution must be suspended until the operation is complete. The WAIT function provides for this.

```
CALL KRWAIT (fileid,NUMIN,errorcode)
```

Check for Completion of Asynchronous Operation

When multiple transfer operations have been initiated, it may be desirable to test individual ones for completion without forcing a WAIT operation. The CHECK function provides for this. A zero value for the status code indicates that the operation has completed successfully. A non-zero value indicates that it is still in progress.

```
CALL KRCHEK (fileid,NUMIN,statuscode)
```

Synchronous I/O READ and WRITE routines are defined analogous to the asynchronous versions simply by dropping the terminal letter A. It is not then necessary to use the WAIT or CHECK primitives, but it is not an error to do so. The routines are then as follows.

Synchronous Write of a Record

```
CALL KRWRW (fileid,NUMOUT,WPVEC,LENWP,errorcode)

CALL KRWRI (fileid,NUMOUT,INTVEC,LENINT,errorcode)
```

Synchronous Update of a Record

```
CALL KRUPW (fileid,NUMIN,NUMOUT,WPVEC,LENWP,errorcode)

CALL KRUPI (fileid,NUMIN,NUMOUT,INTVEC,LENINT,errorcode)
```

Synchronous Read of a Record

```
CALL KRRDW (fileid,NUMIN,WPVEC,LENWP,errorcode)

CALL KRRDI (fileid,NUMIN,INTVEC,LENINT,errorcode)
```

****
* Implementation Notes *
****

1..The error code MUST be initialized to 0 at entry to each primitive.

2..On those machines with very large address spaces, such as the VAX, PRIME, and possibly IBM, a reasonable implementation might be to maintain the entire random-access file in central memory, so that the paging hardware is effectively utilized to swap it in and out as required. If a CLOSE with KEEP specified were issued, the file could then be copied to an external storage device. If an OPEN with KEEP were issued, the interface would probably have to maintain the entire file on the external storage device in case of a premature termination of the job. Both these cases could be handled if the interface were arranged quite generally to maintain the first part of the file in central memory, and any overflow on external storage. Small files would then be processed very efficiently. One could also use this technique on machines with large secondary memory, such as CDC's LCM or ECS.

3..The interface should allow for at least 3 random-access files to be attached to the job at one time. This has implications for the size of any internal storage areas, such as common blocks, which are statically allocated at compile time. The interface shall NOT require the user to provide any internal workspace whatever, because the amount required would immediately introduce an implementation dependence into the user program.

4..The interface should strive to avoid pre-formatting, or skeletonizing, the file with dummy records. Even on systems such as UNIVAC and IBM which require this, it can be avoided if the interface arranges to write records sequentially, provided some means of recording the actual record location in the file is available. The IBM NOTE and POINT macros in BSAM might be suitable candidates for this.

5..The limitation of the record identifier to 2 INTEGER elements shall be maintained for all implementations. No length modifier is permitted on the type declaration INTEGER, so that the standard host computer's fullword integer type will be used. On machines with 24 or more bits per integer, this provides for at least a 16M byte range of identifiers, and if this were made a word address, instead of a byte address, 48M bytes could be addressed. By using a few bits from the second integer, the range can be increased. Implementations are unlikely on 16-bit machines, because of the 65K word address space limitation. However, even there, the interface could use both words to maintain a 32-bit record address, and the length and type could be recorded on the file itself if necessary.

6.. Error message lines should begin with a blank character in Column 1 and be no longer than 80 characters in length. Each message should begin with a common prefix string, e.g. "*** ERROR ***", allowing easy location of error messages with a text editor. It may also be useful to issue a subroutine traceback if this is available on the host computer, and to arrange for error messages to optionally appear in the job log file, if this is possible.

7.. On host computers where double precision is normally used for quantum chemistry computations and single precision for crystallography work, implementors should attempt to make available both single and double precision versions.

Standard Crystallographic File Structure — 79

(Trial Version)

April 1979

Interim Report of the Working Party on a Standard Crystallographic File Structure appointed by the Computing Commission and Data Commission of the International Union of Crystallography.

The working party was appointed at the XIth Congress of the International Union of Crystallography held in Warsaw in August 1978. We were instructed to propose a Standard Crystallographic File Structure that could be used for the transfer of machine-readable files of crystallographic data between laboratories. The present document contains a preliminary version of this file structure for use on a trial basis. It is not intended to be comprehensive but contains the essential elements needed for a crystal structure file. We intend in the final document to provide additional definitions covering, among others, the fields of protein and powder crystallography.

The final report will be presented to our parent commissions at the XIIth Congress of the IUCr at Ottowa in 1981 and at that time we reserve the right to change or withdraw any of the present definitions. We hope, however, that the only changes necessary will be extensions and that this definition will be compatible with the final version.

Any comments on the trial format should be addressed to the Chairman or other members of the working party.

Standard Crystallographic File Structure — 79
(Trial Version)

The Purposes of a Standard Crystallographic File

(in decreasing order of importance)

1. Exchange of crystallographic data between laboratories. Such data may comprise any of lattice parameters, symmetry information, atomic coordinates, structure factors, d-spacings, atomic scattering factors or chemical and bibliographic data necessary to characterize the compound(s).
2. Exchange of data between different crystallographic programs within a laboratory.
3. Storage of crystallographic data in a local database.

Possible Uses of a Standard Crystallographic File

1. Submission of crystallographic data to primary journals and databases. (Acta Crystallographica may soon request authors of crystal structure determinations to submit machine readable data.)
2. Exchange of databases of crystal structures between users examining crystal-chemical properties.
3. Exchange of structure factors between users interested in electron density studies.
4. Exchange of data on biopolymers. (Positional coordinates, phased structure factors, etc.)
5. Exchange of powder patterns.
6. Transferring data from one program to another, e.g., switching a refinement from X-RAY to SHELX or *vice versa*.
7. Exchange of crystallographic programs. It would be an advantage for all crystallographic programs to be able to accept (and output) data in the standard file structure in addition to other formats.

Criteria to be Met by the File Structure

(in decreasing order of importance)

1. *Must be extendable to include all types of crystallographic data.*
2. *Must be compatible with current and future methods of data transmission.* Currently cards are favored with magnetic tape second, but newer technologies must be considered.
3. *Should be easy to program for both reading and writing.* Files written in this structure are designed for machine to machine communication. Not all users will be experienced programmers or have access to large program systems. This implies the use of fixed formats. Users may well prefer to enter data in free format and use the computer to generate an exchange file in the standard format.
4. *The file should not require reread facilities* since these are not supported by IBM or ICL. (The working party has not reached a consensus on the importance of this criterion but it is observed in the present definition.)
5. *A listing of a file written in this format should be easy to read visually*, consistent with #3 above.
6. *The only records that must be included are those required for data management (e.g. END).* A standard crystallographic file will contain information of use to the writer and reader of the file. An author sending structural data to a journal will be interested in different data from workers exchanging powder patterns for phase identification.
7. *Provision should be made for the inclusion of derived data if required.* Some calculations, e.g. of structure factors [see Acta C. A34, p.819] may be based on elaborate models using programs not available in other laboratories. It should be possible to transmit this information in the standard format.
8. *Provision should be made for comments* remembering that the interpretation of information on a comment card by a computer is difficult or impossible. This information is essentially for people only.

The Data Structure of a Standard Crystallographic File

(For simplicity, the file is described in terms of card images but without implying that it must physically exist in the form of cards.)

1. A file consists of entries, each entry being logically independent of other entries. An entry normally will consist of data referring to one crystalline phase. Each entry begins with a TITLE card and ends with an END card.

2. An entry consists of a number of sections each including data of a particular type, e.g. atomic coordinates, structure factors. Each section begins with a Header card and ends with an End of Section card (a card with * in column 1). The End of Section card ensures that the program is ready to read the next card as a header.

3. Each section consists of formatted cards (or lines) containing not more than 80 characters. Five characters at the end of each card are reserved for sequence numbers (this is necessary as long as there is a danger of dropping a deck of cards).

4. The character set is restricted to the 46 characters 0-9, A-Z, , . + - * / ( ) = $\underset{\smile}{b}$. These characters are the only standard ones available on all machines.

5. Cards are of two types:
   Header Cards are used to start a new section. The first 8 characters indicate what information is to be found on the following data cards and in what format it appears. In addition each header card may include alphanumeric column labels (see sample file).
   Data Cards contain the data specified by the most recently read header card. Column 1 is blank except on the final card of the section.

6. Header cards that cannot be interpreted are ignored. Some consequences are:
   i) Blank cards may be used to separate sections for visual effect.
   ii) An incorrect header may result in the data in the following section being skipped, since the program will ignore all cards until it finds a header card it can interpret.

iii) Instruction or data cards for a user's program can be added to a file provided they do not mimic legal header cards. This can be ensured by using a character other than a letter or blank in columns 1-8.

iv) Comments can be inserted between sections, but this procedure could lead to problems and the use of a REMARK section is recommended.

7. Data cards that cannot be interpreted should be avoided. Since these will be read with a fixed format read statement they could cause a fatal read error.

8. Data of one kind (e.g. atomic coordinates) may be split into several sections, but where the file contains duplicate information (e.g. two CELL DIMension sections) the values appearing latest in sequence are the values that are used.

Formats for the Standard Crystallographic File Structure-79

Each section starts with the Header card shown. The first eight characters are reserved for an alphabetic section name. Otherwise the card may contain any other alphanumeric characters to act as column headings for the subsequent data. All the other cards in the section are data cards and have the format shown except the last card in the section which will have an asterisk in column 1. Columns 76-80 are reserved for sequence numbers. The TITLE and END cards must appear in all entries. Other sections may be included as required by the user. All microscopic dimensions (a,b,c,λ) are in Ångstrom units.

1. TITLE (1X,74A1)

   Name of compound and other identification
   This section must begin the file or follow an END card.

2. CELL DIMensions (1X,5A1,4A1,6F10.4)

   Data set key (see CONDITIONS).
   Any non-blank characters written in the 4A1 field, e.g. ERRS, result in the data being read as standard errors.

a, b, c, α, β, γ. Distances in Å, angles in degrees. All values must be given.

3. SPACE GRoup (1X,2A1,3X,5A1,20A1)

Lattice type (P, C, etc., H = hexagonal setting of rhombohedral cell). Center code (C = center of symmetry at origin; b, A or N no center at the origin).
Data set key.
Hermann-Mauguin space group symbol. (In the final version the space group symbol may be defined in a manner that is machine interpretable, obviating the need for including the symmetry section. In this version the symmetry section should be included whenever appropriate.)

4. SYMMETRY (10X,3(3I2,F10.7,4X))

Symmetry matrices for equivalent positions as (xx, xy, xz, xT, yx, yy, yz, yT, zx, zy, zz, zT) where

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} \begin{matrix} = \\ = \\ = \end{matrix} \begin{pmatrix} xx & xy & xz \\ yx & yy & yz \\ zx & zy & zz \end{pmatrix} (x,y,z) + \begin{pmatrix} xT \\ yT \\ zT \end{pmatrix}$$

Give all positions except those related by center of symmetry at origin and by lattice translations (centering).

5. ATOM COOrdinates (1X,2A1,3A1,4A1,3F8.5,2F6.4,3F6.5,2F5.4)

Atom name (element symbol left justified).
Atom identifier (any characters).
Atom type (to identify form factor).
x, y, z, U(isotropic), occupancy (default = 1 means site fully occupied including sites on special positions).
Standard errors in x, y, z, U and O. Note that these can be given as integers and will be interpreted as errors in the fifth decimal place (x,y,z) or fourth decimal place (U,O).
When two or more atoms share a site, each should have a separate card. These cards will be identical except for the identification fields and occupation. The sum of all occupations at a site must not exceed 1.0.

6. ATOM UIJ (1X,2A1,3A1,4A1,6F8.5)

   Atom name.

   Atomic identifier.

   Any non-blank characters written in the 4A1 field (e.g. ERRS) will result in the 6F8.5 fields being interpreted as standard errors. Each temperature factor may therefore be followed by its standard error.

   U11, U22, ... U23 or σ(U11), etc.

7. FORM FACtor (1X,4A1,5A1,3F10.4)

   Atom type, data set key, sin(θ)/λ, f, Δf". (f includes Δf')

8. CONDITIOns (4X,A1,5A1,5F10.5, cols.61-75 reserved for future definition)

   N = Neutron diffraction (X-ray is default)

   Data set key

   Wavelength

   Scale for F (obs)

   Temp (K)

   Linear absorption coefficient

   Observed density

   This section defines the conditions under which various data sets have been measured. When only one set of conditions has been used the data set key can be blank. When the data set key is defaulted in other sections, the relevant data applies to all conditions specified here.

9. HKL (5X,3I5,5A1,2F10.3, cols.46-75 reserved for future definition)

   h, k, l, data set key, F(obs), σ(F)

10. REMARK (1X,74A1)

    Messages for the user may be written in this section. Since it is difficult for the computer to interpret these messages, all data should be included in other sections if at all possible.

11. END

    Must be followed by NAME or End of File.

Members of the Working Party

Dr. I. D. Brown (Chairman)	Institute for Materials Research McMaster University Hamilton, Ontario, Canada, L8S 4M1
Dr. S. C. Abrahams	Bell Laboratories, Murray Hill, New Jersey, USA 07974
Dr. R. Diamond	MRC Laboratory of Molecular Biology Hills Road, Cambridge, CB2 2QH, UK
Dr. S. R. Hall	Crystallography Centre University of Western Australia Nedlands, WA 6009 Australia
Dr. A. C. Larson	P.O. Box 5898, Santa Fe, New Mexico, USA 87502
Dr. A. D. Mighell	Institute for Materials Research National Bureau of Standards Washington, D.C. USA 20234
Dr. E. Parthé	Lab. Cristallographie aux Rayon X, Universite of Geneva 32 Boulevard d'Yvoy CH-1211 Geneva 4, Switzerland
Dr. R. Shirley	Dept. of Chemical Physics University of Surrey Guilford, Surrey, GU2 5HK, UK

```
TITLE
LI2 B4 O7, A NEW STRUCTURE DETERMINATION BY I.O BROWN AND
*M.NATARAJAN

CELL DIMENSIONS  A          B          C         ALPHA      BETA      GAMMA
              9.477      9.477     10.256       90.        90.        90.
*     ERRS    0.0050        50          60

SPACE GROUP
*IA       I 41 C D
SYMMETRY
           1 0 0                    0 1 0                    0 0 1
          -1 0 0                    0 1 0                    0 0 1 0.5
           0 1 0                   -1 0 0 0.5                0 0 1 0.25
           0 1 0                    1 0 0 0.5                0 0 1 0.75
          -1 0 0                    0-1 0                    0 0 1
           1 0 0                    0-1 0                    0 0 1 0.5
           0-1 0                    1 0 0 0.5                0 0 1 0.25
*          0-1 0                   -1 0 0 0.5                0 0 1 0.75

ATOM COORDINATES X       Y       Z       U       O    S(X)   S(Y)   S(Z)   S(U)   S(O)
 LI       1 0.1496  0.1657  0.8529                     50     50     50
 B (1)    2 0.1633  0.0862  0.2010                     30     30     40
 B (2)    2 0.9465  0.1126  0.0824                     20     20     40
 O (1)    3 0.2813  0.1382  0.2653                     40     10     30
*O (2)    3 0.0671  0.1777  0.1565                     20     21     30
REMARK
*NOT ALL ATOMS HAVE BEEN INCLUDED IN THIS SAMPLE

ATOM UIJ      U11      U22     U33      U12       U13       U23
 LI         0.0258   0.0258  0.0385  0.0199   -0.0088   -0.0159
 LI   ERRS      22       23      29      19        20        22
 B (1)      0.0103   0.0098  0.0114  0.0013    0.0006    0.0003
 B (1)ERRS       9       10      10       8         8         9
 B (2)      0.0090   0.0102  0.0133  0.0008   -0.0006    0.0028
 B (2)ERRS      10       10      10       8         9         9
 O (1)      0.0103   0.0068  0.0180  0.0000    0.0037    0.0012
 O (1)ERRS       7        7       8       5         7         6
 O (2)      0.0099   0.0082  0.0173  0.0016   -0.0034    0.0004
*O (2)ERRS 0.0009        9       9       8         8         7

CONDITIONS LAMBDA      SCALE      TEMP
*          0.7109       1.0       297.

HKL       H    K     L          F(CBS)      SIGMA
          1    0     0          103.          9.
          2    0     0           57.          5.
*         3    0     0           82.          7.
REMARK
*THESE ARE DUMMY STRUCTURE FACTORS
```

```
FORM FACTORS  S/L       F           DF//
    1         0.0       2.000
    1         0.05      1.984
    1         0.10      1.936
    1         0.15      1.861
    1         0.20      1.752
    1         0.30      1.523
    1         0.40      1.266
    1         0.50      1.024
    2         0.0       5.0
    2         0.05      4.724
    2         0.10      4.060
    2         0.15      3.316
    2         0.20      2.699
    2         0.30      1.979
    2         0.40      1.681
*   2         0.50      1.526

END
```

APPENDIX

The BDF for XTAL

The XTAL binary data file (BDF) is divided into separate elements called "logical records". Each logical record contains specific crystallographic information that may be referred to by the logical record type numbers 1 to 25. The length of each record (in words) will vary according to type of information it contains, the type of structure being processed, and the current state of the analysis.

For convenience of access, each logical record type is subdivided into packets of words which form the particular logical subset of the data contained therein.

Packet Types

The different types of information stored in the binary data file necessitate three different logical record constructions. Records containing character information are distinct from those containing numerical data. The numerical records are also of two distinct types. One contains information that is of fixed length and is located in specific words of the record, while the other numerical record contains data which may vary greatly from structure to structure. These logical records are now summarized:

Character records contain only packed characters. The packet size of character records varies according to the number of characters per packet and the length of the floating point register. Logical records 1, 2, 7, and 10 are of this type.

Specific Information records contain numerical information data which have fixed location in a specific packet. Each word of data is accessed by adding the appropriate sequence number to the packet pointer provided by the file handling routines AA21 and AA22. Logical records 3 and 5 are of this type.

Directory driven records contain numerical data which can vary according to the size and nature of the structure. The first packet is used as a directory to the data contained in all subsequent packets. In this way the packets need only be as large as the available data or the calculation requires. This is achieved by assigning identification numbers to each unique data type and inserting these numbers into the first packet in the identical order that the actual data appears in subsequent packets. A pointer to the word containing any given data type is provided by the nucleus file handling routine AA23.

A-5.1 Structure and Contents of the Logical Records

LR 1 File History (character)

Log rec	Packet size	Sequence number	
1	number of words to hold 16 characters	Packet 1	Contains the compound ID
		Packet 2	This and subsequent packets contain program ID's and date as hhddmm at the time of updating. After 50 updates, the list is reset and starts over again.

LR 2 Label Information (character)

Log rec	Packet size	Sequence number	
2	number of words to hold 80 characters	Packet 1	Contains date and time of creation of BDF.
		Packet 2	Contains title in force at time of creation.
		Packet 3	This and subsequent packets contain images of any label information supplied.

LR 3 (spare)

Reserved for possible use as BDF status keys to enable lookahead capability in sequential mode.

LR 4 Cell Constants (specific information)

```
Log Packet Sequence
rec size   number
  4    9
         Packet 1
           IP+1     a cell dimension in Angstroms
           IP+2     b
           IP+3     c
           IP+4     cos(alpha)
           IP+5     cos(beta)
           IP+6     cos(gamma)
           IP+7     alpha in cycles (2pi = 1.0000)
           IP+8     beta
           IP+9     gamma

         Packet 2
    IP+1 - IP+9     Estimated standard deviations of the
                    quantities of Packet 1.
```

Packet 3	
IP+1 - IP+9	Reciprocal cell constants in same order as Packet 1.
Packet 4	
IP+1 - IP+9	Transformation matrix from fractional coordinates to orthogonal Angstrom coordinates.
Packet 5	
IP+1 - IP+9	Transformation matrix from Miller indices to orthogonal pseudo Miller indices.
Packet 6	
	Miscellaneous cell information
IP+1	cell volume
IP+2	observed crystal density

<u>LR 5 Symmetry Information</u> (specific information)

Log rec	Packet size	Sequence number	
5	12		
		Packet 1	Contains miscellaneous information
		IP+1	Code to indicate lattice type as.....

lattice type	P	I	R	F	A	B	C
acentric cell	1.	2.	3.	4.	5.	6.	7.
centric cell	8.	9.	10.	11.	12.	13.	14.

IP+2	Centric/acentric indicator 0/1
IP+3	Number of symops
IP+4	Number of distinct rotation matrices and translation vectors exclusive of lattice translations and center, if any.
IP+5	Number of rotation matrices of identical pattern of zeros
IP+6	Cell multiplicity factor to place a and b parts of the structure factor on the scale of int.tab. vol 1. This factor accounts for lattice type.
Packet 2	Contains the rotation matrices and translation vectors for first equivalent position.
IP+1	r(1,1)
IP+2	r(2,1)
IP+3	r(3,1)
IP+4	r(1,2)
IP+5	r(2,2)
IP+6	r(3,2)
IP+7	r(1,3)
IP+8	r(2,3)
IP+9	r(3,3)
IP+10	t(1)
IP+11	t(2)
IP+12	t(3)

Packet 3 to n+1 for the remaining n equivalent positions. The maximum value of n is 24. Matrices involving an inversion center or non-primitive translations are excluded.

__LR 6__ (spare)

__LR 7 Scattering Factor Names__ (character)

Log rec	Packet size	
7	number of words to hold six characters	Names of scattering factors contained in LR 8. Each packet contains the characters supplied as a scattering factor type. One packet for each different scattering factor type.

__LR 8 Atom-type Parameters__ (directory)

Log rec	Packet size	Ident. number	Directory in packet 1, first atom-type in packet 2
8	varies	1	number of atoms of this type per unit cell
		2	atomic weight
		3	atomic number
		4	number of electrons in neutral atoms or ions
		5	atomic bond radius in Angstroms
		6	atomic contact radius in Angstroms
		7	
		8	
		9	effective spin quantum number
		10	neutron scattering length in cm*10**-12
		21	real part of dispersion scatt. factor for data-set 1
		22	real part of dispersion scatt. factor for data-set 2
		..	. . . .
		61	imag part of dispersion scatt. factor for data-set 1
		62	imag part of dispersion scatt. factor for data-set 2
		..	. . . .
		100	atomic scattering factor at s = 0.00
		101	atomic scattering factor at s = 0.01
		102	atomic scattering factor at s = 0.02
		...	. . . .
		1nm	atomic scattering factor at s = 0.nm
		...	. . . .

299 atomic scattering factor at s= 1.99

Note (1) No scattering factor table is required if interpolated values have been stored with each hkl in the reflection record 20.

Note (2) Scattering factors at all s-intervals of 0.01 need *not* be present for interpolation.

Note (3) Additional scattering factors for a given atom type are stored 300-499, 500-699,700-899,...

<u>LR 9</u> (spare)

<u>LR 10 Data Set Definitions</u> (character)

Log rec	Packet size	
10	words/ 12 characters	Strings of 12 characters used to describe data sets. Order of strings corresponds to data-set number. Data sets may be defined as isomorphs, graphs of partial structures, or residues.

<u>LR 11 Experimental Parameters</u> (directory)

Log rec	Packet size	Ident. number	Directory in packet 1, one parameter set per packet
11	varies	1	data-set key (1 designates data-set1, etc.)
		2	wavelength (weighted mean) in Angstroms
		3	wavelength line1 in Angstroms
		4	wavelength line2 in Angstroms
		5	wavelength line3 in Angstroms
		6	relative weight wl line1
		7	relative weight wl line2
		8	relative weight wl line3
		9	measured density
		10	linear absorption coefficient in 1/cm
		11	temperature of measurement in degrees celsius
		12	sorting order of hkl
		13	a cell dimension in Angstroms
		14	b
		15	c
		16	cos(alpha)
		17	cos(beta)
		18	cos(gamma)
		19	alpha in cycles (2pi = 1.0000)
		20	beta
		21	gamma

31-39	diffractometer orientation matrix r11,r21, ,...,r33
100	number of scale groups for this data set
101	frel scale factor for scale-group 1
102	frel scale factor for scale-group 2
...	. . . .
100+n	frel scale factor for scale-group n (maximum allowed 64)

<u>LR</u> <u>12</u> <u>Data</u> <u>Set</u> <u>Information</u> (directory)

Log rec	Packet size	Ident. number	Directory in packet 1, one parameter set per packet
12	varies	1	data-set key (1 designates data-set1, etc.)
		2	overall temperature factor UOV in Angstroms squared
		10	packed word of =eval= fragment types
		11	maximum /h/
		12	maximum /k/
		13	maximum /l/
		14	minimum sin theta/lambda
		15	maximum sin theta/lambda
all initialized to VOIDFLG:		30	scale, data set to parent
		31	temp. factor, delta B, relative to parent
		32	closure error
		33	closure error, anomalous
		101	extinction type (0=none, 1=iso 1, 2=iso 2, 3=gen iso 1,2, and prime, 4=aniso 1, 5=aniso 2, 6=gen aniso)
		102	distribution (0=Gaussian, 1=Lorentzian)
		103	isotropic type1 parameter
		104	isotropic type2 parameter
		105-110	anisotropic type1 parameters
		111-116	anisotropic type2 parameters

<u>LR</u> <u>13</u> (spare)

<u>LR</u> <u>14</u> (spare)

<u>LR</u> <u>15</u> <u>Atomic</u> <u>Identification</u> (character)

Log rec	Packet size	
15	number of words to hold eight charac-	Each packet contains the string of characters which constitute an atom identification (6 characters) plus a 2 character dataset pointer (data number in ASCII). Their relative position in

ters	the packets is linked to the following record 16 which contains the atom parameters.

<u>LR 16 Atom Parameters</u> (directory)

Log rec	Packet size	Ident. number	Directory in packet 1, first atom data in packet 2
16	varies	1	x parameter in fractions of unit cell
		2	y parameter in fractions of unit cell
		3	z parameter in fractions of unit cell
		4	individual isotropic t.f. as B
		5	individual anisotropic t.f. stored as betas beta11
		6	beta 22
		7	beta 33
		8	beta 12
		9	beta 13
		10	beta 23
		11	population parameter
		12	anomalous population parameter
		13	neutron scattering factor
		21	atom multiplicity for atoms in special positions
		22	xray scattering factor pointer as a packet sequence number of LR 8.
		23	temperature factor type (0=overall;1=iso; 2=aniso)
		24	atom-group key for group refinements
		25	model-refinement key for refining different models

<u>LR 17 Std Dev in Atom Parameters</u> (directory)

Log rec	Packet size	Ident. number	Directory in packet 1, first atom s.d. in packet 2
17	varies	1	sigma x
		2	sigma y
		3	sigma z
		4	sigma b
		5	sigma beta 11
		6	sigma beta 22
		7	sigma beta 33
		8	sigma beta 12
		9	sigma beta 13
		10	sigma beta 23
		11	sigma of population parameter
		12	sigma of anomolous population parameter
		13	sigma of neutron scattering factor

<u>LR 18 Refinement Constraints</u> (directory)

```
Log Packet Ident.  Directory in packet 1, first constraint in
rec size   number  packet 2
 18 varies
  Note:  The general form of the constraint equation is...
         p(s)*f(s)=q+p(r1)*f(r1)+p(r2)*f(r2)+...+p(rn)*f(rn)
              1    packet sequence number of subject atom in
                   logical record type 11
              2    parameter identification number of subject
                   atom
              3    multiplication factor of subject parameter
              4    constant Q in constraint equation
              5    constraint classification key
              6    site multiplicity of subject atom

             11    packet sequence number of the reference
                   atom 1
             12    parameter identification number of
                   reference atom 1
             13    multiplication factor for parameter of
                   atom 1

             21    packet sequence number of reference atom 2
             22    parameter identification number of reference
                   atom 2
             23    multiplication factor for parameter of
                   atom 2

             ..    .  .  .  .  .  .  .  .  .  .  .  .

          n1-n3    packet, parameter, and mult. factor for
                   atom n
```

<u>LR</u> <u>19</u> (spare)

<u>LR</u> <u>20</u> <u>Reflection</u> <u>Information</u> (directory)

```
Log Packet Ident.  Directory in packet 1, first reflection
rec  size  number  in packet 2
 20 varies
                    * numbers    1- 999 identify crystal-
                                        specific data
                    * numbers 1000-1999 identify data-set 1
                                        information
                    * numbers 2000-2999 identify data-set 2
                                        information
                    *  .  .   .  .  .     .  .  .  .  .  .
                    * numbers n000-n999 identify data-set n
                                        information
          Crystal
          specific
             1     Miller indices packed word, with bit pattern
                       29-21    20-12    11-3    2-0
                        /h/      /k/      /l/    sign code
                                                 (see below)
```

```
    2     sin(theta)/lambda

    3     reflection multiplicity and reinforcement
          factor                   9-5    4-0
                                epsilon  hkl mult.

 4-15     equivalent indices packed table (up to 12
          words).  The table appears in sets of *two*
          words.

          *** word1 describes index magnitudes
          29-21    20-12    11-3    2-0
           /h/      /k/      /l/  no. of sign/phase
                                  codes in word2

          *** word2 describes the index signs and
              phase shifts

    23-21 20-18 17-15 14-12 11-9   8-6   5-3   2-0
    phase sign  phase sign  phase sign  phase sign
    code4 code4 code3 code3 code2 code2 code1 code1
```

	sign	phase-shift	
code	hkl	degrees	cycles
0	+++	0	0.00000
1	++-	60	0.16667
2	+-+	90	0.25000
3	+--	120	0.33333
4	-++	180	0.50000
5	-+-	240	0.66667
6	--+	270	0.75000
7	---	300	0.83333

```
   501    interpolated scattering factor for atom
          type1
   502    interpolated scattering factor for atom
          type2
   ...     .   .   .   .   .  .   .   .   .   .   .
   510    interpolated scattering factor for atom
          type10

   The 700 numbers are used to store estimated
   phase sets for the "native" structure or
   "parent" substance

   700     Figure of merit; weight of the
           'best' Fourier coefficient
   701     cos alpha for the 'best' Fourier coef.
   702     sin alpha for the 'best' Fourier coef.
   703     cos alpha most probable
   704     sin alpha most probable
705-709    alternate phase set 2

795-799    alternate phase set 20

      *
```

```
           * n000-n199 identify measurement parameters
           * n200-n299 identify reduction parameters
 for all   * n300-n499 identify reduced structure
                       factor data
data sets  * n500-n599 identify Hendrickson coefficient
                       data
   n       * n600-n699 identify normalized s.f. data
           * n700-n799 identify structure factor
                       phase data
           * n800-n899 identify refined structure
                       factor data
           * n900-n999 identify refinement parameters
           *
```

data-set1

1000	total gross counts
1001	total background counts
1002	ratio of scan to background time
1003	net counts
1004	sigma(net counts)
1005	phi diffractometer angle in cycles
1006	chi or kappa
1007	omg
1008	2th
1009	2th scan range
1010	omg scan range
1200	absorption weighted mean pathlength tbar
1201	absorption correction factor to irel
1202	extinction correction factor to irel
1203	thermal diffuse scatt. correction factor to irel
1204	1/lp factor
1205	irel scale factor to scale counts to irel
1300	relative intensity (irel)
1301	sigma(irel)
1302	relative f squared (f2rel)
1303	sigma(f2rel)
1304	relative /f/ (frel)
1305	sigma(frel)
1306	relative /f/ friedel related -h,-k,-l (frel*)
1307	sigma(frel*)
1308	rcode reflection status key (user designated)
1309	scale group number
501-1504	A,B,C,D Hendrickson coefficients Phase probability distribution (isomorphous)
505-1508	A,B,C,D Hendrickson coefficients Phase probability distribution (anamolous)
1600	normalized structure factor 1; assuming random atoms
1601	normalized structure factor 2; with fragment information

1602	expectation value for f**2; assuming random atoms
1603	expectation value for f**2; with fragment information
1604-1630	group s.f. in sequence designated by LR 17 (ID 10)
1631	weight of s.f. phase estimate 1 with id 1701
1632	weight of s.f. phase estimate 2 with id 1702
....	. . . .
1694	weight of s.f. phase estimate 64 with id 1764
1700	current structure factor phase estimate (in cycles)
1701	structure factor phase estimate 1 (in cycles)
1702	structure factor phase estimate 2 (in cycles)
....	. . . .
1764	structure factor phase estimate 64 (in cycles)
1800	calculated /f/ (fcal)
1801	A sum normal S.F. only (=/f/cos(phase))
1802	B sum normal S.F. only (=/f/sin(phase))
1803	A dispersion contribution only
1804	B dispersion contribution only
1805	A total excluding extinction correction
1806	B total excluding extinction correction
1807	translation function coefficient
1810-1817	partial structure factor values in order 1800-1807
1900	least squares weight last used
1901	least squares weight1
1902	least squares weight2
1903	least squares weight3

LR 21 (spare)

LR 22 (spare)

LR 23 (spare)

LR 24 (spare)

LR 25 END-OF-FILE Record (specific)

Log rec	Packet size	Sequence number	Description of contents
25	0		This record serves as EOF signal to nucleus

A-5.2 Physical Structure of the Binary Data File

The physical structure of the BDF on the output or input device is not of particular importance to the XTAL user or programmer. This is because the XTAL nucleus routines handle all the bookkeeping operations and return data in terms of logical records and packets. However, for those who wish to write their own BDF drivers, a brief description of the BDF structure follows. The length of all logical records is determined solely by the crystal and amount of information it contains. Storage requirements in direct-access memory force certain physical constraints on the maximum number of words that can be output or input to or from an I/O device at one time. The memory reserved for this transfer is referred to as the I/O buffer, and in the XTAL system these buffers are located in the data array QX( ). The length of these buffers is specified by the macro (BINSEQBUF:) when XTAL is implemented. This value will depend on the core available, and other hardware constraints, such as the disc track length. Once the buffer length has been set for a given installation, it must not be changed. To optimize the transfer of the binary data file to and from the fixed length I/O buffers, it is necessary to both pack and position logical records according to length. This operation, in turn, requires that three additonal words at the front of each logical record or buffer are used for bookkeeping purposes. These three floating point words are referred to as lead words and set in the following way:

lead word 1 is the length in floating point words, including the three lead words, of the part or all of a given logical record in this buffer. The end of a buffer is signaled when the first word following a record has the value of +1. or -1. The +1. signals that the preceding logical record does *not* continue into the next buffer. The -1. signals that the preceding logical record is incomplete and continues into the next buffer.

lead word 2 is the logical record type number (1 to ENDRECORD:). This number is negative when the last part of a logical record is in the current buffer. It is positive when more of the logical record follows in the next buffer.

lead word 3 is the packet size in floating point words for the given logical record.