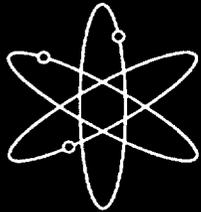
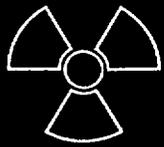
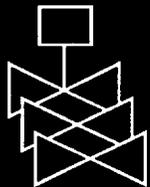


TRAC-M/FORTRAN 90 (Version 3.0) Programmer's Manual



Los Alamos National Laboratory



**U.S. Nuclear Regulatory Commission
Office of Nuclear Regulatory Research
Washington, DC 20555-0001**



AVAILABILITY OF REFERENCE MATERIALS IN NRC PUBLICATIONS

NRC Reference Material

As of November 1999, you may electronically access NUREG-series publications and other NRC records at NRC's Public Electronic Reading Room at www.nrc.gov/NRC/ADAMS/index.html.

Publicly released records include, to name a few, NUREG-series publications; *Federal Register* notices; applicant, licensee, and vendor documents and correspondence; NRC correspondence and internal memoranda; bulletins and information notices; inspection and investigative reports; licensee event reports; and Commission papers and their attachments.

NRC publications in the NUREG series, NRC regulations, and *Title 10, Energy*, in the Code of *Federal Regulations* may also be purchased from one of these two sources.

1. The Superintendent of Documents
U.S. Government Printing Office
Mail Stop SSOP
Washington, DC 20402-0001
Internet: bookstore.gpo.gov
Telephone: 202-512-1800
Fax: 202-512-2250
2. The National Technical Information Service
Springfield, VA 22161-0002
www.ntis.gov
1-800-553-6847 or, locally, 703-605-6000

A single copy of each NRC draft report for comment is available free, to the extent of supply, upon written request as follows:

Address: Office of the Chief Information Officer,
Reproduction and Distribution
Services Section
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001

E-mail: DISTRIBUTION@nrc.gov
Facsimile: 301-415-2289

Some publications in the NUREG series that are posted at NRC's Web site address www.nrc.gov/NRC/NUREGS/indexnum.html are updated periodically and may differ from the last printed version. Although references to material found on a Web site bear the date the material was accessed, the material available on the date cited may subsequently be removed from the site.

Non-NRC Reference Material

Documents available from public and special technical libraries include all open literature items, such as books, journal articles, and transactions, *Federal Register* notices, Federal and State legislation, and congressional reports. Such documents as theses, dissertations, foreign reports and translations, and non-NRC conference proceedings may be purchased from their sponsoring organization.

Copies of industry codes and standards used in a substantive manner in the NRC regulatory process are maintained at—

The NRC Technical Library
Two White Flint North
11545 Rockville Pike
Rockville, MD 20852-2738

These standards are available in the library for reference use by the public. Codes and standards are usually copyrighted and may be purchased from the originating organization or, if they are American National Standards, from—

American National Standards Institute
11 West 42nd Street
New York, NY 10036-8002
www.ansi.org
212-642-4900

Legally binding regulatory requirements are stated only in laws; NRC regulations; licenses, including technical specifications; or orders, not in NUREG-series publications. The views expressed in contractor-prepared publications in this series are not necessarily those of the NRC.

The NUREG series comprises (1) technical and administrative reports and books prepared by the staff (NUREG-XXXX) or agency contractors (NUREG/CR-XXXX), (2) proceedings of conferences (NUREG/CP-XXXX), (3) reports resulting from international agreements (NUREG/IA-XXXX), (4) brochures (NUREG/BR-XXXX), and (5) compilations of legal decisions and orders of the Commission and Atomic and Safety Licensing Boards and of Directors' decisions under Section 2.206 of NRC's regulations (NUREG-0750).

DISCLAIMER: This report was prepared as an account of work sponsored by an agency of the U.S. Government. Neither the U.S. Government nor any agency thereof, nor any employee, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for any third party's use, or the results of such use, of any information, apparatus, product, or process disclosed in this publication, or represents that its use by such third party would not infringe privately owned rights.

TRAC-M/FORTRAN 90 (Version 3.0) Programmer's Manual

Manuscript Completed: April 2001
Date Published: May 2001

Prepared by
B.T. Adams, J.F. Dearing, P.T. Giguere, R.C. Johns,
S.J. Jolly-Woodruff, J.W. Spore, R.G. Steinke, LANL

J.H. Mahaffy, C. Murray, PSU

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Pennsylvania State University
University Park, PA 16802

F. Odar, NRC Project Manager

Prepared for
Division of Systems Analysis and Regulatory Effectiveness
Office of Nuclear Regulatory Research
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001
NRC Job Code W6245



TRAC-M/FORTRAN 90 (VERSION 3.0) PROGRAMMER'S MANUAL

by

B. T. Adams, J. F. Dearing, P. T. Giguere, R. C. Johns, S. J. Jolly-Woodruff,
J. Mahaffy, C. Murray, J. W. Spore, and R. G. Steinke

ABSTRACT

The Transient Reactor Analysis Code (TRAC) was developed to provide advanced best-estimate predictions of postulated accidents in light-water reactors. The TRAC-P program has provided this capability for pressurized water reactors and for many thermal-hydraulic test facilities for approximately 20 years. However, the maintenance and portability of TRAC-P had become cumbersome because of the historical nature of the code and the inconsistent use of standardized Fortran. Thus, the Modernized TRAC (TRAC-M) was developed by recoding the TRAC-P algorithms to take advantage of the advanced features available in the Fortran 90 programming language while conserving the computational models available in the original code.

The TRAC code (i.e., both the versions P and M) features a one-, two-, or three-dimensional (1D, 2D, or 3D) treatment of the pressure VESSEL and its associated internals, a two-fluid nonequilibrium hydrodynamics model with a noncondensable-gas field and solute tracking, flow-regime-dependent constitutive equation treatment, optional reflood tracking capability for bottom- and top-flood and falling-film quench fronts, and a consistent treatment of the entire set of accident sequences, including the generation of consistent initial conditions. The stability-enhancing two-step (SETS) numerical algorithm is used in the solution of the 1D, 2D, and 3D hydrodynamics and permits violation of the material Courant condition. This technique permits large timesteps, and thus, the running time for slow transients is reduced. A heat-structure (HTSTR) component is included that allows the user to model heat transfer accurately for complicated geometries. An improved reflood model that is based on mechanistic and defensible models has been added. TRAC also contains improved constitutive models and additions and refinements for several components.

This manual is one of a four-volume set of documents on TRAC-M. This manual was developed to assist a programmer and contains information on the TRAC-M code and data structure, the TRAC-M calculational sequence, memory management, and data precision. This document provides a code developer with a single source of information to allow either modification of or addition to the code. Sufficient information is provided to permit replacement or modification of physical models and

correlations. Within TRAC, information is passed at two levels. The upper level of information is passed by systemwide and component-specific data modules at and above the level of component subroutines. At the lower level, information is passed through a combination of module-based data structures and argument lists. This document describes the basic mechanics involved in the flow of information within the code. This document directly incorporates significant information regarding the code models and architecture.

CONTENTS

	Page
ABSTRACT	iii
AUTHORS AND ACKNOWLEDGMENTS.....	xiii
1.0. INTRODUCTION.....	1-1
2.0. TRAC-M CALCULATIONAL SEQUENCE	2-1
2.1. General Summary	2-2
2.1.1. Constrained Steady State	2-7
2.1.2. HPSS Initialization	2-9
2.2. Input Processing.....	2-10
2.2.1. 1D Component Input Processing with Subroutine rdcomp	2-14
2.2.2. 3D Component Input Processing with Subroutine rvssl.....	2-17
2.2.3. Component Input Processing with Subroutine rdrest.....	2-18
2.3. Initialization.....	2-20
2.3.1. 1D Component Initialization with Subroutine icode	2-23
2.3.2. 3D Component Initialization with Subroutine civssl.....	2-27
2.4. Prepass, Outer-Iteration, and Postpass Calculations.....	2-28
2.4.1. Prepass Calculation	2-31
2.4.1.1. 1D Component Prepass Calculation with prep1d.....	2-33
2.4.1.2. 3D Component Prepass Calculation with prep3d.....	2-34
2.4.2. Outer-Iteration Calculation	2-35
2.4.2.1. 1D-Component Outer-Iteration Calculation with out1d	2-36
2.4.2.2. 3D-Component Outer-Iteration Calculation with out3d	2-37
2.4.3. Postpass Calculations	2-38
2.4.3.1. 1D Component Postpass Calculation with post	2-39
2.4.3.2. 3D-Component Postpass Calculation with post3d.....	2-39
2.4.3.3. HTSTR-Component Calculation with htstr3	2-39
2.5. Timestep Advancement and Backup	2-40
2.6. Output Processing.....	2-41
2.6.1. ASCII Output Processing with edit	2-42
2.6.2. Graphics Output Processing with xtvdr	2-43
2.6.2.1. Module XtvData (February 2000).....	2-45
2.6.2.2. Module XtvSetup (February 2000).....	2-45
2.6.2.3. Module XtvComps (February 2000).....	2-46
2.6.2.4. Module XtvDump (February 2000).....	2-46
2.6.2.5. Module CXtvXFaces (February 2000).....	2-46
2.6.2.6. The XTV/XMGR5 C Library (February 2000)	2-46
2.6.3. Binary Restart File Processing with dmpit.....	2-46

CONTENTS (cont)

	Page
3.0. CODE ARCHITECTURE.....	3-1
3.1. Code Structure.....	3-1
3.1.1. Fortran 90 Modules.....	3-2
3.1.2. Description of All Structural Elements.....	3-2
3.2. Data Structure and Data Communication.....	3-2
3.2.1. Overview.....	3-2
3.2.1.1. TRAC Databases.....	3-3
3.2.1.2. Database Communication.....	3-4
3.2.1.3. Fortran 90 Modules.....	3-5
3.2.1.4. Derived Data Types.....	3-5
3.2.1.4.1. Global-Database-Derived Types.....	3-6
3.2.1.4.2. Component-Derived Types.....	3-6
3.2.1.4.3. Control-System-Derived Types.....	3-6
3.2.1.4.4. Steady-State-Derived Types.....	3-6
3.2.1.4.5. Radiation-Model-Derived Types.....	3-6
3.2.1.5. Data Precision.....	3-10
3.2.2. Databases.....	3-10
3.2.2.1. Global Data.....	3-10
3.2.2.1.1. Modules Global and GlobalPnt.....	3-11
3.2.2.1.2. Flow Equation Solution and System Services.....	3-13
3.2.2.2. Component-Type Database.....	3-14
3.2.2.2.1. 1D-Hydrodynamic-Component Types (PIPE, etc.).....	3-18
3.2.2.2.2. Pseudo-1D Boundary-Condition- Component Types (BREAK, FILL).....	3-27
3.2.2.2.3. "0D" Multiple-Connection-Component Type (PLENUM).....	3-29
3.2.2.2.4. 3D Hydrodynamic-Component Type (VESSEL).....	3-33
3.2.2.2.5. HTSTR-Component Type.....	3-51
3.2.2.3. Control System Databases.....	3-56
3.2.2.4. Steady-State Databases.....	3-66
3.2.2.5. Radiation Model Databases.....	3-67
3.2.3. Data Communication.....	3-67
3.2.3.1. Intercomponent Communication via System Services.....	3-67
3.2.3.1.1. Specification of the System Configuration.....	3-68
3.2.3.1.2. Setup for Boundary Information Transfer.....	3-75
3.2.3.1.3. System Service Setup Programming Guidelines.....	3-78
3.2.3.1.4. Transfer of Component Boundary Information.....	3-80

CONTENTS (cont)

	Page
3.2.3.2. Data Access—Instantiated Component with Task-Crunch Association.....	3-81
3.2.3.3. Data Access—Instantiated Component—No Task-Crunch Association.....	3-89
3.2.3.4. Data Access—Noninstantiated Component	3-95
3.2.3.5. HTSTR to Fluid Data Communication	3-102
4.0. INPUT/OUTPUT IN SI OR ENGLISH UNITS.....	4-1
5.0. PLATFORM IMPLEMENTATIONS AND PORTABILITY.....	5-1
5.1. Numerical Precision	5-1
5.2. Portability Issues	5-1
6.0. CODE DEVELOPMENT AND MAINTENANCE ENVIRONMENT AND STANDARDS.....	6-1
6.1. Updates and Configuration Control	6-1
6.1.1. Central Repository	6-1
6.1.2. Version Control System	6-1
6.1.3. Update and Version Documentation	6-1
6.2. Shadow Database.....	6-1
6.3. Coding Standards	6-1
6.3.1. Source Format Protocol.....	6-1
6.3.2. Uniform Style.....	6-2
6.3.3. Data Precision.....	6-2
7.0. REFERENCES	7-1
APPENDIX A.....	A-1
APPENDIX B.....	B-1
B.1. PROGRAMs.....	B-1
B.2. MODULEs.....	B-1
B.3. INTERFACES.....	B-28
B.4. PROCEDUREs.....	B-29
B.5. SUBROUTINEs.....	B-29
B.6. FUNCTIONs.....	B-155
B.7. BLOCK DATAs	B-166
B.8. INCLUDE files.....	B-166
APPENDIX C.....	C-1
C.1. Module Bad	C-1
C.2. Module BadInput	C-1
C.3. Module Bits	C-1
C.4. Module Boundary	C-6

CONTENTS (cont)

	Page
C.5. Module BreakArray.....	C-6
C.6. Module BreakVlt	C-7
C.7. Module Ccfl	C-10
C.8. Module CompTyp.....	C-10
C.9. Module ControlDat.....	C-11
C.10. Module EngUnits	C-23
C.11. Module EosData.....	C-28
C.12. Module EosInline.....	C-29
C.13. Module EosNoInline	C-30
C.14. Module FailDat.....	C-30
C.15. Module FillArray.....	C-32
C.16. Module FillVlt.....	C-33
C.17. Module Flt	C-36
C.18. Module Gen1DArray.....	C-38
C.19. Module Global	C-55
C.20. Module GlobalDat	C-56
C.21. Module GlobalDim.....	C-69
C.22. Module GlobalPtr	C-70
C.23. Module HSArray.....	C-72
C.24. Module HeatArray	C-83
C.25. Module HpssDat	C-84
C.26. Module IntArray	C-87
C.27. Module IntrType	C-87
C.28. Module Io.....	C-87
C.29. Module JunTerms	C-89
C.30. Module Linear	C-92
C.31. Module Matrices	C-92
C.32. Module Network.....	C-109
C.33. Module OneDDat.....	C-110
C.34. Module PipeArray.....	C-112
C.35. Module PipeVlt.....	C-113
C.36. Module PlenArray.....	C-117
C.37. Module PlenVlt.....	C-118
C.38. Module Plenum.....	C-120
C.39. Module PrizeVlt	C-121
C.40. Module PumpArray.....	C-124
C.41. Module PumpVlt.....	C-125
C.42. Module Restart.....	C-131
C.43. Module RodCrunch.....	C-132
C.44. Module RodGlobal	C-132
C.45. Module RodHtcrefl.....	C-133
C.46. Module RodVlt	C-134
C.47. Module SemiSolver.....	C-145

CONTENTS (cont)

	Page
C.48. Module Sepd.....	C-145
C.49. Module SepdVlt.....	C-146
C.50. Module SysConfig.....	C-148
C.51. Module SysService.....	C-157
C.52. Module SysTime.....	C-160
C.53. Module Tee	C-160
C.54. Module TeeArray	C-161
C.55. Module TeeVlt	C-161
C.56. Module Temp.....	C-170
C.57. Subroutine tf3ds.....	C-171
C.58. Module Thermocple.....	C-172
C.59. Module TimeStepDat	C-172
C.60. Module Util	C-173
C.61. Module ValveArray.....	C-174
C.62. Module ValveVlt	C-174
C.63. Module VectDrag	C-179
C.64. Module VessArray.....	C-180
C.65. Module VessArray3.....	C-187
C.66. Module VessCon.....	C-206
C.67. Module VessMat.....	C-208
C.68. Module VessTf3dc	C-208
C.69. Module VessVlt.....	C-209
C.70. Module Xtv	C-217
C.71. Module Xvol	C-217
C.72. Include File (Common-Block) bandw	C-218
C.73. Include File bignum	C-218
C.74. Include File (Common-Block) cflow	C-218
C.75. Include File (Common-Block) chfint.....	C-219
C.76. Include File (Common-Block) chgalp.....	C-219
C.77. Include File (Common-Block) ciflim.....	C-220
C.78. Include File (Common-Block) cnrslv.....	C-220
C.79. Include File (Common-Block) concck.....	C-221
C.80. Include File (Common-Block) condht.....	C-221
C.81. Include File (Common-Block) constant.....	C-221
C.82. Include File (Common-Block) decayc.....	C-222
C.83. Include File (Common-Block) defval.....	C-222
C.84. Include File (Common-Block) diddle.....	C-223
C.85. Include File (Common-Block) diddlh.....	C-225
C.86. Include File (Common-Block) diddli.....	C-226
C.87. Include File dlimit (Common-Block dlim).....	C-226
C.88. Include File (Common-Block) dmpck	C-227
C.89. Include File (Common-Block) dtinfo.....	C-228
C.90. Include File (Common-Block) elvkf	C-229

CONTENTS (cont)

	Page
C.91. Include File (Common-Block) film.....	C-229
C.92. Include File (Common-Block) h2fdbk.....	C-230
C.93. Include File (Common-Block) htcav	C-230
C.94. Include File (Common-Block) htcref2	C-230
C.95. Include File (Common-Block) htcref3	C-230
C.96. Include File (Common-Block) htcs.....	C-231
C.97. Include File (Common-Block) ifcrs	C-231
C.98. Include File (Common-Block) infohl.....	C-237
C.99. Include File (Common-Block) junction.....	C-237
C.100. Include File (Common-Block) massck.....	C-237
C.101. Include File (Common-Block) nrcmp	C-237
C.102. Include File (Common-Block) pmpstb.....	C-238
C.103. Include File (Common-Block) refhti	C-238
C.104. Include File (Common-Block) refhti2	C-239
C.105. Include File (Common-Block) rows.....	C-239
C.106. Include File (Common-Block) sepcb	C-239
C.107. Include File (Common-Block) solcon.....	C-240
C.108. Include File (Common-Block) stncom.....	C-240
C.109. Include File (Common-Block) strtnt.....	C-241
C.110. Include File (Common-Block) supres.....	C-241
C.111. Include File (Common-Block) syssum.....	C-241
C.112. Include File (Common-Block) totals.....	C-242
C.113. Include File (Common-Block) tst3d	C-242
C.114. Include File (Common-Block) vckdat.....	C-243
C.115. Include File (Common-Block) vdvmod.....	C-243
C.116. Include File (Common-Block) vellim.....	C-243
C.117. Include File (Common-Block) webnum.....	C-244
APPENDIX D.....	D-1
APPENDIX E	E-1
E.1. Introduction	E-1
E.2. Global Variable Graphics.....	E-1
E.3. Signal-Variable, Control-Block, and Trip-Signal Graphics.....	E-2
E.4. General 1D Hydraulic-Component Graphics	E-2
E.5. BREAK-Component Graphics.....	E-4
E.6. FILL-Component Graphics.....	E-5
E.7. HTSTR (Heat-Structure)-Component ROD- or SLAB-Element Graphics.	E-5
E.8. PIPE-Component Graphics.....	E-7
E.9. PLENUM-Component Graphics.....	E-7
E.10. PRIZER (Pressurizer)-Component Graphics.	E-8
E.11. PUMP-Component Graphics.	E-8

CONTENTS (cont)

	Page
E.12. TEE-Component Graphics.....	E-8
E.13. VALVE-Component Graphics.	E-9
E.14. 3D VESSEL-Component Graphics.....	E-9
APPENDIX F	F-1
APPENDIX G.....	G-1
G.1. New Component Variables	G-1
G.1.1. Summary	G-1
G.1.2. Adding a New Variable To Data-Type genTabT (the component FLT)	G-3
G.1.3. Adding A New Variable To Data-Types "comp_type"TabT (The Component VLTs)	G-13
G.1.4. Adding A New Component Array Variable.....	G-27
G.1.4.1. 1D Hydrodynamic Components	G-27
G.1.4.2. 3D Vessel-Component Arrays.....	G-46
G.1.4.3. System Services	G-57
G.1.5. HTSTR Arrays	G-57
G.2. Adding A New XTV Graphics Variable	G-60
G.2.1. Understanding Variable Attributes.....	G-60
G.2.2. Steps to be Completed before Adding Variables to Output.....	G-62
G.2.3. PrintVarDesc Interface	G-66
G.2.4. WriteStaticVx Interface.....	G-67
G.2.5. XtvBufx Interface	G-67
G.2.6. LuMatch Interface	G-68
APPENDIX H	H-1
H.1. Dump/Restart	H-1
H.2. Graphics	H-1
H.2.1. Overview of Changes in Version 3.0.....	H-1
H.2.2. Summary of XTV Header Format.....	H-2
H.2.3. Summary of XTV Data Format	H-4
H.2.4. Detailed Header File Format.....	H-5

FIGURES

	Page
Fig. 2-1 TRAC-M computational flow.....	2-3
Fig. 2-2 Transient-calculation flow diagram.....	2-4
Fig. 2-3 Steady-state-calculation flow diagram.....	2-6
Fig. 3-1 Module VessCon	3-40
Fig. 3-2 Boundary Array Layout.....	3-69
Fig. 3-3 Graphical representation of the junComp array	3-72
Fig. 3-4 Graphical representation of the coupling between the junCells and junComp arrays for a FILL, TEE, PIPE, BREAK, and BREAK system.....	3-72
Fig. 3-5 Graphical representation of the compSeg array.....	3-73
Fig. 3-6 Flow logic for System Service initialization	3-76

TABLES

	Page
Table 2-1 First Index of the Component-Junction Array jun	2-16
Table 2-2 Component-Specific Driver Subroutines	2-28
Table 3-1 Component Data Types	3-7
Table 3-2 Control System Data Types	3-9
Table 3-3 Steady-State Data Types	3-10
Table 3-4 VESSEL-Array Dimension Variables	3-39
Table 3-5 TRAC Component Data-Access Routines ^a	3-96

AUTHORS AND ACKNOWLEDGMENTS

Many people contributed to recent TRAC-P and TRAC-M code development and to this report. Because this work was a team effort, there was considerable overlap in responsibilities and contributions. The participants are listed according to their primary activity. Those with the prime responsibility for each area are listed first.

Principal Investigators:	J. F. Dearing, John Mahaffy, Jay W. Spore, Susan J. Jolly-Woodruff, Ju-Chuan Lin, Ralph A. Nelson, and Robert G. Steinke
Fluid Dynamics:	Jay W. Spore, Susan J. Jolly-Woodruff, Ju-Chuan Lin, and Robert G. Steinke
Heat Transfer:	Ralph A. Nelson, Kemal Pasamehmetoglu, Norman M. Schnurr, and Cetin Unal
Neutronics:	Robert G. Steinke and Jay W. Spore
Code Development and Programming:	J. F. Dearing, John Mahaffy, C. Murray, Susan J. Jolly-Woodruff, Paul T. Giguere, Ju-Chuan Lin, Jay W. Spore, and Robert G. Steinke
Control Procedure:	Robert G. Steinke
Graphics:	Russell C. Johns, James F. Dearing, Victor Martinez, and Michael R. Turner
Report Compilation:	B. Todd Adams and Paul T. Giguere
Editing:	Lisa G. Rothrock
Word Processing:	Ann B. Mascareñas

In addition to those contributors listed above, we acknowledge all others who contributed to earlier versions of TRAC. In particular, the two-step numerics developed by John Mahaffy is a major part of TRAC. Dennis R. Liles contributed heavily to the thermal-hydraulics modeling and to the overall direction of MOD1 code development. Frank L. Adessio developed the steam-generator component, and Manjit S. Sahota developed the critical-flow model and the turbine component. Thad D. Knight provided direction for improvements to TRAC based on assessment-calculation feedback and coordinated the development of the MOD1 Correlation and Models document. Richard J. Pryor, Sandia National Laboratories, and James Sicilian, Flow Science, Inc., provided major contributions to the code architecture. We also acknowledge useful discussions and technical exchanges with Louis M. Shotkin and Novak Zuber, United States Nuclear

Regulatory Commission; Terrence F. Bott, Francis H. Harlow, David A. Mandell, and Burton Wendroff, Los Alamos National Laboratory; John E. Meyer and Peter Griffith, Massachusetts Institute of Technology; S. George Bankoff, Northwestern University; Garrett Birkhoff, Harvard University; and Ronald P. Harper, Flow Science Inc.

1.0. INTRODUCTION

This manual has been developed to assist the Transient Reactor Analysis Code (TRAC) programmer. Sufficient information is provided to permit replacement or modification of physical models and correlations, as well as either the addition or modification of system components. Within TRAC, information is passed at two levels. Information at the upper level is passed by systemwide and component-specific data modules at and above the level of "component" subroutines. At the lower level, information is passed through a combination of module-based data structures and argument lists. This document describes the mechanics involved in the flow of information within the code. It is written specifically for Modernized TRAC Fortran 90 (TRAC-M/F90), Version 3.0. We will usually refer to this code as TRAC or TRAC-M. Topics of discussion addressed in this manual include the TRAC-M calculational sequence, code and data structure, computer-memory management, and various machine configurations that are supported. Much of the information contained herein is provided in the appendices, which are self-contained and meant to be used as references. The table of contents provides a listing of the appendices. This manual is a complete standalone document for TRAC-M. Occasionally we refer to TRAC-P constructs, but only for the additional benefit of those already familiar with that code. The TRAC-M PathFinder, a set of HTML pages containing a description and source listing for each of the program routines, also has been developed to allow navigating through the code with the use of a web browser.

This manual is one of four documents that form the basic TRAC-M documentation set. The other three are the Theory Manual ([Ref. 1](#)), the User's Manual ([Ref. 2](#)), and the Developmental Assessment Manual, which is yet to be published. The developmental assessment of various TRAC-M code versions will be performed by the NRC, and the results will be published in the future. Some of the material on the TRAC-M's computational flow was adapted from the Programmer's Manual for TRAC-PF1/MOD2 ([Ref. 3](#)).

2.0. TRAC-M CALCULATIONAL SEQUENCE

The full TRAC-M calculational sequence involves several stages: input processing; initialization; prepass, outer-iteration, and postpass calculations; timestep advancement and backup; and output processing. Within TRAC, information is passed via systemwide and component-specific data modules at and above the level of component subroutines, such as `rpipe`, `repipe`, `ipipe`, `pipe1`, `pipe2`, `pipe3`, `dpipe`, `xtvpipe` and `wpipe`. Examples of system-level data modules are `GlobalDat`, `GlobalPnt`, and `GlobalDim`. Examples of component-specific data modules are `Pipe`, `PipeArray`, and `PipeVlt`. Information is passed through a combination of module-based data structures and argument lists below these modules. The code and data structures are described fully in Section 3, and only the high-level aspects of the information passing and storage will be discussed within this section. The most complex and frequently modified interfaces exist in the component-specific subroutines. These subroutines are provided for each of the nine key stages of TRAC execution:

1. Input of initial component data (e.g., `rpipe`);
2. Input of restart information for a component (e.g., `repipe`);
3. Initialization of component-dependent variables (e.g., `ipipe`);
4. Solution of the stabilizer momentum equation, evaluation of various old-time quantities, and other bookkeeping at the beginning of each timestep (e.g., `pipe1`);
5. Iterative solution of basic flow equations for each timestep (e.g., `pipe2`);
6. Solution of stabilizer mass and energy equations, solution of the conduction equations, and other computations to complete each timestep (e.g., `pipe3`);
7. Output of data to the restart dump file (e.g., `dpipe`);
8. Output of data to the XTV graphics files (e.g., `xtvpipe`); and
9. Output of data to the ASCII detailed edit file (e.g., `wpipe`).

Similar component subroutines also exist for each of the nine key stages of TRAC execution for the other system components, e.g., TEE, FILL, BREAK, PUMP, PRIZER, SEPD, VALVE, VESSEL, and PLENUM. Each of these stages is discussed in greater detail, using a PIPE component as an example, in the sections that follow. First, a summary of the overall calculational sequences for transient and steady-state calculations is given.

2.1. General Summary

TRAC-M may perform a steady-state calculation, a transient calculation, or both, depending on the values of the input parameters `stdyst` and `transi` (Main-Data Card 4). A schematic illustrating TRAC's top-level program flow, with emphasis on the computational solution of the flow equations for a transient case, is presented in Fig. 2-1. Referring to the figure, the program construct for advancing the solution one timestep is controlled by subroutine `trans` and begins with (1) the prepass to obtain the stabilizer step for the equation of motion (subroutine `prep`), followed by (2) a call to the Newton iteration subroutine `hout` to perform the outer iteration and thus obtain the basic solution for all equations (subroutine `outer`), and concluded with (3) the postpass to obtain the stabilizer step for mass and energy equations (subroutine `post`). Within a given timestep, subroutine `prep` calls all of the component subroutines twice, subroutine `outer` calls all of the component subroutines twice per Newton iteration, and subroutine `post` calls all of the component subroutines three times. In each case (`prep`, `outer`, and `post`), two of the passes provide setup and solution for a set of equations. Subroutine `post` adds a third pass to calculate some final end-of-timestep values for mass flows and mean cell densities. The internal loops in subroutines `prep`, `outer`, and `post` are indexed by the variable `ibks`. This variable takes on values of one and two in `prep`; values of zero and one in `outer`; and one, two, and three in `post`. The component subroutines use the module `OneDDat` to pass the value of `ibks` to lower-level routines to control the flow of the calculation

The subroutines shown in Fig. 2-1 (`input`, `init`, `steady`, `trans`, `prep`, `outer`, and `post`) all access lower-level subroutines. A complete calling tree for TRAC-M is presented in Appendix A (starting at the entry `NOMOD: :PROGRAM Trac`). The general control sequences for each type of calculation are outlined in the subsections that follow, using the PIPE component as an example, with the specific details of the calculational sequence discussed in more detail.

The complete flow control for subroutine `trans` is shown in Fig. 2-2. The major control variables within the timestep loop are `nstep`, the current timestep number; `timet`, the time since the transient began; `delt`, the current timestep size; and `oitno`, the current outer-iteration number. The timestep loop is controlled by module `TimeStep` and begins with the selection of the timestep size, `delt`, by subroutine `timstp`. Again, a prepass is performed for each component by subroutine `prep` to evaluate the control parameters, stabilizer motion equations, and phenomenological coefficients. At this point in the calculation, with the current timestep number at zero, `trans` calls the `edit` subroutine to print the system-state parameter values and the `xtvdr` subroutine to generate a graphics edit at the beginning of the transient. Subroutine `trans` then calls subroutine `hout`, which performs one or more outer iterations to solve the basic hydrodynamic equations. Each outer iteration is performed by subroutine `outer` and corresponds to one iteration of a Newton-method solution procedure for the fully coupled difference equations of the flow network. The outer-iteration loop ends when the outer-iteration convergence criterion (`epso` on Main Data Card 5) is met. This criterion requires that the maximum fractional change in the pressure throughout the system during the last iteration be $\leq \text{epso}$. Alternatively, the outer-iteration loop may terminate when the number of outer iterations reaches a user-specified limit `oitmax` (Main Data Card 6). When

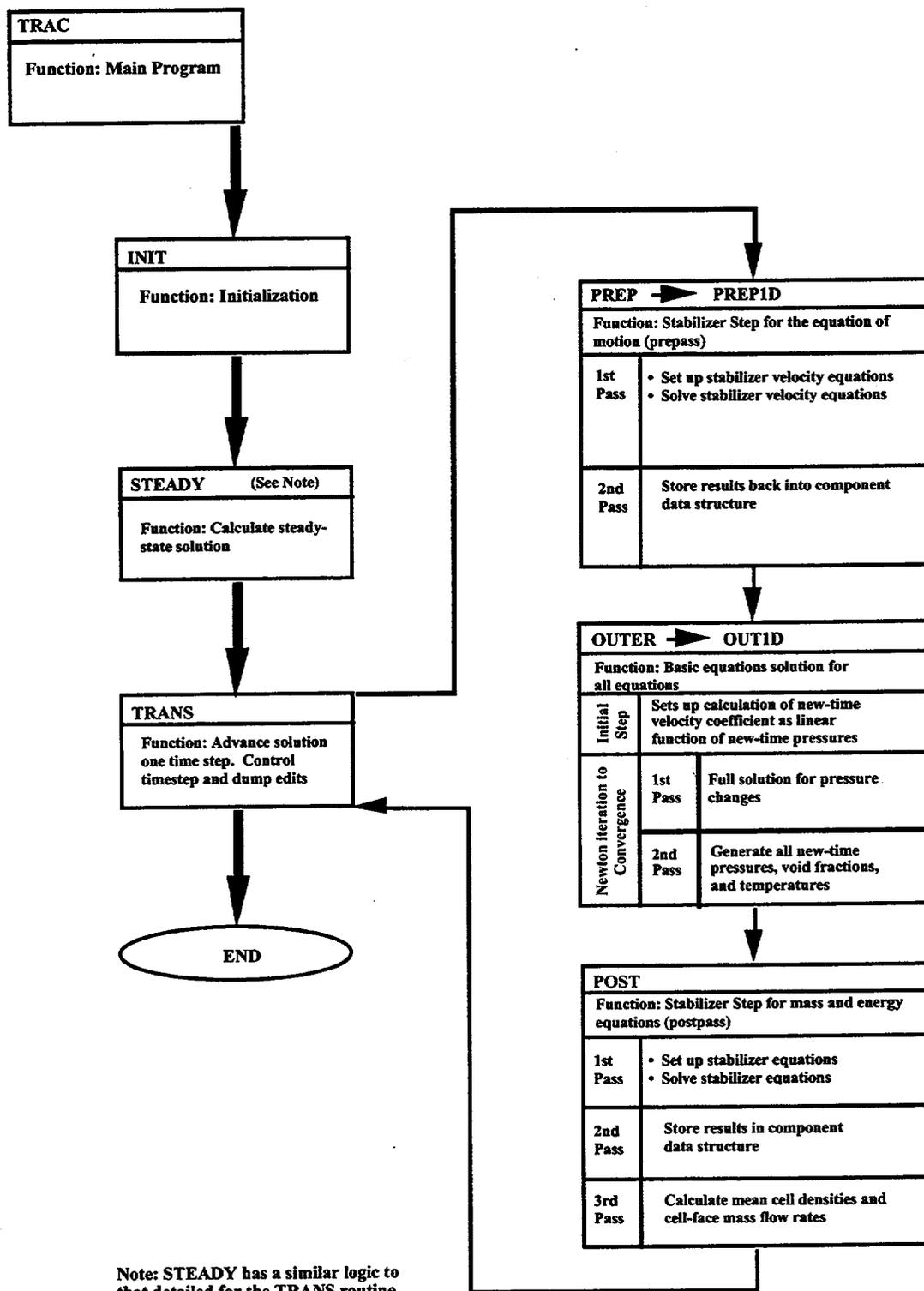


Fig. 2-1. TRAC-M computational flow.

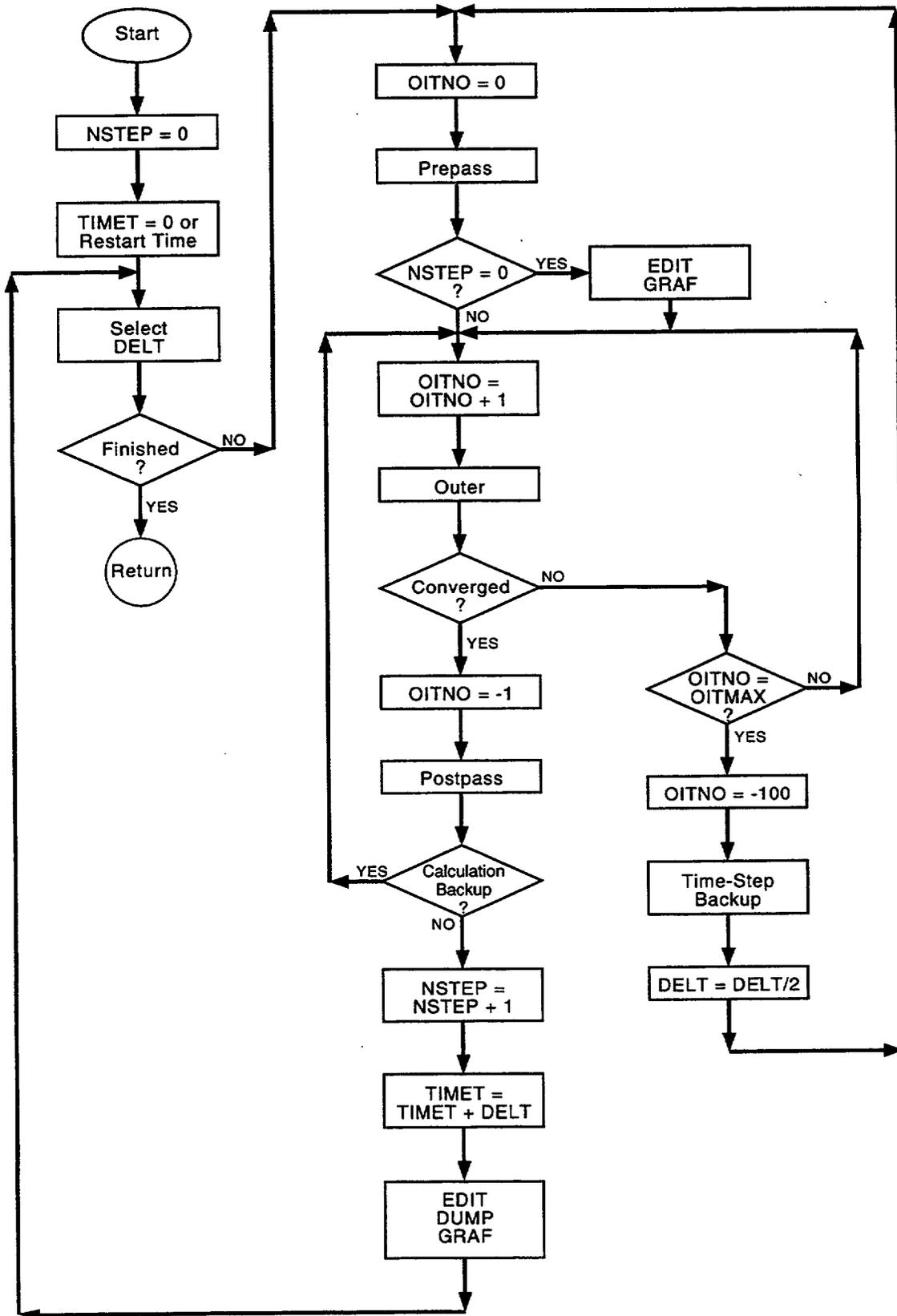


Fig. 2-2. Transient-calculation flow diagram.

this happens, TRAC-M restores the thermal-hydraulic state of all components to what it was at the beginning of the timestep, reduces the `delt` timestep size (with the constraint that `delt` be greater than or equal to the `dtmin` specified on Time Step Data Card 1), and continues the timestep calculation with the new timestep size. This process comprises a backup situation and is discussed in greater detail in Sec. 2.5.

Subroutine `trans` calls the `post` subroutine to perform a postpass evaluation of the stabilizer mass and energy equations and the heat-transfer calculation when the outer iteration converges. The `nstep` timestep number then is incremented by 1, and the `timet` problem time is increased by `delt`. Finally, subroutine `trans` invokes the `edit`, `sedit`, `dmpit`, and `xtvdr` subroutines by calling subroutine `pstepq` to provide the output results required by the user. The calculation is finished when `timet` reaches the last `tend` time (Time Step Data Card 1).

The transient calculation is controlled by a sequence of time domains input with the Time Step Data Cards and stored within module `GlobalDat`. During each of these time domains, the minimum (`dtmin`) and maximum (`dtmax`) timestep sizes (Time Step Data Card 1) and the long- (`edint`) and short-edit (`sedint`), dump (`dmpint`), and graphics (`gfint`) time intervals (Time Step Data Card 2) are defined. Note that the values for these timestep variables may be replaced by the same inputs for the Trip-Initiated Time Step Data Cards 3 and 4 if a trip is activated. When the `edit`, `sedit`, `dmpit`, and `xtvdr` subroutines are invoked, they calculate the time when the next output of the associated type is to occur by incrementing the current time by its time interval. When `trans` later finds that `timet` has reached or exceeded the indicated time, the corresponding output routine is invoked again. Whenever `timet` equals or exceeds the `tend` ending time for a timestep data domain, the next timestep data domain is read by subroutine `timstp`. The output indicators then are set to the sum of the current time and the newly input values for the output time intervals. Subroutine `steady` directs steady-state calculations using the structure shown in Fig. 2-3. Referring to the figure, the same sequence of evaluations used for a transient calculation also is used for a steady-state calculation. The main difference in subroutine `steady` is the addition of a steady-state convergence test, logic to turn on the steady-state power level, an optional evaluation of constrained steady-state (CSS) controllers, and an optional hydraulic-path steady-state (HPSS) initialization of the initial hydraulic-state estimate. To provide output results, `steady`, like `trans`, invokes the `edit`, `sedit`, `dmpit`, and `xtvdr` subroutines by calling subroutine `pstepq`. Subroutine `steady` is called by the TRAC main program, regardless of whether a steady-state calculation has been requested by `stdyst` (Main Data Card 4). If no steady-state calculation is to be done (`stdyst` = 0), `steady` returns to the TRAC main program. The TRAC main program then calls `trans` and performs a transient calculation if requested with `itrans` = 1 (Main Data Card 4).

Timestep control in `steady` is identical to that implemented in `trans`. This includes the selection of the timestep size, the timing for output, and the backup of a timestep if the outer-iteration limit is exceeded. In `steady`, the input variable `sitmax` (Main Data Card 6) is the maximum number of outer iterations used in place of `oitmax`. The maximum fractional rates of change per second of seven thermal-hydraulic parameters are calculated by subroutines `tf1ds3` [for one-dimensional (1D) components] and `ff3d` for

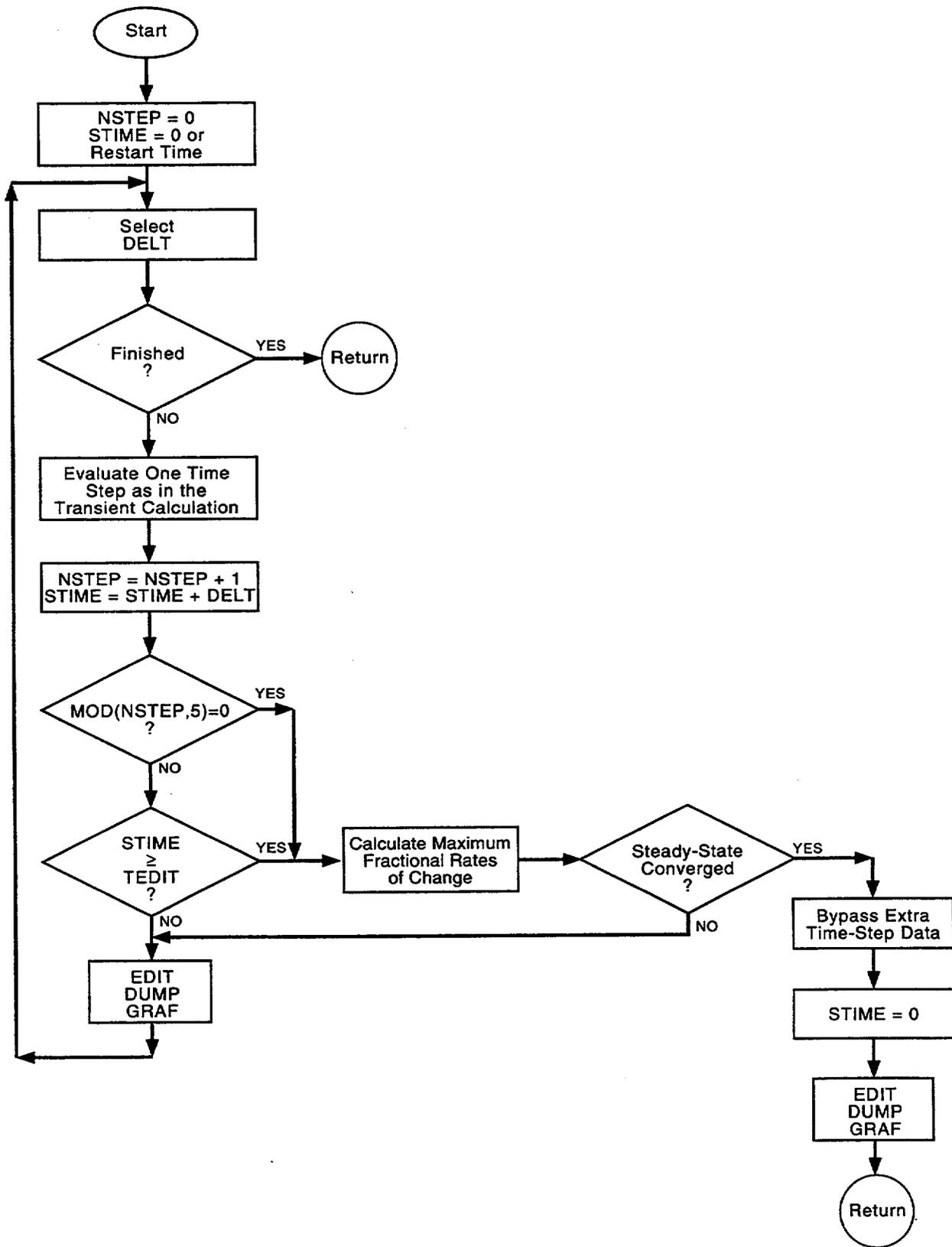


Fig. 2-3. Steady-state-calculation flow diagram.

the three-dimensional (3D) VESSEL components]. These rates and their locations in the system model are passed to subroutine `steady` through the array variables `fmax` and `lok` that are located in module `GlobalDat`. Tests for steady-state convergence are performed every five timesteps and before every large edit. The maximum fractional rates of change per second and their locations are written to the `TRCMSG` and `TRCOUT` files, as well as to the terminal. The total reactor core power is initialized to the input value `rpowri` (HTSTR Component Card 19) after the problem time, `timet`, reaches the value input for namelist variable `tpowr` when namelist variable `ipowr` is set to negative one. The minimum value of the flow velocity, `minvel`, and its maximum fractional rate of change, `fmxl vz`, in the hydraulic channels coupled to powered heat structures determine when the steady-state power should be set on for the case when namelist variable `ipowr` is set to zero (the default). The steady-state power is set to its input value, `rpowri`, once either `minvel` exceeds 0.5 m/s and `fmxl vz` falls below 0.5 or `timet` exceeds input time `tpowr`. Finally, the total reactor core power is initialized at the beginning of the steady-state calculation when namelist variable `ipowr` is set to one. The steady-state calculation is completed when all maximum fractional rates of change per second are below the user-specified convergence criterion `epss` (from Main Data Card 5) or when `stime` reaches the `tend` (Time Step Data Card 1) end time of the last time domain specified in the steady-state calculation timestep data.

Five types of steady-state calculations may be selected based on the value of `stdyst` (Main Data Card 4): generalized steady state (GSS) for `stdyst` = 1 (as described above), CSS for `stdyst` = 2, GSS with HPSS initialization for `stdyst` = 3, CSS with HPSS initialization for `stdyst` = 4, and static steady-state (SSS) check for `stdyst` = 5. A GSS calculation, as described above, evaluates a pseudo-transient timestep solution that asymptotically converges to the steady-state solution. A CSS calculation is a GSS calculation where additional user-defined component-action adjustments are made by a proportional-integral (PI) controller to achieve either a known or desired hydraulic steady-state condition. The nature of the available CSS controllers, their evaluation, and their database are described subsequently. Both generalized and CSS calculations with HPSS attempt to accelerate convergence by allowing the user to input estimates regarding the final steady-state condition. An SSS calculation checks for erroneous momentum and heat sources in a plant model by neglecting evaluation of the pump momentum source and the heat transfer. Thus, the fluid flow is expected to go to zero asymptotically with the expectation that the system temperatures will not change.

Both steady-state and transient calculations may be performed during one computer run. The end of the steady-state timestep cards is signified by a single card containing a `-1.0`. The transient timestep cards should follow immediately. If the steady-state calculation converges before reaching the end of its last time domain, the remaining steady-state timestep data are read in but not used so that the transient calculation proceeds as planned with its own timestep data.

2.1.1. Constrained Steady State

A CSS controller adjusts an uncertain component-action state to achieve a better-known hydraulic condition in the steady-state solution. The TRAC user can select four types of CSS controllers. Each type can be applied to one or more components in a plant model. A

type-1 CSS controller adjusts a pump impeller's rotational speed to achieve a desired fluid mass flow through the PUMP component. A type-2 CSS controller adjusts a VALVE's flow-area fraction to achieve a desired adjacent-cell upstream fluid pressure or fluid mass flow through the VALVE component's adjustable interface. A type-3 CSS controller performs one of three different adjustments (pump-impeller rotational speed of a PUMP component, flow-area fraction of a VALVE component, or mass flow in or out of a FILL component) to achieve a desired fluid mass flow through its component that equals the fluid mass flow at a designated location in the plant model. A type-5 CSS controller performs one of four different adjustments to an HTSTR component or its hydraulically coupled BREAK components (hydraulic-channel fluid pressure at the inner or outer surface; heat-transfer area at the inner, outer, or both surfaces; thermal conductivity of the inner, outer, or both surface nodes or of all nodes; or heat-transfer area of both surfaces and thermal conductivity of all nodes) to achieve a desired single-phase fluid temperature or two-phase gas volume fraction at a designated location in the plant model. The type-4 CSS controller was eliminated when the STGEN component was removed from TRAC. It adjusted the secondary-side fluid pressure or the tube inner and outer heat-transfer areas of a steam generator to achieve a desired primary-side downstream-location liquid temperature. By remodeling an STGEN component with PIPE, TEE, and HTSTR components, the functionality of the type-4 CSS controller is provided by a subset of the functionality of the type-5 CSS controller.

Each of the `ncontr` (Main Data Card 6) user-defined CSS controllers requires one input-data record CSS-Controller Card with four or five values that will be read by subroutine `input` [adjusted-component identification (ID) number, minimum and maximum range of parameter adjustment, either the type or location of the monitored parameter that is to have a desired value, and the type of adjusted parameter]. Each CSS controller's desired hydraulic parameter value is input at its monitored-parameter location in the component data. CSS-controller data are not written to the dump/restart file and so need to be reinput by the TRACIN file if the CSS calculation is continued with a restart. The number of CSS controllers and their input parameters can be changed during a restart. Components defining the desired hydraulic-parameter value for each CSS controller also need to be reinput by the TRACIN file. This later requirement makes restarting a CSS calculation inconvenient. Generally, TRAC users evaluate a CSS calculation to steady-state convergence without doing a CSS-calculation restart.

Interactive feedback between CSS controllers must be considered by TRAC users when defining the controllers. Their derived form assumes no interactive feedback. When the adjustments of two or more CSS controllers are strongly coupled by the thermal-hydraulic solution, their predicted controller adjustments may be bad, causing the solution to wander and not converge to the desired thermal-hydraulic parameter values. One such interaction has been programmed for in TRAC. When a type-5 CSS controller adjusts the fluid pressure where a type-2 CSS controller defines the desired value for an upstream fluid pressure, the pressure adjustment of the type-5 CSS controller also is applied to the desired value for the type-2 CSS controller's upstream fluid pressure. The desired value of the upstream fluid pressure becomes a moving target for the type-2 CSS controller, just as the desired fluid mass flow at a specified location in the plant model for a type-3 CSS controller becomes a moving target when it varies each timestep.

These four CSS-controller types are programmed for user convenience. An equivalent controller (except for the heat-transfer area and thermal conductivity adjustments of a type-5 CSS controller) could be defined directly through input with signal variables, control blocks, and component actions of the TRAC control system. For controller types that are not programmed, the TRAC user can define them through input as long as the controller's adjustment is an existing component action (see the TRAC Theory Manual¹). Additional component action and CSS controller types could be programmed if their availability is required by the user community.

2.1.2. HPSS Initialization

The initial thermal-hydraulic steady-state solution estimate, user specified by the hydraulic-component input data, generally can be improved by the HPSS initialization procedure in TRAC before the steady-state calculation is evaluated. Doing this generally reduces the computational effort of the steady-state calculation. The user selects this option by adding 2 to the value of `stdyst` for a GSS and CSS calculation; i.e., `stdyst = 1` and 2 for a GSS and CSS calculation, respectively, may be defined as `stdyst = 3` and 4 for a GSS and CSS calculation, with its initial thermal-hydraulic steady-state solution estimate internally initialized by TRAC during the initialization phase of the calculation.

Choosing the HPSS initialization procedure option requires the TRAC user to input HPSS initialization data in the TRACIN file. These input data are defined by the input data format description in Section 6.5 of the TRAC User's Manual (Ref. 2). In specifying this data, the 1D hydraulic component network of the plant model is partitioned into a number of `npaths` (specified by variable `npaths` on HPSS Data Card 1) connecting and nonoverlapping 1D flow paths. All possible flow paths in the network are considered unless either the input hydraulic-component data already define such a flow condition (and are not connected to a PLENUM component) or their steady-state flow is not expected to be significant. Even paths without flow may be considered to define an appropriate thermal condition (not defined by the 1D hydraulic-component data). The input hydraulic-component data need only to be defined as isothermal with no flow when selecting the HPSS initialization option. During the initialization phase, TRAC replaces the hydraulic-component gas volume fraction, phasic temperatures, and phasic velocities input data with the thermal-hydraulic parameter values that are specified by the HPSS initialization.

HPSS initialization data are what the TRAC user either knows or estimates the steady-state thermal-hydraulic solution will be along each of the 1D flow paths. Each flow path has its entrance and exit mesh-cell interfaces defined where inflow and outflow occur to the path. A known or estimated steady-state phasic-temperatures and phasic-velocities flow condition is defined at a single mesh-cell interface anywhere within the 1D flow path (inclusive of its end interfaces). The total and noncondensable-gas pressures may be defined as constant along each flow path or defined by the hydraulic-component data. A significant power source or sink along a subrange of mesh cells within the path also must be defined (such as for heat transfer between the primary and secondary sides of a heat exchanger). The flow paths can begin and end at any mesh-cell interface, as long as they are different interfaces and do not overlap internally with the cells of other 1D flow paths. However, the flow paths must begin and end at (1) the internal-junction interface

of a TEE component, (2) a junction of a PLENUM component, and (3) a source-connection junction of a VESSEL component. The internal-junction interface of a TEE component and the junction of a PLENUM component must define the phasic-temperatures and phasic-velocities flow condition of its 1D flow path. PLENUM-component junctions are assumed to have no steady-state fluid flow if they do not define the end interface of a flow path. However, the fluid flow condition at VESSEL-component, source-connection junctions may be either input-specified by hydraulic-component data or initialized by HPSS initialization data. This process provides sufficient information for TRAC internally to initialize the steady-state, thermal-hydraulic condition of all hydraulic components along each flow path, as well as of all the PLENUM and VESSEL components to which such 1D flow paths may be connected.

The hydraulic-component wall and HTSTR-component ROD and SLAB temperatures are defined by the input-component data and are not initialized by the HPSS initialization procedure. The same applies to the total and noncondensable gas pressures unless they are initialized with a constant value for all cells of a flow path. Structure temperatures and coolant pressures need not be initialized accurately because the steady-state calculation quickly determines their steady-state condition consistent with the gas volume fraction, phasic temperatures, and phasic velocities defined by the HPSS initialization procedure. On the other hand, the gas volume fraction, phasic temperatures, and phasic velocities are the slowest to converge to their steady-state solution and usually require at least three or four convective-flow passes through each 1D flow path to converge to their steady-state values if a significant change is required in the initial thermal-hydraulic solution estimate. Providing a good initial estimate for the gas volume fraction, phasic temperatures, and phasic velocities can significantly reduce the TRAC evaluation time needed to satisfy the user-input steady-state convergence criteria.

2.2. Input Processing

The processing of the majority of TRAC-M input data is controlled by the system-level subroutine `input` (the exception being that the timestep data are read by subroutine `timstp`, which is called directly by either subroutine `steady` or `trans`). The data are of two types: input data retrieved from the ASCII input data file `TRACIN` and binary restart data retrieved from the dump-restart file `TRCRST`. The user has the option of creating an echo file of the input data contained in file `TRACIN` by defining namelist variable `inlab = 3`. With this option, a file named `INLAB` (INput LABeled) is created during input data processing and has all the input data from file `TRACIN` output to it, along with variable-name comments contained between asterisks. This provides a useful means of labeling an otherwise difficult-to-interpret `TRACIN` file. It also allows the user to verify the input data being read by TRAC-M. Comments between asterisks in the original `TRACIN` file are not output to the `INLAB` file. All input data from files `TRACIN` and `TRCRST` are either read or echoed to the `TRCOUT` and `INLAB` files by subroutines `loadn`, `readi`, `readr`, `reecho`, `warray`, and `wiarr` that are called by the component input (`rcomp`) and restart (`recomp`) processing subroutines. The input and output echo of all input data has been consolidated in these six subroutines. SI- or English-unit symbols for real-valued input data variables are output echoed to the `TRCOUT` file

when namelist variable `iunout = 1` (default value). In addition to reading the input data, this subroutine also performs error checking; organizes the component data in memory; analyzes the system-model loop structure; and allocates the initial array space for the Control System, VESSEL, and part of the global arrays. The remainder of the space necessary for the global array variables is allocated in the initialization stage by the subroutine `init`.

The input line is echoed to the standard detailed ASCII output file and a warning message printed to that file, the message file, and the terminal when input errors are detected in subroutines at or below subroutine `input`. The error message is produced by a call to the subroutine `error`, with the first argument set to 2 to indicate a warning rather than a fatal message. By convention, the message (passed as the second argument) begins with the name of the subroutine processing the input line, bounded by single asterisks (e.g., `"*rpipe* inconsistent init & table power"`). When additional diagnostic information is necessary, including values of variables, direct WRITE statements are necessary. Pairing of this information to the messages from `error` requires three writes: one to the terminal (unit number in variable `itty` from module `Io`), one to the standard detailed ASCII output (unit number in variable `iout` from module `Io`), and one to the message file (unit number in variable `imout` from module `Io`).

Termination of input processing is flagged at two levels of severity. The lowest-level input routines (`loadn`, `readi`, `readr`, and `nxtcmp`, which are discussed subsequently) set the value of variable `ioerr` (located in the module `Io`) to one when an input error is detected. Subroutine `input` checks the value of `ioerr` after completion of the component-specific input from the TRACIN file (executed by calling the system-level subroutine `rdcomp`) and terminates if it is not zero. The presumption is that input errors are severe enough that it is not worth any processing of the restart file or checking of flow network connectivity. Higher-level routines (`input`, `rpipe`, `rtee`, etc.) flag problems for later termination by setting the variable `jflag` (contained in the module `BadInput`) to one. One exception to this behavior is subroutine `rcomp`, which uses a variable `jflagc` (contained in the common block `conck`) for the same purpose. The class of errors detected at these levels is presumed to be localized enough to make checking of flow network connectivity profitable. Subroutine `input` will terminate execution before returning if `jflag` or `jflagc` are not equal to zero.

Subroutine `input` initially calls subroutine `preinp` to read Main Data Card 1 and thus determines whether the TRACIN file is formatted as TRAC or free input. Control simply returns to subroutine `input` if the former option is selected. Otherwise, subroutine `preinp` reads the free formatted input data file, performs initial error checking, converts the input data to TRAC format, and writes the resulting data to file TRCINP. Subsequent execution of subroutine `input` proceeds in the same manner for both cases, with the TRCINP file accessed rather than TRACIN if the conversion was performed.

Subroutine `input` reads Main Data Card 2, using a call to low-level service subroutine `readi` for inputting integer card data, and subsequently calls `TRACall0` to allocate the required memory for the Title Cards to follow. The Title Cards then are read, and the minimum and maximum allowable fluid pressure and liquid/vapor temperatures and

the choked-flow multipliers are initialized. The namelist data are read with a standard Fortran construct, the namelist variable units are converted from English to SI units via calls to `uncnvts` if namelist variable `ioinp = 1`, and low-level service subroutine `reecho` is called repeatedly to echo the input to the TRCOUT file if variable `inopt` has been set to one on Main Data Card 2. Commonly used unit labels are also initialized, and subroutine `namlst` is called to check all namelist variables for valid values when these data are input by the user. The Solubility Parameters Card is read next, using the low-level service routine `readr` for inputting real card data if a dissolved material other than boric acid is to be traced by the solute tracker, which is indicated by inputting namelist variable `isolcn` as one. The remaining Main Data cards subsequently are read using calls to the low-level service routines `readi` and `readr`. Subroutine `input` then initializes the remaining fluid-equation-of-state constants by calling subroutine `seteos`. The component ID numbers in the TRACIN file are read into the real static array `scratch`, defined in module `Temp`, using a call to the low-level subroutine `loadn` (used to read card data in TRAC LOAD format) from the TRACIN input file. This portion of the REAL `scratch` array is converted and transferred to the static array `ig` (the integer global array defined in module `GlobalDim`) using a call to the low-level service routine `r2ii`, and the component numbers subsequently are placed in ascending order by calling subroutine `isort`, with array `ig` passed as an argument. Subroutine `input` then performs dynamic memory allocation for the HTSTR array `wp` and the boundary arrays `bd` and `vsi` by calling `allocWp` and `allocBoundary`, respectively (In Version 3.0, the `vsi` array has been superseded by the `vSign` array, which is part of the System Service package). Next, the System Service arrays `junCells`, `compSeg`, and `junComp` are allocated with direct use of the Fortran 90 `ALLOCATE` statement (note that the allocation for `junCells` is a conservative estimate because the number of TEE components is not known at this point in input). The countercurrent flow limitation (CCFL) model input data are read from the TRACIN file if `nccfl` (Main Data Card 6) is non-zero, with calls to the `loadn` subroutine. These data are also echoed to the TRCOUT file using calls to the low-level service routine `warray` (used to convert the namelist `ioinp` or `iolab` units to `ioout` units and write the respective array to the TRCOUT file as an echo of input data). The `loadn` and `r2ii` subroutines are called again, repetitively, to process the Material Properties Data, if so requested, by setting `nmat` to a non-zero value on Main Data Card 2. Dynamic memory allocation also is performed for Material Properties Data variable array `prtptb` by calling `allocPrtpb`. For the case with `stdyst = 3` and `stdyst = 4`, the `readi` subroutine is used to input the HPSS Data Card 1, `TRACAllo` is called to perform dynamic memory allocation for the derived-type variable `hps` (defined in module `HpssDat`), and Path Cards 1 through 3 are read using the `readi` and `readr` subroutines. Note that the Interactive Control Panel is not supported by TRAC-M. If a CSS steady-state controller is utilized in the calculation (`stdyst = 2` and `stdyst = 4`), this is followed by dynamic memory allocation of the derived-type variable `cssDat` (defined in module `ControlDat`), a read of the CSS-Controller Card with calls to `readi` and `readr`, a call to subroutine `unnumb` to assign the units label to the controlled parameters, and dynamic memory allocation for derived-type variable `cssTp` (also defined in module `ControlDat`) for secondary-side BREAK component numbers that need to have their pressure adjusted. The remaining Control-System-derived-type variables are next dynamically allocated, and subroutine `input` calls subroutine `rcntl` to read signal-variable, control-block, and trip control parameter data input from the

TRACIN file. Subroutine `rcnt1` reads this card data using calls to the `readi`, `readr`, and `loadn` subroutines. At this point in the calculation, the system-level data input processing is complete and subroutine input is ready to process the component-specific input data.

An infinite `WHILE` loop is used within subroutine `input` to read the component data until a card with the component type specified with "end" is read. The subroutine `nxtcmp` is first called to find the location of the next component if the TRCINP file is being accessed (this subroutine simply returns control to subroutine `input` if the TRACIN files is being utilized). The component Card 1 input then is read. The component's number, ID number, and title information is stored in the derived-type variable `genTab` [the fixed-length table (FLT) that is generic to all components, defined in module `Flt`]. The input-component type is converted to the internal TRAC CHARACTER variable-component-type representation using a call to subroutine `settype`, and the component number is stored in INTEGER global array `ig`. With the component type thus determined, either subroutine `rdcomp` is called to read the input data for 1D components or subroutine `rvssl` is called to read the 3D VESSEL-component data. During this process, the component number of VALVE components that are closed and not adjusted by CSS controllers is saved in variable `numvc` using a call to subroutine `GetValveTab`. Control parameter and component data not provided in the TRACIN file are retrieved from the restart-data file TRCRST by subroutine `rdrest`.

The subroutine `order` is called to arrange the signal variable, control block, and trip ID numbers in ascending order after all component information has been input. The list of closed VALVE components unaffected by CSS controllers, previously saved in variable `numvc`, then is used by subroutine `fbrcss` when called by `input` to determine all VALVE (type-2 controller adjusted for a desired pressure) and BREAK components that are hydraulically coupled to HTSTRs; this information is used for CSS type-4 and type-5 controllers. Subroutine `input` calls subroutine `srt1p` to sort through the 1D hydraulic components of the system model and group them by loops that are isolated from one another by VESSEL components; the `iorder` array is rearranged to reflect this grouping and provide a convenient sequence within each group for the component calculational order. The i^{th} element of the array `iorder` is the number of the component that is processed after the $i-1^{\text{th}}$ component but before the $i+1^{\text{th}}$ component. In earlier versions of TRAC, subroutine `srt1p` played a key role in setting up the network solution logic. However, this has been superseded by new logic at the start of subroutine `init` (see Section 2.3). Currently, `srt1p` determines only the component calculational order (also, currently, information from `srt1p` is used indirectly by the HPSS logic for VESSEL components—see the discussion on the old VESSEL matrix array `vmap` later in this section). For problems that contain more than one VESSEL component, the subroutine `vmcell` is called to convert a VESSEL-component cell number to a VESSEL-matrix cell number. Subroutine `allocvmap` is called to allocate memory dynamically for the VESSEL matrix array `vmap`. However, array `vmap` has been superseded by new network-solution logic that is set up in subroutine `init` (see Section 2.3); the only remaining use of `vmap` is to provide storage for VESSEL-matrix solution by the HPSS logic (subroutine `ihpss3`, called by `civssl`).

Finally, subroutine `assign` is called by input to define the component `POINTER` array, `comptr`, according to the order of the `iorder` array. The i^{th} element of array `comptr` is the starting location in the `ig` array of the component `iorder` (`i`) data block containing the component numbers.

2.2.1. 1D Component Input Processing with Subroutine `rdcomp`

The calling tree associated with subroutine `rdcomp` can be traced from the `NOMOD::SUBROUTINE rdcomp` entry in Appendix A. Subroutine `rdcomp` simply calls the component-specific input processing subroutines to read and process each component type. These routines have names that begin with the letter "r" followed by the letters of the component-type name. For example, the PIPE-component input processing subroutine is named `rpipe`. In addition to reading hydraulic and HTSTR-component data from the TRACIN file, these component-specific input processing routines also initialize the FLTs and variable-length tables (VLTs), define the `jun` array with component-junction connective information, and register the component with TRAC's system configuration to establish the intercomponent connectivity. Each 1D component-specific input processing subroutine calls subroutine `rcomp` to process input data common to 1D hydraulic components. All input data are echoed as output to the TRCOUT file.

The interface to a 1D component follows one basic pattern best seen in the PIPE component, with minor variations for boundary conditions and TEE-type components. Throughout this document, the PIPE component is used as the primary vehicle for discussing the lower-level subroutines in the calling tree. TEEs involve duplication of PIPE coding and special internal generation of boundary conditions at the internal TEE junction. Boundary conditions (FILL and BREAK) generate junction boundary information on the same cycles as a PIPE but perform relatively few other operations.

Input of the initial PIPE data is driven by `rpipe`, which is called by the subroutine `rdcomp`. Creation of a new component similar to a PIPE would require the addition of a call in `rdcomp` to process that component's input. As previously stated, the component type, component number, ID number, and descriptive title are obtained in subroutine `input` before it calls `rdcomp`. This information is passed to `rpipe` via the module for the FLT (`Flt`) as the variable elements `type`, `num`, `id`, and `title`. The order of the component in the TRACIN file, `cci`, is passed to the subroutine via the module `Global`. Subroutine `rpipe` obtains values of other scalar variables for the component from the ASCII input file, using the subroutines `readr` and `readi`, and stores the information in the derived-type-variable `pipeTab` (the VLT for PIPE-type components, defined in module `PipeVlt`). The `readi` and `readr` subroutines also echo this input to the detailed output file (TRCOUT). Subroutines `readr` and `readi` should be used for input of any scalar data for a component to maintain a consistent interface with the input file and its reflection to the output. Subroutine `rpipe` then calls subroutine `AllocGen1D` to perform dynamic memory allocation for the arrays that are generic to all 1D components (these arrays previously were accessed using the pointer tables `dualpt`, `hydropt`, `intpt`, and `heatpt` in TRAC-P). The arrays are stored in the derived-type variable `g1DAr`, which is declared in module `Gen1DArray`. Dynamic memory allocation for the additional arrays that are specific to a PIPE component is performed by subroutine

rpipe via calls to TRACallo. These array data are stored in the derived-type-variable pipeAr, which is defined in module PipeArray.

Subroutine rpipe then calls the subroutine rcomp to obtain array information on the geometry and initial state of the fluid for all cells (dx, vol, fa, fric, gravp, elev, hd, hdht, nff, lccfl, alpn, vln, vvn, tln, tvn, pn, pan, and, where appropriate, wfmfl, wfmfv, qppp, matid, twn, concn, and sn). This information is common to all 1D hydraulic components. Subroutine rcomp in turn uses the subroutine loadn to bring array data from the input and the subroutines warray and wiarn, respectively, to echo real and integer array values to the output. The subroutines loadn and warray also are used directly by rpipe to obtain additional array information. These subroutines (rcomp, loadn, warray, and wiarn) are the standard interfaces for reading and echoing array values from the input file and should be used for this purpose in any new component.

The input data specific to a PIPE component are read following the return from rcomp using calls to loadn and echoed to the output file with calls to warray (note that subroutine wiarn is used for this same purpose by other component-specific subroutines). Some processing of the input data also is performed by rpipe: scaling of table input is performed with calls to subroutine scltbl, subroutine unsvcb determines the units label and units-label subscript of the signal variable or control block associated with the PIPE component, units conversion is accomplished with calls to wmxymb, and linear interpolation of input data arrays is performed by subroutine linint0. These example actions are specific to the PIPE component, and other component-specific input subroutines (e.g., rvalve) will call different subroutines to process the input data into the desired format. The lower-level calling trees for the other TRAC component-specific input routines are given in Appendix A.

Subroutine rpipe and similar component-specific routines have one other important, but subtle, interface that must be replicated in new components. By supplying values to the jun array (and incrementing jun's current index jptr), rpipe supplies information to the system necessary to establish the order of calculation. The jun array is a doubly subscripted array, jun(4, *). The four values of the first index are defined in Table 2-1. The second index indicates the order in which the component junctions were encountered during input processing. The jun array is scanned after all input is processed when subroutine input calls srtlp (see Appendix A under TracInput::SUBROUTINE input) and the order of component processing is placed in the array iorder.

Formerly, the jun array saw much more use than in the current TRAC version; it has been superseded largely in recent TRAC versions by new network and intercomponent-communication logic. The new intercomponent communication has been implemented as a system service. A component must register its flow connections with the system services to permit correct intercomponent communications. In older versions of TRAC, this was accomplished within input and restart subroutines (rpipe, repipe, etc.) by filling in entries to the jun array. The current registration involves passing information to a junction cell data structure for each junction in a component with a call to subroutine

Junctions from a component input or restart subroutine (*rpipe*, *repipe*, etc.). In this context, registration is required for both standard intercomponent junctions and intracomponent junctions, such as the junction of a TEE side leg to the primary leg. Complete details on subroutine Junctions are given in Section 3.2.3.1. Each component also must register general information about the computational mesh segments that it contains, where a mesh segment is defined as a contiguous set of adjacent cells that are contained entirely within a component. This means that a PIPE, VALVE, PUMP, PRIZER, or PLENUM each contain just one mesh segment; a TEE contains two mesh segments (one each for the main leg and side tube); and the FILL and BREAK do not contain any mesh segments. A single call to subroutine SetSegment is used to establish the number of mesh segments for the current component. This currently can be either 0, 1, or 2, depending on the component type. If the number of mesh segments to be registered is > 0, then an appropriate number of calls to subroutine AddSegment1D is made (as described in Section 2.2.2, there is a related routine called AddSegment3D for VESSELS). More information on segment registration is given in Section 3.2.3.1.

Subroutines *rpump*, *rvalve*, *rfill*, and *rhtstr* determine if their component is being adjusted by a CSS controller when *stdyst* = 2 or 4 after reading the component data from the TRACIN file. If a CSS controller is being applied to the component, the desired hydraulic-parameter value is obtained from its specified location for type-1, -2, and -5 controllers. Type-3 CSS controllers get their desired fluid mass flow each timestep from their specified location in the plant model. For the i^{th} CSS controller (where $i = 1, 2, \dots, n_{\text{contr}}$), (Main Data Card 6) a signal variable with ID number $9900 + i$ is created to monitor the desired hydraulic-parameter value at its specified location and a PI-controller control block with ID number $-(9900 + i)$ is created to evaluate the adjustment of the component-action parameter. Signal variable ID numbers >9900 and <9999 and control-block ID numbers <-9900 and >-9999 are reserved for CSS-controller parameters defined internally by TRAC.

TABLE 2-1 FIRST INDEX OF THE COMPONENT-JUNCTION ARRAY *jun*

Index	Description
1	Junction number
2	Component number
3	Component type
4	Junction direction flag
	0 = positive flow is into the component at this junction (a <i>jun1</i> junction);
	1 = positive flow is out of the component at this junction (a <i>jun2</i> or <i>jun3</i> junction)

The k^{th} type-3 CSS controller, which adjusts either a PUMP or VALVE (where $k = 1, 2, \dots, n_{\text{contr}}$), requires a second signal variable with ID number $9900 + n_{\text{contr}} + k$ to monitor the pump-impeller interface or adjustable-valve interface fluid mass flow. The difference between the $9900 + n_{\text{contr}} + k$ signal-variable fluid mass flow and the $9900 + i$ signal-variable fluid mass flow drives the PI-controller control block adjustment of the pump-impeller rotational speed or the VALVE adjustable-interface flow-area fraction. A PI-controller control block is not defined for a type-3 CSS controller, which adjusts the in- or out-fluid mass flow of a FILL component, because the $9900 + i$ signal-variable fluid mass flow determines the FILL-component fluid mass flow directly for the next timestep. An absolute-value function control block with ID $-(9900 + i)$ of the $9900 + i$ signal variable's fluid mass flow is defined instead. It is this absolute-value fluid mass flow with a positive sign for outflow from the FILL and a negative sign for inflow to the FILL that is defined as the adjusted fluid mass flow of the FILL component.

The n_{contp} type-5 CSS controllers adjust the hydraulic-channel fluid pressure at the inner or outer surface of an HTSTR component. They each have 50 elements of the `cssDat`-derived-type variable reserved to save the ID numbers of all BREAK components that are hydraulically coupled to the adjusted HTSTR. The HTSTR's PI-controller adjusts the fluid pressure of those hydraulically coupled BREAK components. As previously stated, an ID list of VALVE components that are closed and not adjusted by a CSS controller is saved in array `numvc(n)` for $n = 1, 2, \dots, n_{\text{vc}}$ ($n_{\text{vc}} < 50$) by subroutine `input`. This ID list is used by subroutine `fbrcss` (called by `input`) to determine all BREAK components that are hydraulically coupled to the HTSTR. BREAKs separated from the HTSTR by these VALVE components that are closed and not adjusted by a CSS controller are not considered to be hydraulically coupled to the HTSTR component.

2.2.2. 3D Component Input Processing with Subroutine `rvss1`

Subroutine `input` calls the routine `rvss1` to input data from the TRACIN file that is specific to 3D VESSEL components. In addition to reading VESSEL input data parameters from the TRACIN file, this subroutine also initializes the FLTs and VLTs, reads VESSEL general-array and level data, registers the VESSEL component with TRAC's System Services to support intercomponent communication, and performs input data testing. As with subroutine `rdcomp`, this subroutine also uses the low-level subroutines `readi`, `readr`, `loadn`, and `warray` to input the data and echo it to the output file.

The basic geometric input data for the VESSEL-component VLT are first read into the derived-type variable `vessTab` (defined in module `VessVlt`) using a combination of `readi` and `readr` subroutine calls. With this information specified, subroutine `AllocVess` is called to perform dynamic memory allocation for the VESSEL general array data to be stored in the derived-type variable `vsAr` that is defined in module `VessArray` (this information previously was addressed with the VESSEL pointer table in TRAC-P). The subroutine `AllocVess3` subsequently is called to allocate memory dynamically for the 3D VESSEL level data that will be stored in the derived-type variable `vsAr3` (defined in module `VessArray3`). Data for the VESSEL general arrays specified in `vsAr` next are read from the TRACIN file using the `loadn`, `readr`, and

readi subroutines and echoed to the output file TRCOUT with the warray and wiarn subroutines.

The System Services junction array junCells is set up, and subroutine Junctions is called (once for each junction) to register the VESSEL's flow connections. System Services subroutines SetSegment and AddSegment3D then are called to register computational mesh segment information for the VESSEL. (Although its structure might seem to be somewhat discontinuous, a VESSEL is defined as having just one mesh segment.) Details on the VESSEL's System Services registration are given in Section 3.2.3.1.

Subroutine chksr is called to check the VESSEL source connections after the initial input processing from TRACIN is completed.

The VESSEL level arrays are specified in TRACIN on a level-by-level basis (data for the various arrays are grouped together for each successive level). The loadn subroutine is used to input a level's worth of data for each such array into a rank-one scratch array called scr. Subroutine rlevel is called for each of these chunks of level data to echo the input data from array scr to file TRCOUT and to call subroutine levelr, which stores the data into the (rank-three) vsAr3 array. (The VESSEL data structure is described in detail in Section 3.) Subroutine rlevel also checks the cfzv, cfrl, and cfrv arrays for negative values.

2.2.3. Component Input Processing with Subroutine **rdrest**

Subroutine rdrest opens file TRCRST and obtains restart data from the data dump corresponding to the requested timestep number of a previous calculation (as specified by variable dstep on Main Data Card 3 of file TRACIN). If the requested timestep number is negative, rdrest uses the last data dump available. If the requested timestep number is -99, the problem time from the last data dump is replaced by timet (Main Data Card 3), which is read from file TRACIN. The restart data initialize the signal variable, control block, trip, and component data that were not provided by the TRACIN file. Component data are read from the TRCRST file by calls to component restart processing subroutines. These subroutines have names that begin with the letters "re" followed by the letters of the component-type name. For example, the PIPE component restart processing subroutine is called repipe. These subroutines function in much the same way as the component input processing subroutines that begin with the letter "r". The restart data common to 1D hydraulic components are processed from the restart data using a call to subroutine recomp. Details on the structure of the dump restart TRCRST file are given in Sec. 2.5.3. All restart data are echoed as output to the TRCOUT file.

The calling tree associated with restart input can be traced from the entry NOMOD::SUBROUTINE rdrest in Appendix A. Restart input begins with communication of the lists of all system components iorder (Component List Card) and all components in the ASCII input deck (nbr) to rdrest via the module Global. Subroutine bfaloc is used to initialize the TRCRST file for processing, and the low-level service routine bfin is used to read all of the header information from the dump restart

file. An infinite WHILE loop then is entered to read the data for each dump timestep until the selected dump restart timestep is located. The majority of the data is read using the `bfin` subroutine, with the exception being that the Control System data must be read via a call to subroutine `CSRestart`. Upon initial entry, subroutine `CSRestart` performs the dynamic memory allocation for the temporary-derived-type Control System variables (defined in module `ControlDat`) that are only used for input processing of the `TRCRST` file. The restart data are read into these derived-type variables using calls to `bfin`. The infinite WHILE loop is exited once the desired timestep data is located, and the subsequent data are read with multiple calls to `bfin` and a single call to `CSRestart`. Subroutine `rdrest` then calls subroutine `recntl` to add the data stored in the temporary Control System derived-type variables to the information previously read from the `TRACIN` file (stored in the permanent Control System derived-type variables) and calls `CSFree` to deallocate the temporary variable memory. The subroutine `rdcomp` then enters another infinite WHILE loop to read the missing component data contained on the restart file (exiting the loop occurs when all component data had been read). The length of the tabular data `lcomp` (i.e., the sum of the `FLT`, `VLT`, and component-specific array parameter values) and the component number is then read using calls to `bfin`. If a missing component is found, subroutine `rdcomp` calls subroutine `GenTabRestart` to read the component `FLT` information into the derived-type variable array `genTab` (defined in module `Flt`). The component type is included in this data. With the component type defined, `rdcomp` calls the appropriate component-specific restart routine. The PIPE component will again be used as an example. Creation of a new component similar to a PIPE would require the addition of a call in `rdrest` to process that component's input.

Subroutine `rdcomp` calls `repipe` to process the PIPE component restart data with the component number and the pointer to the beginning of the junction array passed as arguments. The order of the component in the input processing, `cci`, is passed to the subroutine via the module `Global`. Subroutine `repipe` uses the subroutine `rstVLT` to read the PIPE `VLT` from the restart file into the derived-type variable `pipeTab` defined in module `PipeVlt`. It then echoes values of the `VLT` to the standard detailed output file using subroutine `reecho`. Subroutine `repipe` supplies values to the `jun` array (and increments `jun`'s current index `jptr`) in the same manner as `rpipe`, previously discussed. Similarly, subroutine `repipe` also calls subroutine `AllocGen1D` to perform dynamic memory allocation for the arrays that are generic to all 1D components (these arrays previously were accessed using the pointer tables `dualpt`, `hydropt`, `intpt`, and `heatpt` in `TRAC-P`). The arrays are stored in the derived-type variable `g1DArray`, which is declared in module `Gen1DArray`. Dynamic memory allocation for the additional arrays that are specific to a PIPE component is performed by subroutine `repipe` via calls to `TRACAllo`. These component-specific array data are stored in the derived-type variable `pipeAr` that is defined in module `PipeArray`.

Standard arrays required for restart of 1D flow (`dx`, `vol`, `fa`, `fric`, `grav`, `hd`, `nff`, `lccfl`, `wa`, `qppp`, `matid`, `alpo`, `alpn`, `vln`, `tln`, `pn`, `pan`, `wfmfl`, `wfmfv`, `aran`, `twm`, `tvn`, `alvn`, `chtin`, `vvn`, `arvn`, `arln`, `arevn`, `areln`, `rmvm`, `rvmf`, `vmn`, `bitn`, `hiv`, `hil`, `hig`, `higo`, `cifn`, `rhs`, `vvt`, `vlt`, `gamn`, `elev`, `chtan`, `alven`, `twan`, `twen`, `tcen`, and, when appropriate, `sn`, `concn`, and `qppc`) are acquired by a call to `recomp`. Those arrays that

would normally appear in an echo of the input data are printed by a call to `wrcomp`, which in turn uses the standard low-level routines `warray` and `wiarn` to write array values to the standard detailed output file. Actual input of either values or arrays of values in `repipe` or `recomp` is accomplished with the subroutine `bfin` rather than a direct Fortran READ statement because TRAC contains its own buffered I/O routines (`bfaloc`, `bfin`, `bfout`) for output to binary files. These buffered I/O subroutines should be used with any new component, as should standard routines to echo values to the standard detailed output. At a higher level, component-specific subroutines are used to process the input restart data into the desired data structure.

Subroutine `repipe` finishes with calls to System Services subroutines `Junctions` (two calls for a PIPE), `SetSegment`, and `AddSegment1D` (see Sections 2.2.1 and 3.2.3.1).

Recall that CSS-controller data are not written to the dump/restart file and so must be reinput by the TRACIN file if the CSS calculation is continued with a restart. The number of CSS controllers and their input parameters can be changed during a restart. Components defining the desired hydraulic parameter value for each CSS controller also need to be reinput using the TRACIN file.

2.3. Initialization

The calling tree associated with initialization can be traced from the entry `NOMOD::SUBROUTINE init` in Appendix A.

The initialization stage begins with the TRAC main-program calling subroutine `init`, which in turn calls four subroutines that set up data structures that are used for the solution of the governing flow equations: `GenJunInfo`, `SetSysVar`, `SetSysMat`, and `SetJunAvgPtrs`.

`GenJunInfo` processes information about system connectivity to fill out a data structure describing all junctions between hydrodynamic computational mesh segments. This data structure is contained in the derived-type array `junCells`. Subroutine `SetSysVar` establishes the structure of the sparse matrices associated with all flow equations, assigning unique system variable indices to each cell center in the system for pressure, stabilizer mass, and stabilizer energy equations. Subroutine `SetSysVar` also assigns a second set of unique variable indices to cell edges that is associated with the stabilizer velocities. This information is stored both in the component junction data structure (`junCells`) and in a data structure associated with mesh segments named `compSeg`. Subroutine `SetSysMat` assigns space needed for storage of equation information and creates indices needed to locate matrix coefficients, including those required to make substitutions between the fundamental equations and the network equations. Subroutine `SetJunAvgPtrs` establishes pointers needed to obtain edge-average quantities at junctions between mesh segments.

Subroutine `init` then calls subroutine `InitBDArray` to register into the System Service transfer tables the information that each hydrodynamic mesh segment needs from adjacent mesh segments to evaluate the flow equations; this relies on the System

Configuration set up by the component input routines (see Section 2.2) and `GenJunInfo`. `InitBDArray` sets up a pointer table for transferring information between TRAC's `bd` array and the generalized component array data structures. Detailed information is provided on this setup in Section 3.2.3.1. The elements of the `bd` array are described in Section 2.3.1.

Following the call to `InitBDArray`, subroutine `init` calls subroutine `TableTransAll` to populate the `bd` array with values that are needed to begin the component initialization.

Subroutine `icomp` is called next to perform the initialization of arrays and variables for each component type that is required by TRAC but is not read in directly from the files `TRACIN` and `TRCRST`. Subroutine `icomp` is a driver for component-specific initialization routines. For example, initialization of PIPE data is driven by the component-specific subroutine `ipipe`, which is called by `icomp`. Any new component would require creating an initialization routine and adding an appropriate call from `icomp`.

Following the component initialization by `icomp`, subroutine `init` makes another call to `TableTransAll` to ensure that the `bd` array is updated properly before the transient or steady-state calculation starts.

The `init` subroutine also initializes the graphics catalog using calls to the subroutines `CSSetLuIdx`, `xtvinit` and `xtvdr`. Subroutine `CSSetLuIdx` initializes the control block, signal, and trip unit label indexes, alleviating the need for further lookup. Subroutine `xtvinit` initializes graphics variables and opens the graphics file `TRCXTV`. The subroutine `xtvdr`, which is called by `init` with the argument `xmode` set to zero, simply calls the component-specific graphics routines (e.g., `xtvpipe`), with `xmode` again passed as an argument. The 1D component-specific graphics routines for the PIPE, PRESSURIZER, PUMP, TEE, and VALVE call the low-level service routine `xtv1d` to write the generic 1D component information to the graphics file. The remaining component-specific graphics subroutines write their information to the file directly (i.e., without using the low-level service routine). Each of the component-specific subroutines also calls the routine `PrintVarDesc` to generate the variable description graphics line when the argument `xmode` is zero.

The overall component-initialization subroutine `icomp` first calls subroutine `TRACallo` to allocate memory dynamically for the temporary pointer array `ijtrnPtr` that is required to process the PLENUM component boundary information (this array is deallocated upon completion of subroutine `icomp`).

Subroutine `icomp` then sets the values of arrays `jseq` (junction sequence) and `vsi` (velocity sign indicator), which are no longer used by TRAC. The functionality of arrays `jseq` and `vsi` has been included in the System Service logic (`vsi` has been superseded by array `vSign`).

Subroutine `icomp` next calls subroutine `cihtst` to initialize the data for HTSTR components, if present. Subroutine `cihtst` controls the initialization of all HTSTR components with calls to subroutines `irod1` and `irod`. Subroutine `irod1` initializes arrays that provide information on the location of hydrodynamic data for heat-transfer coupling. Subroutine `irod` initializes various power-related arrays that are not input.

A check is performed next to determine if either steady-state (`stdyst`) option 3 or 4 is selected to perform an initial estimate of steady-state temperature and velocity distributions. `TRACAL10` is called to allocate memory dynamically for the derived-type variable `hps` (defined in module `HpsDat`) when this is the case. The subroutine subsequently enters a DO loop that will cycle either two times for normal execution or three times if an HPSS initialization is to be performed. The status of the loop counter is stored in the variable `iin1` (named common block `e1vkf`) and can take on the values of 0, 1, or 2. Calls to the component-specific initialization subroutines (e.g., `ipipe`, which is described subsequently in greater detail) are contained inside this DO loop, but these calls occur only when `iin1` has values of either 1 or 2. For the initial loop with `iin1 = 0`, the subroutine `ihpss1` is called for each of the components in the `nloop` 1D hydraulic paths in the problem. This procedure replaces the phasic-temperature and velocity (and possibly pressure) values input for the 1D hydraulic components with the fluid mass- and energy-conserving values based on input-specified known or estimated thermal-hydraulic flow conditions along 1D-flow hydraulic paths of the system model. This procedure provides a better initial estimate of the thermal-hydraulic solution so that steady-state solution convergence is satisfied with fewer timesteps and less computational effort. This saves the TRAC user the effort of inputting such detail in the solution estimate defined by the component data in order to converge the steady-state solution more quickly with a better initial-solution estimate.

The basic work of component initialization takes place when `iin1 = 1`. Subroutine `ihpss1` again is called for each component in the `nloop` hydraulic paths to reevaluate the gas void fraction and phasic velocities donored from two-phase cells; this conserves the input-specified coolant inventory of the hydraulic loop. The 1D component-specific initialization routines also are called this time through the DO loop. Subroutine `setnet` also is called for each hydraulic path to provide the information needed to set up the network solution matrices. Subroutine `allocNet` then is called to allocate memory dynamically for the network solution (array `rnet`). Subroutines `setnet` and `allocNet` are largely obsolete; most of their functionality has been replaced by TRAC-M's modularized equation solution logic. Usage of the old network solution variables has been eliminated from all coding beyond initialization. Some of the network index information still is used at the end of `icomp` in a check to enforce the rule that all VESSEL connections in a given 1D loop must be to 3D faces of the same kind (all *r*, all *theta*, or all *z*), if the SETS numerics have been selected by the user for the VESSEL (using namelist variable `NOSETS`). This restriction will be eliminated (and with it the `rnet` data structure) once a planned parallel implementation of subroutine `Solver` has been completed. Before this loop-connection check, subroutine `civss1` is called from `icomp` to set up arrays for the 3D VESSEL initialization routine `ivss1`. This subroutine also calls subroutine `ihpss3` to perform HPSS initialization of the VESSEL if this option has been selected.

The third iteration of the loop with `iin1 = 2` functions in the same manner as when `iin1 = 1`, with the major difference being in the actions performed by the component-specific subroutines such as `ipipe`. These subroutines check the consistency of cell edge quantities at the junction, compute elevation changes across components, and convert loss coefficients to TRAC's specific form of friction factors.

As indicated above, a check for source connections that would couple VESSEL SETS predictor velocities in off-diagonal directions also is performed in subroutine `icomp`. This is necessary to ensure that the predictor and stabilizer velocities remain independent of one another for numerical stability at high fluid flows.

Finally, subroutine `icomp` deallocates memory for the pointer array `ijtrnPtr` and returns control to the calling subroutine `init`.

2.3.1. 1D Component Initialization with Subroutine `icomp`

The 1D hydrodynamic-component initialization routines have names that begin typically with "i" followed by the letters of the component-type name. For example, the PIPE component initialization subroutine is called `ipipe`.

Subroutine `ipipe` begins by obtaining values for indices to the `junCells` array (which was set up at the start of Subroutine `init`—see Section 2.3) for the current and adjacent component's junction cells and the `cco` index for each adjacent component (the `cco` index is described in Section 3.2). The four `junCells` indices then are used to obtain values for six intercomponent communications index variables that are contained in the derived-type variable `pipeTab` (`pipe VLT`, defined in module `PipeVlt`): `js1`, `js2`, `js1get`, `js1put`, `js2get`, and `js2put`. Each of these six variables provides a column index that is necessary for accessing the proper `bd` array elements (see Sections 2.3 and 3.2.3.1). Currently, `js1` and `js1get` have the same value, as do `js2` and `js2get`. (The numbers 1 and 2 indicate the current PIPE's left and right junctions, respectively.)

Junction-data consistency is checked using a call to subroutine `chkbd`. Subroutine `elgr` is called to compute FRICs and GRAVs from input form losses and elevations if these particular input options are selected using the namelist options `ikfac` and `ielv`, respectively.

Subroutine `ipipe` next calls to subroutine `junsol` twice (once for each junction) to set the elements `isollb` and `isolrb` of `pipeTab`, indicating the nature of the velocity calculation at the junction. A value of 0 from one of these variables indicates that the velocity is fixed by a FILL boundary condition. A value of 2 indicates that a BREAK is across the junction and that no other active component contributes to the momentum equation. A value of 1 indicates that another active component (PIPE, TEE, VESSEL, etc.) is on the other side of the junction but that the current component (this PIPE) performs the evaluation of the momentum equation. A value of -1 indicates that another component evaluates the momentum equation; that component appears before the current one in the order of computation. The same calls to `junsol` initialize the network index array `iou`. If the junction being processed is an active participant in the network solution (`isollb` or `isolrb` is +1 or -1), then the input value for that junction number is

placed in the appropriate location in `iou`. A later call to `setnet` from `icomp` converts these junction numbers to unique indices for the network junction variables associated with the component. (Note: Subroutine `setnet`'s functionality has been superseded by TRAC's new equation solution logic.)

Subroutine `volfa` is called to calculate volume-averaged cell flow areas and to perform several input data tests on valid flow-area configurations between cells and cell interfaces after the junction connection and component sequencing routines. Subroutine `comp_i` is called to initialize several variable arrays (e.g., tilde velocities).

Subroutine `ipipe`, as with all other existing 1D component initialization routines, uses a call to the subroutine `iprop` to initialize dependent fluid-state variables (density, internal energy, etc.), physical properties such as viscosity, and mixture properties such as the mean density. Actual computation of these properties is done or driven by the subroutines `thermo`, `fprop`, and `mixprp`, respectively. Information is communicated between `iprop` and these subroutines via their argument lists. Subroutine `iprop` communicates the information directly to the component-derived-type data structure `g1DAR` (defined in Module `Gen1DArray`) and should be used whenever possible for new components. If a replacement is constructed for a special component, care should be taken to understand and mimic the use of the variable `irest` (from module `Flt`) in `iprop`. Many properties (particularly macroscopic densities and energies) must be generated from more basic variables when a component is first input. However, when `irest = 1`, the component data are coming from a restart file, bringing values for many of these variables from the restart file, which must not be overwritten during initialization.

Subroutine `ipipe` then calls subroutine `CheckAcc` to determine if the friction factors for each junction cell opposing the current component (PIPE, in this case) are set according to the accumulator-phase separation model. If so, it copies the adjacent component's right-hand junction (`jun2`) `g1DAR` friction values to the current component's `g1DAR` locations.

Subroutine `ipipe` then calls subroutine `TimeUpGen1D` with argument `.TRUE.`; this has `TimeUpGen1D` copy the values of the generic arrays common to 1D components that are defined at old and new times from the new-time arrays into the old-time arrays (the new-time arrays have been set up to this point in `input` and `init`).

The last call in `ipipe` is to subroutine `TableTransComp`, which updates the `bd` array information that the current component provides to each of its neighbors. (A related routine, `TableTransAll`, is used elsewhere in the code to perform a similar service on a systemwide basis.) Note that for TEE components there is also a call to subroutine `jbd4` to update the boundary information directly at the TEE's internal junction (using the `bd4` array in the TEE data structure).

Initialization is the first stage at which boundary information is generated and passed. Currently in TRAC, the destination for component-boundary information transfer is still a form of the `bd` array that has been used since the earliest versions of the code. The

setup of the boundary transfer is driven by System Services subroutine `InitBDArray` (see Section 2.3), and the low-level hydrodynamics routines still access boundary information with the same references to `bd` as before. The actual `bd` array is defined in module `Boundary`; it is of rank 2 and is referenced in the low-level routines (such as the hydrodynamic routines) via rank-one dummy arguments `bd1` or `bd2`, which correspond to the left- and right-component junctions [the hydrodynamic routines are passed, via their argument lists, an appropriate column from `bd` ($72, 2 * n_{\text{jun}}$), where the first index specifies the type of data required, and the second index specifies the junction]. The data define the current solution state of the adjacent component across the junction and are evaluated at one of three possible space points: the edge of the mesh cell at the junction, the midpoint of that mesh cell, or the opposite-side edge of that mesh cell. References to `bd1` correspond to junctions `jun1` and `jun4` (the internal junction of a TEE component); \rightarrow to `bd2` correspond to either junction `jun2` or `jun3` (the external junction of the TEE-component side channel). The boundary data for the TEE internal junction are stored in a special array called `bd4` as part of the TEE data structure. The components of the `bd` array contain all of the geometry and fluid-state information necessary for one component to model flow across the junction from another using a first-order difference method. For the i^{th} junction, the elements of the array are as follows:

- `bd(1)` = adjacent-cell length
- `bd(2)` = adjacent-cell volume
- `bd(3)` = adjacent-cell, old mean density
- `bd(4)` = adjacent-cell, new, macroscopic-gas density
- `bd(5)` = adjacent-cell, new, macroscopic-liquid density
- `bd(6)` = junction-velocity-sign convention translation
- `bd(7)` = adjacent-cell, old void fraction
- `bd(8)` = adjacent-cell, old gas density
- `bd(9)` = adjacent-cell, old liquid density
- `bd(10)` = new-time liquid velocity one face past the junction *`vsign`
- `bd(11)` = new-time gas velocity one face past the junction *`vsign`
- `bd(12)` = TEE side-leg momentum equation coefficient
- `bd(13)` = TEE side-leg momentum equation coefficient
- `bd(14)` = adjacent-cell, old pressure
- `bd(15)` = adjacent-cell, new void fraction
- `bd(16)` = adjacent-cell, new gas density
- `bd(17)` = adjacent-cell, new liquid density
- `bd(18)` = new-stabilizer liquid velocity one face past the junction *`vsign`
- `bd(19)` = new-stabilizer gas velocity one face past the junction *`vsign`
- `bd(20)` = TEE side-leg momentum equation coefficient
- `bd(21)` = TEE side-leg momentum equation coefficient
- `bd(22)` = adjacent-cell new pressure
- `bd(23)` = junction, new liquid velocity *`vsign`
- `bd(24)` = junction, new gas velocity *`vsign`
- `bd(25)` = adjacent-cell, old surface tension

bd(26) = junction derivative of liquid velocity with pressure
 bd(27) = junction derivative of gas velocity with pressure
 bd(28) = adjacent-cell, new, macroscopic-liquid internal energy per volume
 bd(29) = adjacent-cell, new, macroscopic-gas internal energy per volume
 bd(30) = adjacent-cell, old gas viscosity
 bd(31) = adjacent-cell, old liquid viscosity
 bd(32) = junction flow area
 bd(33) = junction hydraulic diameter
 bd(34) = old-stabilizer liquid velocity one face past the junction*vsign
 bd(35) = old-stabilizer gas velocity one face past the junction*vsign
 bd(36) = adjacent component type
 bd(37) = adjacent component number
 bd(38) = adjacent-cell, old bit flags
 bd(39) = adjacent-cell, old, noncondensable gas density
 bd(40) = adjacent-cell, new, noncondensable macroscopic-gas density
 bd(41) = adjacent-cell, old, macroscopic-gas density
 bd(42) = adjacent-cell, old macroscopic-liquid density
 bd(43) = adjacent-cell, old macroscopic-gas internal energy per volume
 bd(44) = adjacent-cell, old macroscopic-liquid internal energy per volume
 bd(45) = adjacent-cell, void fraction from step before old time
 bd(46) = adjacent-cell, old, noncondensable, macroscopic-gas density
 bd(47) = adjacent-cell, old, noncondensable partial pressure
 bd(48) = adjacent-cell, new gas temperature
 bd(49) = adjacent-cell, new liquid temperature
 bd(50) = adjacent-cell, center gas velocity*vsign
 bd(51) = adjacent-cell, center liquid velocity*vsign
 bd(52) = new-time interfacial drag coefficient one face past the junction
 bd(53) = adjacent-cell, new bit flags
 bd(54) = gravity vector one face past the junction*vsign
 bd(55) = adjacent-cell, new solute concentration
 bd(56) = adjacent-cell, new mass-transfer term
 bd(57) = junction, old liquid velocity*vsign
 bd(58) = junction, old gas velocity*vsign
 bd(59) = adjacent-cell, liquid-specific internal energy
 bd(60) = adjacent-cell, gas-specific internal energy
 bd(61) = flow area one face past the junction
 bd(62) = junction, new liquid stabilizer velocity*vsign
 bd(63) = junction, new gas stabilizer velocity *vsign
 bd(64) = junction, old liquid stabilizer velocity *vsign
 bd(65) = junction, old gas stabilizer velocity *vsign
 bd(66) = junction, liquid wall friction input scale factor

bd(67) = junction, gas wall friction input scale factor
bd(68) = flow-area fraction of PLENUM faces
bd(69) = flag for "ell"-type TEE components
bd(70) = adjacent-cell, center x position
bd(71) = adjacent-cell, center y position
bd(72) = adjacent-cell, center z position

The current bd array is significantly different from older versions. Adjacent components no longer share a column of the bd array. Also, columns of the bd array now align with elements of the junCells array to give direct access to boundary data from the SysConfig data structure.

2.3.2. 3D Component Initialization with Subroutine **civssl**

Subroutine **civssl** assigns junction sequence numbers, performs HPSS initialization for $iin1 = 1$, and controls the remaining initialization of all 3D VESSEL components by calling subroutine **ivssl** for the subsequent passes. Subroutine **ivssl** performs analogous initializations for the VESSEL component, as does subroutine **ipipe** for the PIPE component. Clearly, because of the differences in the 1D and 3D databases, using many of the same low-level subroutines for initializing both component types is not possible.

Subroutine **ivssl** begins by setting up indexing for the VESSEL mass, energy, and momentum sources (i.e., loop connections) in the axial, radial, and azimuthal directions. The VESSEL mesh-cell side area and volume parameters are calculated directly within **ivssl**, without using a subroutine such as **volfa** used in the 1D case. This information is stored in the derived-type variable array **vsAr3** that is defined by module **VessArray3**. Subroutine **wlevel** is called to write this VESSEL level data to the file **TRCOUT**. Subroutine **Therm3D** is used to initialize fluid thermal-hydraulic properties in the VESSEL and calls subroutine **thermo** for the actual property evaluation, as was done in the 1D case via subroutine **iprop**. Subroutine **Fprop3D** also functions similarly to the 1D counterpart and calls **fprop** for the actual fluid property evaluations. However, mixture properties such as the mean density and solute concentration are evaluated directly in **ivssl**. The subroutine **initbc** is called to initialize VESSEL phantom cells and set some boundary conditions. Subroutine **rdzmom** defines reciprocal cell lengths for momentum cells (**rdxra**, **rdyta**, and **rdza**) and weighting factors for momentum cell averages or interpolation of cell-centered quantities. The input friction factors are divided by the hydraulic diameter in subroutine **iwall3**. The stabilizer equations for the VESSEL are initialized via a call to **mix3d** if these values have not been read from the restart file ($iirest = 0$) already. Momentum conservation is improved by setting up geometric-scale factors for coefficient velocities in cross terms of the momentum equation and for all velocities in diagonal vVv terms within subroutine **sclmom**. Subroutine **dvpscl** performs a similar function by initializing scale factors on the derivative of velocities with respect to pressure for each VESSEL level. After some additional initialization and checking, subroutine **ivssl** calls **setbdt** to set the values for the boundary of the first VESSEL theta cell equal to values for the last theta cells. Finally, subroutines **set3dbd** and **TableTransComp** are called to set up boundary

information at the VESSEL's junctions with 1D components (VESSEL sources); set3dbd uses array vsSrcAr as a target location for bd array pointers that point to the VESSEL data structure.

2.4. Prepass, Outer-Iteration, and Postpass Calculations

One complete timestep calculation consists of a prepass, outer-iteration, and postpass stage. These stages of the calculation are controlled by subroutines steady and trans calling subroutines prep, hout, and post, respectively. The names of the component-specific prepass driver subroutines end with "1", the names for the outer-iteration driver routines end with "2", and the postpass driver routines end with "3". These driver routines are identified for each component in Table 2-2. Each of these subroutines is contained in the associated component module. For example, the Pipe module contains the PIPE component prepass subroutine called pipe1, the outer-iteration subroutine called pipe2, and the postpass subroutine called pipe3. In the current version of TRAC, the Separator (SEPD) component is driven by subroutines sepd1, sepd2, and sepd3, which call tee1, tee2, and tee3, respectively (sepd1 currently is not used).

The prepass stage is responsible for calculating the control system state, much of the constitutive package (e.g., interfacial and wall-drag wall-to-fluid heat-transfer coefficients), and the solution of the stabilizer momentum equations. The outer stage calculates the interfacial heat transfer and then solves the basic (semi-implicit) equation set with a Newton iteration. The post stage solves heat conduction within metal structural elements and solves the stabilizer mass and energy equations.

**TABLE 2-2
COMPONENT-SPECIFIC DRIVER SUBROUTINES**

Component Type	Prepass	Outer	Postpass
BREAK	break1	break2	break3
FILL	fill1	fill2	fill3
PIPE	pipe1	pipe2	pipe3
PLENUM	plen1	plen2	plen3
PRIZER	przr1	przr2	przr3
PUMP	pump1	pump2	pump3
ROD or SLAB	htstr1		htstr3
SEPD or TEE	tee1	tee2	tee3
VALVE	vlve1	vlve2	vlve3
VESSEL	vssl1	vssl2	vssl3

The basic and stabilizer equations involve very different numbers of equations and generate two different matrix structures. As a result, two separate subroutines are used for the solution of global systems of linear equations. The more basic of these, Solver, operates on equations that are dominantly tridiagonal in structure (the stabilizer equations and pressure equation). The solution of the more complex linear system associated with the basic (semi-implicit) step is driven by subroutine BlockSolver. Subroutine Solver is described in the following subsection, and subroutine BlockSolver is described in Section 2.4.2 (on the outer-iteration logic).

Subroutine Solver: The interface to this subroutine is relatively simple. It uses the module Matrices; therefore, it has full access to this data structure. Only two arguments are passed:

- an abbreviated name (character string) for the array of independent variables; and
- an optional argument set to "factored" when the coefficient matrix already has been factored by a previous call to Solver (only applicable during the solution of the stabilizer mass and energy equations).

An example of use of subroutine Solver is the solution of the stabilizer mass and energy equations driven by subroutine post. The following code is inserted just before the end of the DO loop on ibks:

```

IF (ibks.EQ.1) THEN
....
CALL Solver ('arl')
CALL Solver ('arv')
CALL Solver ('arel','factored')
CALL Solver ('arev','factored')
CALL Solver ('ara', 'factored')
IF( isolut.NE.0) CALL Solver ('arc','factored')
ENDIF

```

This argument choice permits a single point within Solver for association of auxiliary arrays needed by solution methods and transfers knowledge of the data structure to a lower level for parallel methods based on distributed memory machines.

The arrays to be used in the actual solution are selected via pointer association. As an example, the current implementation contains allocatable arrays in module Matrices, such as

```

TYPE (sparseMatrix), ALLOCATABLE, TARGET :: al(:), ag(:)
REAL, POINTER, DIMENSION (:) :: arlS, arvS,      &
&   arelS arevS, araS, arcS, vvtS, vltS, arlRHS,      &
&   arvRHS, arelRHS arevRHS, araRHS, arcRHS,      &
&   vvtrHS, vltrHS
INTEGER, POINTER, DIMENSION (:) :: splitRowsC, splitRowsE
INTEGER, POINTER, DIMENSION (:) :: splitRows
TYPE (sparseMatrix), POINTER :: at(:)

```

```
REAL, POINTER, DIMENSION (:) :: rhs(:), ans(:)
```

Operations within Solver are on generic arrays such as `at`, `rhs`, and `ans`, which are associated by a call to subroutine `SetNetPointers` at the beginning of Solver. The pointers are associated based on the array name passed through from Solver's argument list. For example:

```
SELECT CASE (varname)
CASE ('arl')
  at => al
  rhs => arls
  splitRows => splitRowsC
CASE ('vvt')
  at => ag
  rhs => vvtS
  splitRows => splitRowsE
...
END SELECT
```

Following this initial decision on array usage, the solution proceeds as follows. The array `splitRows` is used to divide the 1D problem into a set of tridiagonal blocks. These block systems are solved and coefficient arrays are stored for later back-substitution. A substitution of these results is made into the splitting rows by subroutine `EqnSubstitute` to generate the network equation system, which is solved with calls to Linpack subroutines `sgfat` and `sgeslt`. If 3D components are present, the solution of the network equations involves the generation of coefficient arrays multiplying undetermined 3D variables. In this case, a section of Solver is used to substitute these network results into the 3D equations, and the 3D equations are solved for final values of 3D variables. The initial implementation of Solver uses the original TRAC-P Capacitance Matrix coding (subroutine `matsol`) to handle the solution of the 3D portion of the problem.

All three major equation solution stages just outlined use lower-upper (LU) factorization and store sufficient information so that the factorization need not be repeated. When subroutine `Solver` is called with the optional dummy argument "factored" present, processing jumps immediately to the back-substitution step of the 3D solution, then proceeds through back-substitution of the network equations and of the initial tridiagonal systems to obtain the final values for the variables (stored in `rhs`).

The current separation of the solution steps provides immediate opportunities for more parallel execution. In previous versions of TRAC, the contents of subroutines such as `femomx` (now `StbVelX`), `tf3ds`, and `stbme3` had to be executed after all similar 1D subroutines. Now these subroutines, with their reduced scope of activity (they no longer do the equation solution, but only evaluate their terms) can be executed in parallel with 1D subroutines. A planned later version of Solver will provide the opportunity for additional parallel computation within the solution process. The order of equation reduction will be altered so that operations on the sparse blocks associated with 3D components can be performed at the same time as those for the tridiagonal blocks associated with 1D components. Only the solution of the network matrix will remain as

a serial step. The greatly reduced amount of information required by the network matrix will make solution of the preceding steps more amenable to a distributed memory environment.

Each stage of the timestep-advancement calculation is described in the following sections.

2.4.1. Prepass Calculation

This stage of the calculation includes the stabilizer momentum equation solution and evaluation of various old-time quantities and other bookkeeping necessary at the beginning of each timestep. The prepass calculation uses the modeled-system solution state at the completion of the previous timestep (the beginning of the current timestep) to evaluate numerous quantities to be used during the outer-iteration-stage and postpass-stage calculations. The calling tree associated with the prepass is controlled by subroutine `prep` and can be traced from the `NOMOD::SUBROUTINE prep` entry in Appendix A. The prepass begins by evaluating the signal variables and the control blocks and determining the set status of all trips for the control procedure. Subroutine `trips` (not to be confused with subroutine `trip` that interrogates a trip's set status to decide on initiating specific consequences controlled by the trip) calls for these evaluations. Subroutine `prep` then loops over all of the components twice via calls to subroutine `prep1d` (which calls the 1D component-specific prepass subroutines), `htstr1` (for HTSTR component prepass processing), and `prep3d` (for 3D VESSEL components). The prep loop index number is communicated to these lower-level routines using the variable `ibks` (defined in module `OneDDat`). In the first loop through `prep`, each of the component-specific routines called by `prep1d` begins the prepass by moving its end-of-timestep values (its new-time values) from the previous timestep into the variable storage for its old-time values for the current timestep. Next, wall and interfacial friction coefficients are evaluated. The predictor stabilizer velocities (which are locally defined) and the setup of the stabilizer motion equations are evaluated. For components that require heat-transfer calculations, the prepass evaluates material properties and heat-transfer coefficients. The HTSTR component prepass is evaluated only in the first loop through subroutine `prep`. The prepass for HTSTR components can be more complex than that for 1D components. Besides calculating material properties and heat-transfer coefficients for both average and supplemental rods, the prepass evaluates quench-front positions and fine-mesh properties if the reflood model has been activated. Subroutine `prep3d` also is called from `prep` in the first pass to evaluate the 3D predictor motion equations and to set up the 3D stabilizer motion equations (`prep3d` also handles VESSEL constitutive quantities needed by the motion equations). The stabilizer motion equations subsequently are solved by calls to subroutine `Solver` (one call each for liquid and vapor) at the end of the `ibks = 1` pass. The second loop through the `prep` subroutine (`ibks = 2`) stores the results from `Solver` for the stabilizer velocities ("tilde" velocities) into the individual components' databases by again calling `prep1d` and `prep3d`.

Control System Details: Subroutine `trips` calls subroutines `svset`, `cbset`, and `trpset`. Subroutine `svset` uses beginning-of-timestep values of system-state variables to define the signal variables. Subroutines `cbset` and `conblk`, which are called by

subroutine `cbset`, evaluate control-block function operators. Subroutine `trpset` uses the current signal-variable and control-block values to determine the set status of `trips`.

State-Transition Method for Laplace Transforms: TRAC-M uses state-transition-method analytic solutions to evaluate the first-order lag, first-order lead lag, second-order lag Laplace-transform control blocks, and the first-order lag Laplace transform in the PI- and proportional-integral-derivative (PID)-controller control blocks. These analytic solutions were developed originally for TRAC-PF1/MOD2 (Version 5.4.02) to replace explicit numerics for evaluating the three Laplace transform control blocks and to replace semi-implicit numerics for evaluation of the PI- and PID-controller control blocks. The state-transition-method analytic solutions are unconditionally stable for all TRAC-M timestep sizes, whereas explicit numerics limits the TRAC-M timestep size to be less than the smallest lag constant for a numerically stable solution. For a given numerically stable timestep size, these analytic solutions are slightly more accurate than explicit and semi-implicit numerics. Implementing the state-transition method affected three subroutines: `cbset` was modified to set up for the three Laplace transforms to pass additional information in the form of two new actual arguments to subroutine `conblk` for the second-order lag Laplace transform and to evaluate the first-order lag function for the PI- and PID-controller control blocks with the analytic solution; `conblk` was modified to perform the analytic solutions for the three Laplace transform control blocks and to receive the new arguments from `cbset`; and `rcntl` was modified to perform additional input-error checking (on the Laplace-transform function constants). Details on the implementation and testing of these analytic solutions are given in [Ref. 4](#). Note that the original update that is described in [Ref. 4](#) used the TRAC-PF1/MOD2 version of the Control System database. The current Control System database is a direct mapping of that database onto Fortran 90-derived types; details on the current Control System database are given in [Section 3.2.2](#).

Stabilizer Velocity Solution Details: In older versions of TRAC, the order of the velocity variable array was effectively assigned by subroutine `srtlp` (called by `input`). The information needed to set up the network solution matrices was established by subroutine `setnet` during initialization. The current code requires storage of more information to link the component and systemwide views of the equations. Data are needed within each component for the matrix subscript corresponding to the stabilizer velocity at each cell face. These data are created by a call to `SetSysVar` (set system variables) near the beginning of subroutine `init` (see [Section 2.3](#).) and stored in the `compSeg` data structure as the variable values at the upper and lower bounds of each mesh segment and the increment (+1 or -1) in system variable index as the local cell index is increased. Network equations are selected during the matrix setup in subroutine `SetSysMat` (called by `init`). Indices of the matrix rows representing the network equations are placed into a dynamically allocated pointer array named `splitRowsE` ("E" for edge) contained in module `Matrices`. A similar array named `splitRowsC` marks network equations for matrices related to cell-centered variables.

The subroutine `SetSysMat` also stores indices for the off-band coefficients, which are stored in arrays with the type `sparseIndicesT` for 1D portions of the matrix and in arrays with type `vssMatIndT` for 3D portions. These indices provide information

necessary to recover actual coefficients from arrays of derived-type `sparseMatrixT` (1D) and `vssMatrixT` (3D). A detailed definition of these derived types is provided in Appendix C. For the current difference equations, two allocatable `sparseIndicesT` arrays and two allocatable `vssMatIndT` arrays are generated: one set for cell-edge-based equations (`aIndE` or `i3DE`) and one for cell-centered equations (`aIndC` or `i3DC`). Coefficients in 1D regions are stored in `sparseMatrixT` arrays `a1` for liquid and `av` for gas equations. Coefficients for 3D regions are stored in `vssMatrixT` arrays `a3Dl` for liquid-cell-centered equations, `a3Dv` for vapor-centered equations, `a3DlE` for liquid-cell-edge equations, and `a3DvE` for vapor-cell-edge equations. If necessary for future difference methods, this derived type can be cloned to produce types with more than one bandwidth or altered so that component `a` is an allocatable pointer. The choice of a fixed-dimension "bandwidth" for main-coefficient-component `a` was made based on the fixed structure associated with a given difference method and on timing results on the use of allocatable pointers within derived types.

Calculation of the coefficients and right-hand side of the stabilizer velocity equations is performed on a component-by-component basis with calls to `StbVel1D` (1D), and `StbVelx`, `StbVely`, and `StbVelz` (3D). These subroutines use the component data structure to obtain basic physical variables and store coefficient information directly into the systemwide equation data structure (in module `Matrices`). Solution of these equations is driven by calls to `Solver` from subroutine `prep` at the end of the first pass through its loop on `ibks`. Results are stored back into the component data structure by `bksmom` (1D), and `StbVelx`, `StbVely`, and `StbVelz` (3D).

Heat-Transfer-Coefficient Details: Subroutine `htstr1` initially calls subroutine `htstrv` to initialize the VESSEL-component hydrodynamic-cell arrays by setting HTSTR parameters to zero. This occurs before some of their elements have HTSTR parameters stored in them by subroutine `vssrod` after subroutine `core1` evaluates the HTSTR parameters. Subroutine `fltom` subsequently is called to transfer hydrodynamic data into the necessary HTSTR arrays; subroutine `core1` is called to evaluate heat-transfer coefficients, fine-mesh properties, and quench-front positions; and subroutine `fltom` again is called to transfer heat-transfer information back into the hydrodynamic database. From subroutine `core1`, subroutine `rfdbk` is called to evaluate reactivity feedback, and subroutine `rkin` is called to evaluate the point-reactor kinetics model.

Note: Subroutine `htstrv`: In future code versions, subroutine `htstrv` will be renamed and its call moved.

2.4.1.1. 1D Component Prepass Calculation with `prep1d`. The PIPE prepass is driven by subroutine `pipe1`, which is called by `prep1d`. The `prep1d` calling tree can be traced from the `NOMOD::SUBROUTINE prep1d` entry in Appendix A. As can be seen under the `prep1d` entry, the `pipe1` calling tree can be traced from `Pipe::SUBROUTINE pipe1`. Creation of a new component similar to a PIPE would require the addition of a call in `prep1d` to handle the prepass for that component. In the first pass, subroutine `pipe1` initially calls subroutine `savbd` to move boundary information for adjacent components into the derived-type component array `g1DAr` (defined in module `Gen1DArray`) and to move data from the last completed timestep into the old-time

arrays via a call to subroutine TimeUpGen1D. The first pass through pipe1 also results in a call to preper (see Appendix A, Gen1D Task::SUBROUTINE preper, for preper's call tree). Subroutine preper communicates directly with the component-derived-type data structure g1DAR and moves most information to lower levels via argument lists. Subroutine preper currently handles four types of tasks in its calls to other subprograms. The first of these is general bookkeeping, including calculation of the mass flow at the boundaries of the component via calls to flux and a call to volv to compute cell-centered velocities. The second task is calculation of basic physical properties, such as wall-friction coefficients (call to fwall), wall-metal properties (call to mprop), and heat-transfer coefficients for the heat-conduction model that is built into the PIPE component itself (call to htpipe), and interfacial drag coefficients (StbVel1D). The third task that is driven by preper is the setup for the stabilizer momentum equations, also delegated to StbVel1D (which is a modified version of the subroutine femom that is in older TRAC versions). The fourth task is the evaluation of any component-specific model. For instance, if the component type is a pump, then the subroutine pumpsr is called to provide pump momentum source terms. For a specific component, any or all steps may occur during a call to preper by its component prepass driver routine. Subroutine preper is a transition routine from the standpoint of data communication. It uses the systemwide and component-specific data structures, but passes on information on the state of the fluid through the argument lists of lower-level subroutines (more information on TRAC's data structures and internal data communication is given in Section 3.2). After preper is called, subroutine pipe1x is called to calculate the liquid volume discharged (qout), collapsed liquid level (z), and volumetric flow rate (vflow). Subroutine TableTransComp is called to establish the PIPE boundary conditions, and subroutine evfxxx is called to evaluate the xxx component action function. This completes the first pass (ibks = 1) of the PIPE prepass calculations.

Note: pipe1 First Pass. In future code versions, the logic flow in the first prep1d-driven passes will be modified.

On the second pass through all components (ibks = 2), subroutine bkmom stores the results from subroutine Solver's global calculation for the stabilizer momentum velocities into the individual 1D component's databases via a call to subroutine bksmom.

2.4.1.2. 3D Component Prepass Calculation with prep3d. ibks = 1: A new-time to old-time variable update initially is performed in the first pass by vss11 calling subroutine timupd. The subroutine dvpscl is called to initialize scale factors on the derivative of velocities with respect to pressure if water packing in the VESSEL has been detected. Subroutine vrdb is called to define velocities in the upstream radial direction for the inner ring of cylindrical VESSELS. The 3D interfacial shear is initialized with a call to ifset. Donor-cell weighting factors and mixture densities are initialized, vent-VALVE calculations are performed, and momentum source terms are defined within vss11. The boundaries of the first and last theta cells are equilibrated with a call to setbdt. Subroutine cif3 is called to evaluate the interfacial shear coefficients. Subroutine prefwd is called to evaluate the wall-shear coefficients. Subroutines StbVelx, StbVely, and StbVelz are called to set up the 3D stabilizer motion equations. Finally, boundary data are updated by calls to subroutines set3dbd and

TableTransComp, and (for VESSELS modeled in cylindrical coordinates) subroutine setbdt matches values for the first and last azimuthal-cell boundaries.

ibks = 2: Subroutines StbVelx, StbVely, and StbVelz are called to store the results from subroutine Solver for the 3D stabilizer velocities into the individual VESSEL databases. As on the first pass, boundary data are updated by calls to subroutines set3dbd and TableTransComp, and (again, for VESSELS modeled in cylindrical coordinates) subroutine setbdt matches values for the first and last azimuthal-cell boundaries.

2.4.2. Outer-Iteration Calculation

The hydrodynamic state of the modeled system is analyzed in TRAC-M by a sequence of Newton iterations that use full inversion of the linearized equations for all 1D hydraulic component loops and 3D VESSELS during each iteration. The convergence criterion is based on the calculated pressure changes (specifically, on the variable epso that is provided by user input). Throughout the sequence of iterations that constitute an outer calculation (each called an outer iteration within TRAC-M), the majority of the properties that were evaluated during the prepass and the previous-timestep postpass remains fixed. Such properties include wall (SLAB and ROD) temperatures, heat-transfer coefficients, wall- and interfacial-shear coefficients, stabilizer tilde velocities, and quench-front positions. The remaining fluid properties can vary to obtain a consistent hydrodynamic-model solution.

Subroutine hout controls the overall structure of an outer iteration, as shown under NOMOD: :SUBROUTINE hout in Appendix A, although the majority of the processing is handled by the subroutine outer. Subroutine hout contains an infinite loop that functions only to call subroutine outer and will exit only if the problem converges to the specified criteria; water-packing occurs as indicated by ipakon = 1 (set by either subroutine tf1ds3 for 1D components or subroutine out3d for 3D VESSEL components); the outer-iteration number oitno exceeds the input limit noitmx; or a velocity reversal occurs in a cell, as indicated by the logical variable lbckv being set true in either subroutine tf1ds3 or subroutine tf3ds3. For the case of water-packing, subroutine outer resets the outer-iteration-number oitno to zero and control returns all the way back to either trans or steady. These subroutines subsequently set iofail to zero and call hout again to retry the timestep. Subroutine out1d performs the timestep backup for water-packing for 1D hydraulic components via calls to BackUpP1en and BackUpGen1D, whereas subroutine vss12 calls subroutine backup for this function. The fluid thermodynamic properties also are reevaluated in this case. For the cases of a velocity reversal and excessive outer iterations, the number of outer iterations oitno is set to -100 within subroutine outer, the subroutine post is called from subroutine outer (the action of the post subroutine under this condition is described in a subsequent section), and program control is returned to either the subroutine trans or steady to select a new timestep size.

The outer subroutine calling tree can be traced from NOMOD: :SUBROUTINE outer. Subroutine outer loops over most component subroutines twice (no action is taken on HTSTRs), communicating the pass number through the variable ibks in module

OneDDat. In the outer stage, variable `ibks` has the values 0 and 1. Subroutine `outer` calls `out1d` and `out3d` for 1D and 3D components, respectively.

Subroutines `tf1ds`, `tf3ds`, and `tfpln` set equation terms for the individual mesh cells in 1D components, VESSELS, and PLENUMs, respectively: they generate coefficients for linearized mass and energy equations and store the coefficients in the `blockMatrixT` derived-type array named `blocks`. After the first loop over all components (`ibks = 0`), the solution of the full system of equations is driven by a call to subroutine `BlockSolver` from subroutine `outer`. `BlockSolver` provides a full solution for the pressure changes during the current iteration only. Final generation of the new time pressures, temperatures, and void fractions is accomplished during subroutine `outer`'s second loop over all components within the subroutines `tf1ds3`, `tf3ds3`, and `tfplbk` for 1D components, VESSELS, and PLENUMs, respectively. Details on subroutine `BlockSolver` are given in the following subsection.

Subroutine `BlockSolver`: Subroutine `BlockSolver` communicates entirely through module variables. It has no argument list and thus, none of the special pointer assignments that begin subroutine `Solver` (see Section 2.4). Solution begins with a block reduction in a loop over all elements of the derived-type array `blocks` (type `blockMatrix`). Pressure equations are isolated from the reduced system, using subroutines `PressCoef1D`, `PressCoef3D`, `PressCoefJun1D`, and `PressCoefJun3D`. The results of these operations are stored directly into the sparse matrix data structures used by subroutine `Solver`, and `Solver` then is called to obtain the pressure variations for the current iteration. Subroutine `DpJun` calculates the difference between iteration pressure changes in the two cells adjacent to each mesh segment junction (see Section 2.3) and stores them in `blocks%cDp`, ending the work performed by `BlockSolver`. Completion of the formal solution process involves substitution of pressure-change values into intermediate equations to obtain the next approximation for all independent variables (pressures, temperatures, and void fraction). These operations are performed by subroutines `tf1ds3` (for 1D components), `tfplbk` (PLENUM components), and `tf3ds3` (VESSEL components).

Variable `oitno` (named common `istat`) holds the iteration count; it is updated in subroutine `hout` and used to trigger special first-iteration operations in low-level routines such as `tf1d` and `vss12`. There are also tests on `oitno` as part of the logic for time-level weighting of convected mass and energy (the `xvset` logic). The iteration count also is used by subroutine `newd1t` as a contribution to the calculation of timestep size. As previously noted, it also takes on the function as a flag to the postpass containing a value of -100 in the event of an iteration failure. Subroutine `out3d` also temporarily sets this variable to a value of 2000 for problems with multiple VESSEL components and uses this value as a flag to control the operation of the `vss12` outer-iteration subroutine. The value is reset to the original iteration count following this processing.

2.4.2.1. 1D-Component Outer-Iteration Calculation with `out1d`. All 1D hydraulic components in a particular loop are handled by a single call to subroutine `out1d` in each pass. This routine calls the appropriate component-specific, outer-iteration subroutine.

Component-specific, outer-iteration subroutines have names that begin with the component type and end with the numeral 2, as previously illustrated in Table 2-2. For example, the PIPE-component, outer-iteration subroutine is called `pipe2`. Creating a new component similar to a PIPE would require adding a call in `out1d` to handle the outer iteration for that component. The `pipe2` outer-iteration routine's calling tree can be traced from `Pipe::SUBROUTINE pipe2` in Appendix A.

The outer-iteration subroutines for most of the 1D hydraulic components call subroutine `inner` to perform common functions. Subroutine `inner` obtains boundary information, calls subroutine `tf1d` to perform the appropriate hydrodynamic calculation, and resets the `bd` array by calling subroutine `TableTransComp`. Subroutine `tf1d` calls subroutine `tf1ds1` to set up the initial velocity approximations and their pressure derivatives for 1D components (first outer iteration only), subroutine `tf1ds` to solve the basic semi-implicit finite-difference equations, and subroutine `tf1ds3` to obtain the next approximation to the new-time pressures, temperatures, and void fractions. The BREAK- and FILL-component, component-specific, outer-iteration subroutines (`break2` and `fill2`, respectively) simply do boundary updating, with no call to `inner` performed. The component-specific outer-iteration routine for the PLENUM component, `plen2`, also deviates from the norm by calling the lower-level service routines directly with the actions performed by subroutine `tf1d` for the other 1D components replaced by those of subroutines `tfplen`, `auxplen`, and `tfplbk`.

2.4.2.2. 3D-Component Outer-Iteration Calculation with `out3d`.

Subroutine `out3d` functions in a similar manner to subroutine `out1d`, except that each 3D VESSEL component calls subroutine `vss12` to set up the basic semi-implicit 3D finite-difference equations and to update its independent variables, with calls to low-level service routines specific to 3D VESSEL components (e.g., `tf3ds1`, `tf3ds`, and `tf3ds3`). The calling tree from `outer` to `out3d` and lower is provided in Appendix A under `NOMOD::SUBROUTINE outer`.

Subroutine `outer` initially calls subroutine `out3d`, with both of the variables `ibks` and `iff3d` set to zero. This subroutine calls the component-specific, outer-iteration routine `vss12` for each VESSEL present in the problem. Subroutines `bakup` and `Therm3D` are called at this point if a water-packing backup is necessary, as indicated by `ipakon = 1`. For the normal case without water packing, the `vss12` subroutine initially calculates the VESSEL source terms and donor-cell contributions in the first iteration (`oitno = 1`). This calculation is followed by a call to `tf3ds1` to generate an estimate of the new-time velocities from the motion equations and evaluate the variation of velocities with respect to pressure. Subroutine `cella3` is called to evaluate cell-averaged quantities needed for the interphasic heat-transfer calculation, and `Htif3D` is called to perform these calculations. The boundaries of the first and last theta cells are equilibrated with a call to `setbdt`. The mass transfer to the VESSEL from 1D components is determined with a call to `flux`. Subroutine `tf3ds` then is called to set up the basic mass and energy equations, and `setbdt` is called a second time.

When `vss12` is called with variable `iff3d = 1`, subroutine `tf3ds3` is called to update the VESSEL's independent variables (after `BlockSolver` has been called). Subroutine

vssssr also is called to perform a steady-state, change-rate calculation for the VESSEL, if required. Subroutine setbdt also is called to equilibrate the first and last theta cells. When `ibks = 1`, VESSEL boundary data are updated with calls to `set3dbd` and `TableTransComp`.

2.4.3. Postpass Calculations

TRAC performs a postpass to solve the stabilizer mass and energy equations and to evaluate both fluid mixture properties and heat conduction in metal structures after the modeled-system hydrodynamic state has been evaluated by a sequence of outer iterations that converge. Subroutine `post` performs this postpass. This same subroutine also begins implementation of the timestep backup procedure, which is explained in detail in the next section. Subroutine `post` also can initiate a timestep backup if either the logical variable `lbpst` or `lbcyl` (defined in module `GlobalDat`) is set true during the postpass. Variable `lbpst` is set by (1) subroutine `bksstb` for 1D components, (2) subroutine `bkspln` for PLENUM components, and (3) subroutine `bkstb3` for 3D VESSEL components. This variable indicates that the backup occurred because the results of the `post` calculations either violate required stability criteria or exceed maximum allowed variations in hydraulic parameters. Logical variable `lbcyl` is set in subroutine `htstrp` and indicates that the backup is forced because heat-transfer energy conservation is not satisfied. When either of these conditions occurs, `post` returns control to the calling subroutine (either `trans` or `steady`), the number of outer iterations `oitno` is reset to -100 and `iofail` is set to 1, and `post` is called again to begin the timestep backup.

The calling tree associated with the postpass can be traced from the Appendix A entry `NOMOD: :SUBROUTINE post`. The postpass is driven by subroutine `post`, which loops through all of the hydraulic components three times. The postpass for the HTSTR component is treated with a single call to subroutine `htstr3` after all hydraulic components have been processed. As with `prep` and `outer`, the index for this loop is the variable `ibks` in module `OneDDat`. However, unlike these other subroutines that utilized infinite WHILE loops, the variable `ibks` is a DO loop index within subroutine `post`. This stage of the calculation performs the solution of stabilizer mass and energy equations when `ibks = 1` and 2, performs the solution of the conduction equations and evaluation of fluid properties (viscosity, specific heat, conductivity, surface tension, and heat of vaporization) when `ibks = 2`, and performs other minor computations necessary to complete each timestep (mass flows and mean velocity) when `ibks = 3`. The subroutine `post` receives information on the success of the outer-iteration solution through the variable `oitno`, as defined in the module `OneDDat`. On an iteration failure, as indicated by the number of outer iterations, `oitno` having a value of -100, subroutine `post` sets the variable `ibks` to two and thus skips the equation solution steps.

Subroutine `post` loops over all hydrodynamic components, calling driver routines specific to the 1D components that have the suffix "3" (e.g., `pipe3`) and subroutine `post3d` for VESSEL components. Equation setup is done at the component level by subroutines `stbme`, `StbME3D`, and `stbmpl`. At the conclusion of the `ibks = 1` pass, subroutines `StbMEJun`, `StbME3DJun`, and `Solver` are called to solve the global stabilizer mass and energy equations. On the second pass, values are stored in the

component data structure by subroutines `bksstb`, `bkstb3`, and `bkspln` for 1D, 3D, and PLENUM components, respectively.

2.4.3.1. 1D Component Postpass Calculation with `post`. Subroutine `post` calls each of the 1D hydraulic component-specific postpass routines directly (i.e., intermediate subroutines such as either `prep1d` or `out1d` are not used in the 1D postpass calculations). The call tree of the PIPE-component postpass routine, `pipe3`, is shown in Appendix A, starting under entry `Pipe::SUBROUTINE pipe3`. At the component level, boundary information is passed with the same mechanism as in the prepass.

`ibks = 1`: Setup of the stabilizer mass and energy equations for the 1D components is driven by a call to subroutine `constb`, which in turn calls `stbme`.

`ibks = 2`: Subroutine `pipe3` first calls subroutine `savbd` to retrieve `bd` array boundary conditions. Subroutine `efvxxx` is called to evaluate the `xxx` component action function, if required. Subroutine `poster` then is called to update the individual component's database with the results from Solver with a call to `bksstb`, along with several other tasks (including evaluation of fluid properties). Finally, the subroutines `evaldf1d` and `evaldf2d` are called in this pass to evaluate the absolute change in various hydraulic parameters (this information is used in the timestep-size logic). On a timestep backup condition, `poster` drives the restoration of all new-time variables to their original old-time values needed to restart the iteration via a call to `TimeUpGen1D`, but most other tasks in `poster` are suppressed. Subroutine `pipe3` skips the table evaluation of heat sources (call to subroutine `evfxxx`) for this condition.

`ibks = 3`: Subroutine `post` calls `poster` only in the final postpass. The final call to subroutine `poster` functions only to define the end-of-timestep mass-flow void-fraction-to-density ratios for the `bd` array in this pass.

2.4.3.2. 3D-Component Postpass Calculation with `post3d`. The intermediate subroutine `post3d` is called by `post` to perform the VESSEL postpass. The calling tree of `post3d` can be traced from the entry `VessTask::SUBROUTINE post3d` in Appendix A. As with subroutine `out3d` for the basic equations, `post3d` loops over the individual VESSEL components in the modeled system, in this case calling `vss13` for each VESSEL. Subroutine `post3d` calls `set3dbd` and `TableTransComp` to update VESSEL boundary information.

Subroutine `vss13` calls `StbME3D` to set up the mass and energy stabilizer equations and stores the results from the global calls to Solver from `post` (which concluded the `ibks = 1` pass over the hydrodynamics components in `post`) into the individual VESSEL components' databases with a call to `bkstb3`.

2.4.3.3. HTSTR-Component Calculation with `htstr3`. Subroutine `htstr3` controls the HTSTR postpass, as shown in Appendix A under the entry `RodTask::SUBROUTINE htstr3`. In the event of a timestep backup, the new-time values are reset to the values at the beginning of the timestep with calls to `TimeUpHS` and

TimeUpHS1. Under normal conditions, the postpass is performed by htstr3 first calling fltom to transfer data between the HTSTR and hydraulic databases. Subroutine core3 then is called to perform the HTSTR-component postpass calculations. In core3, subroutine frod is called to evaluate the temperature distribution and gap heat-transfer coefficients by calling subroutines rodht and gapht, respectively. Subroutine htstrp then is called to evaluate the HTSTR instantaneous power and energy in each ROD or SLAB element.

2.5. Timestep Advancement and Backup

The modeled-system solution state is updated to reflect the new-time (end of the previous timestep or beginning of the next timestep) conditions upon the successful completion of a timestep calculation (evaluated by the prepass, outer iteration, and postpass stages). This is accomplished at the start of the next timestep's prep stage and is handled on a component-by-component basis within their "1" subroutines, i.e., pipe1. During this step, all dual-time variables are updated by copying the values of the new-time variables into the old-time variables. The prepass, outer iteration, and postpass steps that follow during the next timestep then attempt to evaluate new values for the new-time variables for the end-of-timestep condition. This process is repeated as the problem time advances with each timestep.

Calculating a new timestep size occurs just before the prep stage and is controlled by subroutine timstp. Two types of algorithms, inhibitive and promotional, are implemented in subroutine newdlt to evaluate the next timestep size. The inhibitive algorithms limit the new timestep size to ensure stability and reduce finite-difference error. The promotional algorithm increases the timestep size to improve computational efficiency (by requiring fewer timesteps during a time interval). A new maximum timestep size is calculated based on each of the following conditions: the 1D and 3D material Courant limits; the VESSEL and total mass error limits; the outer-iteration count; the maximum allowable fractional change in gas volume fraction, temperature, and pressure; the diffusion number for heat transfer; and the maximum allowable fractional change in reactor-core power and adjustable-VALVE flow area. The new timestep size selected is the minimum imposed by the above conditions and the dtmax maximum timestep size specified by the user in the timestep data (Time Step Data Card 1). Subroutine newdlt is called by timstp to calculate each conditional maximum timestep size, except for those based on the reactor-core power level and VALVE flow-area adjustment. The reactor-core power-change maximum timestep size is evaluated by subroutine rkin during the prepass stage for HTSTR components, and the VALVE flow-area adjustment-change maximum timestep size is evaluated by subroutine vlvex during the prepass stage for VALVE components. During the outer-iteration stage, subroutine hout applies the lesser of these two maximum timestep sizes to define delt when it is less than the timestep size defined in subroutine newdlt.

TRAC-M will back up and try to reevaluate the modeled-system new-time solution state if a timestep solution is not completed successfully. A backup occurs either when the outer iteration does not converge (necessitating a reduction in the current timestep size) or when a flag indicating an extraordinary condition is activated. Either one will require

that the outer-iteration procedure be reevaluated. It is important to understand that there are two types of backups, one corresponding to each scenario. When the outer iteration fails to converge during the outer subroutine, the current timestep size is reduced and the calculation backs up to the start of the prep stage after the control-parameter evaluation. This is necessary because any variable calculated during the prepass that is dependent on the timestep size was computed for the original timestep size and not for the newly reduced timestep size. In addition, all new-time variables are reset to reflect their beginning-of-timestep values. This enables TRAC-M to begin again in the prep stage as for any other timestep calculation, except for having reduced the timestep size because of the backup. When the timestep requires one or more backups, the timestep size is halved for the first, second, and third backup, quartered for the fourth and fifth backup, and tented for backups thereafter. This backup process continues until either a small-enough timestep size is reached to allow outer-iteration convergence to be satisfied or the timestep size needs to be reduced below the `dtmin` minimum timestep size from the timestep data, wherein TRAC-M stops the calculation.

The second type of backup is initiated by a flag being set, signaling an extraordinary condition such as a water pack. This indicates that the outer iteration needs to be repeated to account for the extraordinary condition. TRAC-M resets any new-time variables that potentially have been evaluated incorrectly by the current attempt through subroutine `outer` with their old-time values, makes appropriate adjustments to prevent the extraordinary condition, and repeats the outer-iteration calculation. For this type of backup, the timestep size does not change, making it unnecessary to repeat the prep-stage calculation.

The difference between the two types of backups is that for a backup to the start of the prep stage, the timestep size is adjusted, all new-time variables are reset to their beginning-of-timestep values, and variables evaluated during the prep stage are reevaluated using the newly adjusted timestep size. For a backup to the start of the outer iteration, no change occurs in the timestep size and only new-time variables calculated during the outer iteration are reset to reflect their beginning-of-timestep values.

2.6. Output Processing

The TRAC-M program normally produces four different output files: TRCOUT, TRCMSG, TRCXTV, and TRCDMP. TRAC-M also may produce a TRAC-format input data file TRCINP and a labelled input data file INLAB. The TRCOUT-, TRCXTV-, TRCMSG-, TRACIN- and TRCINP-, and INLAB-file real-valued variables can have SI or English units based on the 0 (default value) or 1 value of namelist variables `ioout`, `iogrf`, `ioinp`, and `iolab`. SI- or English-units symbols can be output to the TRCOUT and TRCMSG files along with their real-valued variable values when namelist variable `iunout` = 1 (default value). The TRCDMP file real-valued variables have SI units. The output processing for each timestep during normal execution is performed via a call to subroutine `pstepq` from either subroutine `trans` or `steady`. Subroutine `pstepq` calls subroutines `edit` (large edit) and `sedit` (short edit) to write information to the TRCOUT file, subroutine `xtvdr` to generate graphics data in the TRCXTV file, and subroutine `dmpit` to write information to the TRCDMP file.

The TRCOUT file is in ASCII format and contains a user-oriented presentation of the calculation's input data and output results. During the input process, an echo of the input and restart data is output, and at selected times during the calculation, values of the current solution state of the modeled system are output. The TRCMSG file is in ASCII format and contains diagnostic messages concerning the progress of the calculation. File TRCINP is output only when input data TRACIN file is in the free format and file INLAB is output when namelist variable `inlab = 3` is input, as was previously discussed in Sec. 2.2. File TRCXTV is in both ASCII and binary formats; it provides data for XTV graphics. The TRCDMP file is a binary file designed to provide solution-state data for problem restarts by TRAC-M.

Note: XDR Format. For namelist-input variable `iogrf = 2`, the entire file TRCXTV is encoded in the XDR format, including the ASCII information that is written for `iogrf = 0` or `1`. For `iogrf = 2`, file TRCXTV is suitable for use with XMGR5; it is in SI units.

2.6.1. ASCII Output Processing with `edit`

Subroutine `edit` is the main driver routine for program ASCII output and calls subroutine `sedit` to write summary information for the time and subroutine `wcomp` to output information on the signal variables, control blocks, and components. Subroutine `edit` also outputs GSS convergence-test data and CSS adjusted/monitored data after all of the components have been processed. The first `edit` written to the TRCOUT file occurs during the first timestep after the prep stage via a direct call to subroutine `edit` from either `steady` or `trans`; however, all subsequent time edits are written after the post stage when `pstepq` is called from these subroutines. The calling tree associated with the output task can be traced from entry `NOMOD::SUBROUTINE edit` of Appendix A. Subroutine `wcomp` outputs general and control system data first, then invokes lower-level routines to output the solution state of each component. The component-specific `edit` routines, which have names that begin with the letter `w` followed by the letters of the component-type name, output the variable data that are important for that component to the TRCOUT file in an appropriate format for readability. For example, the PIPE-component `edit` routine is called `wpipe`, and the VESSEL-component `edit` routine is called `wvss1`. The component-specific subroutines use subroutine `ecom` to convert data to the requested output units (calls to `uncnvt`) and to write this data to the output file TRCOUT. The component-specific `edit` routines also output any additional data special to that particular component.

The dominant communications channels for the ASCII output `edit` are modules (systemwide and component-specific data structures) down to the calls to `uncnvt`, which relies on its argument list. The form of the output is very difficult to trace from the programming. However, the resulting output file is meant to aid the code user and is not intended as an interface to another program. As a result, no description of this file is provided here. This information can be obtained from the TRAC-M/F90 User's Manual (Ref. 2).

Subroutine `input` opens the TRCINP file and calls subroutine `preinp` to determine if the input data TRACIN file is in free or TRAC format. A free-format TRACIN file is read as ASCII data and parsed for numerical values to output the input data to the TRCINP

file in TRAC format. Then either the TRACIN file or the TRCINP file in TRAC format is read by the `readi`, `readr`, `warray`, and `wiarr` subroutines to process the TRAC-M input data.

When namelist variable `inlab = 3`, the `readi`, `readr`, `warray`, and `wiarr` subroutines output to file INLAB an input data echo of the TRACIN-file data with variable-name label comments in the free format. Outputting variable-name label comments between asterisks makes it a free-format file, even though the input data values are right-justified in 14-column fields. With a variable-name label above its scalar value or to the left of its array-element values, file INLAB provides input data with parameter variables that easily can be identified rather than that require the input data format description to define them. This makes the input data infinitely more readable in a standard form that all TRAC-M users become familiar with and that reduces input data defining errors. File INLAB is renamed TRACIN for use subsequently as the input data file to TRAC-M. The file-INLAB option also is convenient for converting SI- or English-units input data in the TRACIN file to English- or SI-units input data. This is done with namelist variables `inlab = 3`, `ioinp = 0` (SI) or 1 (English) for the TRACIN file, and `iolab = 1` (English) or 0 (SI) for the INLAB file, respectively.

2.6.2. Graphics Output Processing with `xtvdr`

Names of subroutines initializing and writing graphics information begin with the letters "xtv". However, this does not mean that the output is useful only to the graphics postprocessor named "XTV". This key program interface is well indexed and contains all of the information necessary to extract data for other data postprocessing, including translation for use by other graphics packages.

The graphics output is initialized with a call from subroutine `init` to `xtvinit` (see Appendix A, `NOMOD::SUBROUTINE init`), which sets the descriptive names of all array variables to be written to the graphics file TRCXTV and opens TRCXTV. Initialization continues with a call to subroutine `xtvdr`, using an argument of zero for the variable `xmode`. This value of the dummy argument `xmode` is propagated to lower-level subroutines through argument lists and triggers a mode that writes time-independent component information and index information about time-dependent variables to file TRCXTV (in ASCII format, unless namelist-input variable `iogrf = 2`).

The actual binary graphics output is driven by subroutine `xtvdr`, with a value of one passed to the dummy argument `xmode`. This call appears in subroutines `trans`, `steady`, and `pstepq`. The calling tree associated with the writing of graphics data can be traced from Appendix A entry `Xtv::SUBROUTINE xtvdr`. Subroutine `xtvdr` begins writing for a timestep by calling `xtvbufs` to output the edit time. It then loops over components calling the component-specific output subroutines (e.g., `xtvpipe`, `xtvtee`, `xtvvalv`, `xtvvs1`, and `xtvpln`). It ends with calls to subroutines to output heat structures (`xtvht`); signal variables, control blocks, and trips (`xtvcnt1`); and general problem parameters (`xtvgnpr`). Output of timestep data to file TRCXTV is buffered and at the lowest level written with the C function `fwrite`.

File TRCXTV provides data for X-TRAC-VIEW (XTV), a phenomena visualization package. (In addition, the XDR option selected with `iogrf = 2` supports XMGR5.) Subroutines `xtvinit` and `xtvdr` are called to create the XTV graphics catalog in file TRCXTV. The XTV graphics catalog contains information for setting up the component and variable visualizations. This includes the component name, type, connectivity, and geometry, as well as a list of available variables with their types. File TRCXTV contains timestep-edit information as arrays of Institute of Electrical and Electronics Engineers (IEEE) double-precision values. Each timestep edit contains the problem time, followed by all the variables described in the graphics catalog, in the order listed. The graphics-edit frequency is specified by user input, and the maximum size of file TRCXTV is internally defined (currently 750 Mb). This internal limit can be overridden through the use of the optional XTVTIN input file, which contains the maximum size of the data file in megabytes as an integer. If file TRCXTV reaches this limit, no further edits will be output and an error message will be written to the TRCMSG (message) file for each time edit that is not output. A description of the XTV file format is contained in Appendix H.

The exact contents of the file TRCXTV vary with the components in the problem and order of execution selected by TRAC for those components. The file begins with the first line from the TRAC input title information. After that, blocks of information follow for all components. Special information blocks exist for PLENUM and VESSEL components. Information for the flow components is followed by blocks for all HTSTR components, then those for the Control System.

As noted above, TRAC also contains an XDR interface for graphics output. The XDR interface is implemented via module `Xtv`. The Fortran interface to low-level XDR C routines is contained in module `CXtvXFaces`.

Note: XTV/XMGR5 Graphics Output Structure. Future versions of TRAC-M will use a new XTV/XMGR5 graphics system that is a fairly radical departure in terms of implementation from the current (Version 3.0) version. The new implementation is scheduled for February 2000. Foremost among the changes is the adoption of internal tables that store the component and variable information that is to be output. This change allows the variable setup and output information to be contained completely in a single call to a variable setup routine, instead of former implementations that required two or more synchronized lists to output variables, which allowed bugs to creep in when the lists became unsynchronized. The second major change in the graphics system is the adoption of graphics display templates. Each variable references a graphics display template, which contains all of the information needed to create a display of the variable in the postprocessor. Because each variable can have its own template, the restrictions on variables and the complexity of graphics components are significantly reduced. The remainder of Section 2.6.2 and its subsections (2.6.2.1 through 2.6.2.6) describe the new TRAC-M graphics system.

The new XTV/XMGR5 graphics file TRCXTV will utilize XDR encoding, which is provided through the Open Network Computing group's Remote Procedure Call Applications Programming Interface, generally referred to as the ONC RPC API. This provides the graphics system with platform-independent graphics output.

The new XTV/XMGR5 graphics system uses five Fortran source modules and two C source modules to accomplish its task. The data structures, allocation routines, and parametric settings are all contained within one module, and all of the component independent subroutines for initializing the tables are in a second module. All of the component implementations are in a third module, and the routines that output the information to the graphics file are in the fourth. The last Fortran module provides an interface specification to the C library. There are still only two routines that are called outside of the XTV/XMGR5 subsystem, `XtvInit` and `AddXtvDump`. Subroutine `xtvdr` will be replaced by subroutines `CreateXtvHeader` (for header output) and `AddXtvDump` (for data output).

The new XTV/XMGR5 system is initialized by a call to `XtvInit` inside the regular `init` sequence. `XtvInit` loops through all of the TRAC components, initializing the internal data tables, and then calls `CreateXtvHeader` to open the TRCXTV file and output the header section. Data are output to the graphics file through calls to `AddXtvDump`. In the initial implementation of the new XTV/XMGR5 system, `AddXtvDump` is called from subroutine `pstepq` for most dumps, as well as `steady` or `trans` at the beginning of a calculation.

Sections 2.6.2.1 through 2.6.2.6 describe the coding that implements the new XTV/XMGR5 logic to be incorporated into TRAC-M in February 2000.

2.6.2.1. Module `XtvData` (February 2000). Module `XtvData` contains three principal items: derived type definitions for the graphics component and variable tables, the parametric constants used as entries in those tables, and the routines associated with allocating those tables. Eight derived types create the graphics tables for XTV/XMGR5. The master type is `xtvCompT`, which is instantiated as `xtvCompList`. All other derived types, with the exception of `plenAux`, are nested inside the `xtvCompList` structure. Each derived type has a corresponding allocation routine and a routine to clear the elements of the derived type. The clearing routine is called by the allocation routine and need not be externally referenced.

The master variable definition derived type is `xtvVarT`, which stores a pointer to the variable, as well as name, length, and a few other generic items. Because Fortran 90 pointers are type and rank specific, there are actually eight pointers in the derived type and a variable `ptType`, which identifies which pointer is active; all other pointers are nullified.

2.6.2.2. Module `XtvSetup` (February 2000). Module `XtvSetup` contains the component-independent routines for initializing the internal graphics tables with component-specific information. Eight routines are targeted at initializing the component portions of the internal graphics tables. Eight routines also initialize the variable description tables. Module `XtvSetup` employs two generic interfaces to simplify the use of the internal routines for the component-specific routines. The most useful is `AddXtvVar`, which has an essentially identical interface for each type of variable, differing only in the rank and type of variable input. The second generic

interface initializes templates. Because templates have differing contents, depending on their internal rank, this is only a cosmetic name change.

2.6.2.3. Module `XtvComps` (February 2000). Module `XtvComps` contains the individual implementations for each of the TRAC component types, plus the master initialization routine `XtvInit`. Subroutine `XtvInit` is present in this module to avoid circular module dependencies between `XtvComps` and `XtvSetup`. Note that each of the individual component routines are called only once for each component instance; once initialized, the pointers contained in the data tables directly access the information for output. This means that variables cannot be output on the fly from the component routines. See the VESSEL variable `tw` for an example of calculating a variable for graphics output. New components will typically require only modifications to this module; most components will not require any changes to any of the remaining three graphics modules.

2.6.2.4. Module `XtvDump` (February 2000). Module `XtvDump` contains all of the routines that access the graphics file, albeit indirectly. All routines other than `AddXtvDump` and `CreateXtvHeader` are called by one of these two routines. This module is the only one to reference the C library and its interface specification module `CXtvXFaces`. Because of the pointer system used in the internal tables, this module does not reference individual component data tables, with the exception of the pressurizer (PRIZER) and VESSEL components, which require special adjustments before entering the main dump loop. The PRIZER component has the steady-state convergence adjustments removed before dumping and then reapplied after the graphics edit. See the routines `AdjPrizerDVars` and `UnAdjPrizerDVars` for more details on the PRIZER. The variable `tw` is not calculated during the normal course of the VESSEL, so routine `CalcVesselTw` is called by `AddXtvDump` before entering the main output loop.

2.6.2.5. Module `CXtvXFaces` (February 2000). Module `CXtvXFaces` provides the Fortran routines with the subroutine interface for accessing the C library. Not only does this provide error checking for the calls; it is necessary to get proper linkage under Fortran 90 syntax for arrays. Note that arrays passed to C are passed by supplying the first value of the array rather than being passed in traditional array format. This is necessary because Fortran passes a form of platform- and compiler- dependent array descriptor when passing arrays with explicit interfaces. The interface specification deliberately misleads the compiler as to what is being passed; be sure to check the C implementations for the true interface.

2.6.2.6. The XTV/XMGR5 C Library (February 2000). Because of the Fortran to C interface, the C Library functions are divided into two files: a set of TRAC specific routines for performing any special Fortran to C adjustments (`CXtvXdr.c`) and a set of routines that are used by TRAC and the postprocessors for common access (`xtvxdr.c`). All programs accessing the graphics file utilize the second file, `xtvxdr.c`.

2.6.3. Binary Restart File Processing with `dmpit`

The TRCDMP file is a structured binary file written with unformatted write statements. It contains sufficient data to restart the TRAC-M calculation at the problem time of a

data-dump edit. File TRCDMP consists of a general data section at the beginning followed by a series of time-edit blocks. A time-edit block is output at each edit time during a calculation. The number of time-edit blocks output to the file is determined by the dump-edit frequency specified by the timestep data. The last time-edit block is followed by a "EOF" to signify the end of file.

File TRCDMP is created by a sequence of calls to subroutine `dmpit`. As the main driver routine for dump file generation, subroutine `dmpit` outputs the dump-header data, calls the subroutine `CSDump` to output the control system data, calls the component-specific data-dump subroutines, and then calls subroutine `dhstr` to dump the HTSTR component data to the restart file. The names of the component data-dump subroutines begin with the letter `d` followed by the letters of the component-type name. For example, the PIPE component data-dump routine is called `dpipe`, whereas the VESSEL-component data-dump routine is called `dvssl`. All such dump routines ultimately use a set of five low-level, binary-dump routines that handles all output to the dump file; examples of their individual use are given in Appendix G. At the very lowest level are subroutines `bfout` and `bfoutn`; in many situations, these routines are first called by subroutines `bfoutis`, `bfoutni`, or `bfouts`. The 1D hydraulic-component data-dump routines call subroutine `dcomp` to output to the TRCDMP file data common to 1D hydraulic components and then output any additional data special to that particular component using individual low-level calls. The VESSEL-component data-dump routine `dvssl` also makes low-level calls to output general VESSEL arrays and calls subroutine `dlevel` to output level arrays.

The calling tree associated with the restart dump can be traced from Appendix A entry `NOMOD::SUBROUTINE dmpit`. The restart process is driven by `dmpit`, which uses `bfout` (and `bfouts`, which calls `bfout`) to write general variables and writes component-specific data with subroutines such as `dpipe`. The vast majority of communication throughout the chain of calls is via modules associated with systemwide and component-specific data structures. Switching from modules to argument lists as the means of communication occurs only at the low-level calls to the binary output routines.

Subroutine `dpipe` is typical of the component-specific dump routines and is very brief, using `bfout` directly to write pipe-specific arrays, and calling `dcomp` to dump information generic to 1D components. Subroutine `dcomp` drives the output of the FLT with subroutine `GenTableDump` and the VLT with subroutine `dmpVLT` (both use the low-level binary write routines for actual output). It then issues a series of calls to `bfoutn` to write the array data that is general to all 1D components.

It is important to remember, from the standpoint of the restart data interface, that all data are routed to the restart dump file via the binary-write set of routines. The structure of the dump file itself can be deduced fairly quickly by following the string of `bfout` (or `bfouts`) calls under `dmpit`.

In subroutine `dcomp`, the variable `lcomp` is calculated for each 1D hydraulic component and is the total number of all variable values output to the time-edit block for each component. This is the sum of the number of the variable values output by subroutine

dcomp and its calling routine. The number of any additional variable values special to a particular component and output by the component data-dump routine is reflected in the variable lextra. It is important to remember to increment either the variable lcomp or lextra accordingly when adding new component-variable values to the TRCDMP-file output.

A more detailed look at the dump logic is given in Section 3 and in Appendix G.

3.0. CODE ARCHITECTURE

TRAC's architecture comprises the organization of its subroutines into functional groups, the organization and membership of its various databases, and the interaction of the subroutines among themselves and with the databases. The logic of TRAC's calculational sequence is described in Section 2. In Section 3 we describe TRAC's organization according to the functions of its coding and the structure of and communication among its databases.

TRAC's architecture is very modular in terms of the organization of its coding, the organization of its databases, and the appearance of the databases to the code user. Section 3.1 gives an overview of TRAC's general coding structure in terms of its functional organization. Section 3.2 gives a detailed description of TRAC's data structures and the data communication within the code.

3.1. Code Structure

In an effort to strive for a code structure that minimizes the problems of maintaining and extending the code, the programmers originally developed TRAC as a modular code. This modularity manifests itself in two important ways. First, because TRAC analyzes nuclear-reactor systems that consist of specific component types (PIPES, VALVES, PUMPS, etc.), the code is written to utilize subroutines that handle specific component types. For example, calculations (and data) for a PIPE component are handled separately from calculations (and data) for a VESSEL component. Component-specific subroutines typically are called by driver routines that branch according to the component type to be calculated. The different TRAC components are described in greater detail in the TRAC User's Manual. The data structures for each component type are described here in Section 3.1.

Second, TRAC is written to be functionally modular; that is, each TRAC subprogram performs a specific function. Some low-level subprograms are used by all components, thereby strengthening this modularity (e.g., this is seen in the organization of the fluid-property routines). Another example of the code's functional modularity is found in the separation of the control-system logic into clearly defined subroutines.

TRAC comprises the following structural elements:

- PROGRAM TRAC;
- Fortran 90 modules that may both declare data and contain coding;
- Fortran 90 and C interfaces that encapsulate routines with a common functionality (but with differing argument lists, for example) within a generic name;
- procedures within the Fortran 90 interface(s);

- Fortran 90 subroutines;
- Fortran 90 and C functions;
- Fortran 90 blockdata; and
- include files (".h" files).

3.1.1. Fortran 90 Modules

A major advance with the development of TRAC-M is in the organization of the code into Fortran 90 modules. TRAC's modules provide the framework for the code's databases and the bulk of its coding. The data and coding are grouped into different modules according to their specific functionality. This functionality organization of the Fortran 90 modules further strengthens the original two-fold modularity built into TRAC: there are component-type-specific modules, and there are generic modules used by components in common.

A further advantage in TRAC's module structure lies in the dummy/actual argument checking that Fortran 90 can provide, thereby reducing the chances for errors; this argument checking is maximized in TRAC by carefully choosing the modules' use associations. In describing TRAC's database structure, Section 3.2 also indicates the module organization of the code's routines.

3.1.2. Description of All Structural Elements

Appendix B describes all of TRAC's structural elements: program TRAC and its modules, interfaces, procedures, subroutines, functions, blockdata, and include files. A brief description is provided of the purpose of each individual structural element and the name of the source file where it is located. For each module, lists are given of elements that are contained in the module, of other modules it uses, and of modules that use it. For each interface, the name of the module in which it is contained is given. For each subroutine, function, and procedure, the name of the module (if any) it is contained in, files it includes, modules it uses, other elements it calls, and elements that call it are given (for the functions, callers are not listed). For TRAC's blockdata, included files and its caller are given. For each included file, a cross reference of elements that include the file is provided.

3.2. Data Structure and Data Communication

3.2.1. Overview

This document describes the structure and internal communication of TRAC's database at three levels of detail: first there is this overview, which is followed by an expanded discussion later in this section. Finally, there are detailed examples for modifying the database in Appendix H. The purpose and use of the code's many individual database variables are described in Appendix C.

3.2.1.1. TRAC Databases. The TRAC database comprises a global database and several databases that are concerned with specific aspects of a calculation: the component-type database, the control-system database, two databases to support options for steady-state runs, and the radiation model database. TRAC's databases and related coding are organized into approximately 102 Fortran 90 modules, with each module having a well-defined function. Additionally, other variables that support a calculation are grouped into approximately 44 Fortran 90 header (".h") files; again, each .h file has a specific function. Some of TRAC's subroutines that are of a generic nature are not maintained in Fortran 90 modules, but rather are maintained as separate ".f" files.

TRAC's general global data include

- modules that implement global solution of the flow equations and related System Services modules that support intercomponent communication;
- array ag—A container array accessed by pointers that supports the equation solution;
- array ap—A container array accessed by a pointer for pressure variations; and
- four modules with miscellaneous functions, including declaration of arrays ig and rg and the component-index array compIndices, declaration of the pointers used by array ig, and initialization of various variables (including default values).

Note: Global Arrays ig and ag: In future code versions, arrays ig and ag will be removed.

The component-type database includes

- 1D hydrodynamic component types (PIPE, VALVE, etc.):
 - FLT, generic for all component types
 - VLT, specific for each component type
 - Array data generic for all 1D component types
 - Array data specific for each 1D component type
- Pseudo-1D boundary-condition-component types (BREAK, FILL):
 - FLT, same as for 1D
 - VLT, specific for each type
 - Array data generic for all 1D component types (subset of 1D)
 - Array data specific for each component type
- "0D" multiple-connection component type (PLENUM):
 - FLT, same as for 1D

VLT, specific for PLENUM
Array data generic for all 1D component types (subset of 1D)
Array data specific for PLENUM

- 3D hydrodynamic component type (VESSEL):

FLT, same as for 1D
VLT, specific for 3D component type
Array data for the 3D component type

- HTSTR-component type (ROD):

FLT, same as for 1D
VLT, specific for HTSTR component
Array data for the HTSTR-component type

- The control system database includes

Signal variables
Trips
Control blocks

- The steady-state databases include

The CSS
The HPSS initialization

The radiation model is related to the HTSTR database but has not yet been implemented in TRAC-M/F90.

3.2.1.2. Database Communication. TRAC's data and subroutines are grouped into many Fortran 90 modules that are organized and named according to their function. A module can both declare data variables and contain routines that operate on those variables. The use association of these modules is carefully arranged to provide logical and maintainable paths for data integrity, availability, and communication. TRAC provides information to its many subroutines through four mechanisms:

- At the most basic level, a module can declare data variables and contain routines that operate on those variables.
- Direct access to data in modules can be made available to a routine through use association.
- Argument lists are passed to lower-level (typically, data-crunching) routines that do not directly use the data they process.
- Interface routines are called to get (and sometimes overwrite) specifically requested data. Typically, these routines are used for communication among

different databases. The interface routines have access to the appropriate modules.

3.2.1.3. Fortran 90 Modules. TRAC's modules often declare data (or access data by Fortran 90 use association), performing a role in Fortran 90 that is similar to common blocks but that provides better data integrity. TRAC's modules often contain subroutines and functions that have a role specific to a certain task and that often involve data also in the same module or that are used by the module. An important aspect of TRAC's module use associations is in the explicit procedure-interface checking they allow, thus reducing the chance for programming errors. A complete list of TRAC's modules is given Appendix B; this includes a breakdown of the code's use and used-by associations.

Naming Convention: For the initial development of TRAC-M, each module is in a separate file. If the name of the module is Name, the corresponding file is called NameM.f (or, for Version 3.0, NameM.f90) (the coding uses Name).

3.2.1.4. Derived Data Types. Much of TRAC's database is organized into many Fortran 90-derived data types. These include derived types for the global database, component databases, control system, and steady-state models.

Naming Convention: The names of all derived data types that are defined in TRAC have a trailing "T". Often a variable that is declared to be of a derived data type is an array with the name of the derived type minus the "T".

examples -- derived-type naming:

(from MODULE PipeVlt)

```

TYPE pipeTabT                                     <<<--- Data-type name
      REAL(sdk) bsmass
      REAL(sdk) cpow
---
---
      INTEGER(sik) js2get
      INTEGER(sik) js2put
END TYPE pipeTabT

```

```

TYPE(pipeTabT), DIMENSION(maxComps)              :: pipeTab    <<<---

```

Declare variable

(from MODULE Gen1DArray)

```

TYPE g1DArrayT                                     <<<--- Data-type name
REAL(sdk), POINTER, DIMENSION(:)                :: driv
---
---
```

```

---
REAL(sdk), POINTER, DIMENSION(:) :: tcen
END TYPE g1DArrayT

```

!

```

TYPE (g1DArrayT), DIMENSION(maxComps) :: g1DAR <<<--- Declare variable

```

In this document, we refer to the entities that are declared within a Fortran 90-derived data-type definition as the “elements” of that derived data type (the term “component” also is seen in the literature, which we do not use, to avoid possible confusion with TRAC’s component types). We also use the standard term “array element” when referring to a specific item in any array [e.g., $x(i)$ is the i^{th} element of array x].

3.2.1.4.1. Global-Database-Derived Types. These are described in Section 3.2.2.1.2, with an overview, and in detail in Section 3.2.3.1 and Appendix C.

3.2.1.4.2. Component-Derived Types. TRAC employs Fortran 90-derived data types for the entire component database. This includes the 1D and 3D hydrodynamic components, the boundary-condition components, the PLENUM component, and the HISTR component. Table 3-1 lists the component data types: column 1 gives the name of the data type; column 2 gives names of variables that are declared to be of this type; and column 3 gives the module in which the data type is defined and the purpose of the type.

We are careful to distinguish between TRAC’s various Component types (the derived data types) and the individual Components that are in a given calculation. A typical Component data reference is

```

g1DAR(cco)%pn(j) ,

```

where cco is the specific component index, pn is one of the data arrays belonging to $g1DAR(cco)$ (in this case the new-time pressure array), and j is the index of the j^{th} mesh cell in component cco . Details on this construction and on the component-index logic are given below in the section on the component databases.

3.2.1.4.3. Control-System-Derived Types. Table 3-2 lists the derived data types that are defined for TRAC’s control system. The control-system variables with a name of the form $csrName$ are used for the dump/restart logic.

3.2.1.4.4. Steady-State Derived Types. The steady-state derived-types hold data for the CSS options and the HPSS initialization option. Table 3-3 lists these data types.

3.2.1.4.5. Radiation-Model Derived Types. The TRAC-P radiative heat-transfer model has not been implemented in TRAC-M; it has been retained in commented-out form.

**TABLE 3-1
COMPONENT DATA TYPES**

Type Name	Variables	Module and Purpose
genTabT	genTab	Flt—FLT's for all component types
array1DPtrT	array1DPtrs	Gen1DArray—data interface to 1D-component generic arrays
arrayNodeT	faceArs	Gen1DArray—data interface to 1D-component generic arrays
g1DArrayT	g1Dar	Gen1DArray—1D-component generic arrays
heatArrayT	heatAr	HeatDArray—Arrays for wall-heat-transfer model built into 1D components (not the HTSTR)
intArrayT	intAr	IntArray—Additional arrays generic to 1D Components
breakTabT	breakTab	BreakVlt—BREAK VLT
breakArrayT	breakAr	BreakArray—Arrays specific to BREAK
fillTabT	fillTab	FillVlt—FILL VLT
fillArrayT	fillAr	FillArray—Arrays specific to FILL
pipeTabT	pipeTab	PipeVlt—PIPE VLT
pipeArrayT	pipeAr	PipeArray—Arrays specific to PIPE
plenTabT	plenTab	PlenVlt—PLENUM VLT
plenumArrayT	plenAr	PlenArray—Arrays specific to PLENUM
prizeTabT	prizeTab	PrizeVlt—Pressurizer VLT
pumpTabT	pumpTab	PumpVlt—Pump VLT
pumpArrayT	pumpAr	PumpArray—Arrays specific to Pump
rodTabT	rodTab	RodVlt—HTSTR VLT
hsArrayT	hsAr chs	HSArray—Arrays for HTSTR (chs is pointer to hsAr, only to reduce statement-lengths)

**TABLE 3-1
COMPONENT DATA TYPES (cont)**

Type Name	Variables	Module and Purpose
sepdTabT	sepdTab	SepdVlt—Separator (Sepd) VLT
sepdArrayT	sepdAr	Sepd—Arrays specific to Separator (Sepd)
teeTabT	teeTab	TeeVlt—TEE VLT
teeArrayT	teeAr	TeeArray—Arrays specific to TEE
teeJcellT	teeJCellAr	TeeVlt—Stores TEE momentum source coefficients for "ell" and "i" configurations; acts as target location for bd array elements.
valveTabT	valveTab	ValveVlt—VALVE VLT
valveArrayT	valveAr	ValveArray—Arrays specific to VALVE
vessTabT	VessTab	VessVlt—3D VESSEL VLT
vessArrayT	vsAr	VessArray—3D VESSEL special arrays
vsSrcArT	vsSrcAr	VessArray—Provides a target location for bd array pointers that point to the VESSEL data structure.
vessArray3T	vsAr3	VessArray3—3D VESSEL fluid-mesh arrays

**TABLE 3-2
CONTROL SYSTEM DATA TYPES**

Type Name	Variables	Module and Purpose
csGlt	csG1 csrG1	ControlDat—global data; hold storage information and problem time.
csCPEDT	csCPED	ControlDat—control parameter evaluation-pass data
csSigT	csSig csrSig	ControlDat—signal variable data
csCBT	csCB csrCB	ControlDat—control block data
csULCBT	csULCB csrULCB	ControlDat—control block units labels
csULTRT	csULTR csrULTR	ControlDat—trip units labels
csULSET	csULSE csrULSE	ControlDat—signal-variable-units labels
csTripT	csTrip csrTrip	ControlDat—trip data
csTSET	csTSE csrTSE	ControlDat—trip signal expression signal data
csTCTT	csTCT csrTCT	ControlDat—trip-controlled-trip signal data
csTSFT	csTSF csrTSF	ControlDat—trip-set-point-factor table data
csTDPT	csTDP csrTDP	ControlDat—dump/problem termination data
csTSDT	csTSD csrTSD	ControlDat—trip-initiated timestep data

TABLE 3-3
STEADY-STATE DATA TYPES

Type Name	Variables	Module and Purpose
cssGlT	cssGl	ControlDat—global CSS data for storage allocation
cssDatT	cssDat	ControlDat—CSS input data
cssTPT	cssTP	ControlDat—CSS data for secondary-side break-pressure adjustment
hpsT	hps	HpssDat—HPSS dynamically allocated arrays

3.2.1.5. Data Precision. TRAC (approximately) requires the precision of an IEEE 64-bit word to perform the floating-point arithmetic for its hydrodynamics finite-difference solution. We say “approximately” because the code also runs with Cray 64-bit words and was originally developed on a 60-bit CDC 7600 computer. TRAC uses a few very large integers as special-purpose flags, but these all fit within 32 bits.

The precision of variables in TRAC is specified with the Fortran 90 KIND attribute. Variables `sdk` and `sik` are used to specify the precision of real and integer variables, respectively. Module `IntrType` has the following declarations for `sdk` and `sik`:

```

MODULE IntrType
  IMPLICIT NONE
!
!   These same definitions are repeated in all FUNCTION
!   declarations, for the NagWare F90 compiler
!
  INTEGER, PARAMETER :: sdk = selected_real_kind (13,307)
  INTEGER, PARAMETER :: sik = kind (10000000)
END MODULE IntrType

```

3.2.2. Databases

3.2.2.1. Global Data. Among the data that can be considered of a “global” nature in TRAC are

- four modules that contain a variety of data: `GlobalDat`, `GlobalDim`, `Global`, and `GlobalPnt`. Uses of the variables in these modules are described in Appendix C. More detail on modules `Global` and `GlobalPnt` (and the `ig` and `ag` arrays they support) is given in Section 3.2.2.1.1.

- The modules that implement the global flow equation solution and the System Services modules that support intercomponent communication for boundary information. These are described in Section 3.2.2.1.2, with an overview, and in detail in Section 3.2.3.1 and Appendix C.

3.2.2.1.1. Modules Global and GlobalPnt. Module Global declares high-level variables that are accessed throughout the code. These include an internal buffer for binary I/O, global arrays ig and ag, and index variables that are used to access the component database:

```

MODULE Global
!
! BEGIN MODULE USE
!   USE IntrType
!   USE GlobalDim
!
! Global Database
!
! Run Title
! REAL(sdk), POINTER, DIMENSION(:) :: RunTitle
!
! Buffer Container
! REAL(sdk), DIMENSION(2047*2) :: Buffer
!
! INTEGER(sik) ifreeIG,ifreeAG
! INTEGER(sik) igSize
! PARAMETER (igSize=10000)
! INTEGER(sik) ig(igSize)      <<<--- declare array ig
! REAL(sdk) ag(igSize)        <<<--- declare array ag
!
! replaces lenttl = lorder-ltitle in dmpit, set in input
! INTEGER(sik) lentitle
!
! component indices into as input
! INTEGER(sik) cci
! component indices as reordered in SUB assign
! INTEGER(sik) cco
!
! h1Ind - component index of first heat struct in tracin
! r1Ind - component index of first restart comp
! rh1Ind - component index of first heat struct in restart
!
! INTEGER(sik) h1Ind,r1Ind,rh1Ind
! INTEGER(sik) compIndices(maxComps)
!
! -----
!
! DATA h1Ind,r1Ind,rh1Ind/0,0,0/
! END MODULE Global

```

Arrays ig and ag are “mini-container” arrays; they store variables of a global nature, which typically are arrays themselves. The overall thermal hydraulics of ig and ag are

set in module Global by parameter igSize. Array ig holds integers, and ag stores reals. Arrays ig and ag are accessed by pointers, which are declared in module GlobalPnt:

```

MODULE GlobalPnt
!
! BEGIN MODULE USE
! USE IntrType
!
! INTEGER(sik) licvs,ldpmax,lijvs,lilcmp,liou,lisvf,livcon,      &
& livljn,ljout,ljseq,ljun,llcon,lloopn,lmab,lmcmsh,lmsct,      &
& lnbr,lnjn,lnsig,lnsigp,lnvcnl,lorder,lptbln,ltitle
!
! INTEGER(sik) lidpcv
! INTEGER(sik) lilprb,livlfc,livvto,livlto
! INTEGER(sik)      nmat,nvcell
!
END MODULE GlobalPnt

```

The specific uses of the various subarrays in ig and ag are described in Appendix C (the only use of array ag is for the pressure-variation array ldpmax, which is new to TRAC-M; it replaces TRAC-P array liitno).

Note: Global arrays ig and ag: In future code versions, arrays ig and ag will be removed.

The pointer offsets into ig and ag are set in subroutines input and icomp (note the Fortran comment below about the VESSEL component). Subroutine checksize is called to ensure that the pointer offsets into array ig do not exceed the value of parameter igSize:

```

SUBROUTINE input
---
---
---
lorder=1    <<<--- start at 1 (cf. TRAC-P)
---
---
---
lilcmp=lorder+ncomp
lnbr=lilcmp+ncomp
lm1dp=lnbr+ncomp
ldpmax = 1    <<<--- only ag pointer, replaces TRAC-P liitno
llcon=lm1dp+nhtst
ljun=llcon+ncomp
ljseq=ljun+8*njun
lmatb=ljseq+njun
lptbln=lmatb+nmat
ifreeIG=lptbln+nmat
CALL checksize('ig',ifreeIG,igSize,.TRUE.)
---
---
```

```

---
! these are re-evaluated for the VESSEL, leaving holes in ig
lijvs=ifreeIG
lnjn=lijvs+njun
licvs=lnjn+njun
liou=licvs+njun
ifreeIG =liou+njun
CALL checksize('ig',ifreeIG,igSize,.TRUE.)
---
---
---
it=nlt
l=licvs
livcon=lijvs
!
ljout=ifreeIG
lisvf=ljout+it+1
lnvcnl=lisvf+ncompt
---
---
---
lidpcv=linvs+nvcon+1
ifreeIG=lidpcv+nvcon
CALL checksize('ig',ifreeIG,igSize,.TRUE.)
---
---
---
! define the VESSEL matrix array pointers
!
liou=ifreeIG
ii= max(3,3*(ig(ljout+nloops)-1))
ifreeIG=liou+ii
CALL checksize('ig',ifreeIG,igSize,.TRUE.)
---
---
---
SUBROUTINE  icomp(comptr,jun,jseq,iorder)
---
---
---
livlfc=ifreeIG
livvto=livlfc+nvcon
livlto=livvto+nvcon
lilprb=livlto+nvcon
lidpcv=lilprb+nloops
ifreeIG=lidpcv+nvcon+1
CALL checksize('ig',ifreeIG,igSize,.TRUE.)
---
---
---

```

3.2.2.1.2. Flow Equation Solution and System Services. TRAC fully separates the evaluation of terms in the flow equations from the solution of the resulting system of linear equations. This provides a well-defined location for equation terms and eliminates

the need to generate this data for 1D components before evaluating the equations in 3D components (as was done in older code versions). TRAC also requires only one request at initialization to establish automatic information passing between components. This has been implemented as a system service, with sufficient generality to permit later use by higher-order and more implicit-difference methods.

There are currently four types of computational mesh in TRAC: 1D hydrodynamic, 3D hydrodynamic, 1D conduction, and 2D conduction. Many subroutines are associated directly with actions on a computational mesh (e.g., `tf1ds`, `tf3ds`, and `rodht`), and various array data structures are linked directly to the computational mesh (e.g., the contents of `Gen1DArray`, `VessArray`, and `HSArray`). TRAC also views components as collections of mesh segments and contains data describing the relationships between these mesh segments. In the current version of TRAC, capabilities of mesh-specific subroutines have been made more general to meet the needs of the range of physical components. Where possible, direct references to component types have been removed from mesh-specific subroutines and the necessary features are driven by the components in a more general way. It is our hope that future programming efforts will continue this effort to pull component-specific operations up to a higher level in the program or into component-specific subroutines called from a higher level.

Currently, the system services provide data transfer to a form of TRAC's original `bd` array, which is still used by the code's lower-level routines. However, the System Service logic is general enough to facilitate other uses.

Modules `JunTerms`, `Matrices`, `SetMat`, `SemiSolver`, `SysConfig`, and `SysService`, which comprise the databases (including derived types) and logic that implement the global equation solution and system services, are described in detail in Appendix C.

3.2.2.2. Component-Type Database. TRAC currently has 11 component types, which may be grouped into five functional categories. We list them by their names as they are generally found in the coding, giving more complete names in parentheses.

1D Hydrodynamic Components

Pipe (PIPE)
Prize (PRIZER, Pressurizer)
Pump (PUMP)
Sepd (SEPD, Separator)
Tee (TEE)
Valve (VALVE)

3D Hydrodynamic Component

Vess (VESSEL)

Pseudo-1D Boundary Condition Components

Break (BREAK)
Fill (FILL)

"0D" Multiple-Connection Component
Plen (PLENUM)

Power and Heat Conduction Component
Rod (HTSTR, Heat Structure)

Naming convention: The standard component-type abbreviations are found in subroutine names, module names, and derived data-type names (and corresponding instances of those data types). Minor variations in some component-specific subroutine names were inherited from TRAC-P. Subroutine and module names should start with an upper-case letter; data-type and variable names should start with a lower-case letter. Underscores are not used, but identifiable words within a name should start with an upper-case letter.

Parameter `maxComps`: As explained below, the specific data arrays of the component database are dynamically allocated at runtime as the input is read, but the various array declarations of the variables that are of TRAC's component-derived types are typically each dimensioned by parameter `maxComps`, which is set in module `GlobalDim`:

```
MODULE GlobalDim

  INTEGER(sik) maxComps <<<--- declare maxComps
  PARAMETER (maxComps=500) <<<--- set parameter maxComps
  --
  --
  --
END MODULE GlobalDim
```

For example, module `Gen1DArray` has the declaration:

```
TYPE (g1DArrayT), DIMENSION(maxComps) :: g1Dar ,
```

and module `PipeArray` has

```
TYPE (pipeArrayT), DIMENSION(maxComps) :: pipeAr .
```

Parameter `maxComps` also is used in the more restrictive declarations of the `FLT` derived-type array (used by all component types) and the component-index array (also used by component types); therefore, `maxComps` sets an upper limit on the total number of components in an input model: this limit includes all components of all component types in an input model taken together.

Component Indices `cci` and `cco`: All direct access to the component database is via one of two index variables, `cci` or `cco`, into the component-derived-type arrays, which selects a specific component in the calculation. Component-index `cci` is used at points in the calculation before subroutine `assign` is called from subroutine `input` (where the component reordering is done for the network-solution logic). Component-index `cco` is used after the call to `assign`. The reordered component indices are stored in array

compIndices, which has dimension maxComps. We refer to a specific component in a calculation as being "instantiated" when its data are being referenced directly via either index-variable cci or index-variable cco.

Component indices cci and cco and array compIndices are declared in module Global; compIndices is also dimensioned by parameter maxComps in module Global:

```

MODULE Global
---
---
---
!   component indices into as input
INTEGER(sik) cci           <<<--- declare cci
!   component indices as reordered in SUB asign
INTEGER(sik) cco         <<<--- declare cco
!
!   h1Ind - component index of first heat struct in tracin
!   r1Ind - component index of first restart comp
!   rh1Ind - component index of first heat struct in restart
!
INTEGER(sik) h1Ind,r1Ind,rh1Ind
INTEGER(sik) compIndices(maxComps) <<<--- declare compIndices
---
---
---
DATA h1Ind,r1Ind,rh1Ind/0,0,0/
END MODULE Global

```

The typical use of array compIndices is shown in the following examples. In the first example, TRAC is looping over an input model's individual 1D hydrodynamic components, instantiating each of them in turn; in the second example, the value .TRUE. or .FALSE. is passed to a routine with logical flag reordered that accesses data from a noninstantiated component.

examples -- use of array compIndices:

```

-----
SUBROUTINE out1d(imin,imax,jflag)
---
---
---
!   controls outer calculation for one-thermal-hydraulical
!   components.
---
---
---
DO icmp=imin,imax
  cco=compIndices(icmp) <<<--- instantiate component (set cco)
  icme=icme+1
!

```

```
IF (.NOT.(genTab(cco)%type.EQ.breakh.OR..... <<<---use cco
to access specific component
```

```
SUBROUTINE GetGenTable(name,compInd,ival,rval,reordered)
----
----
----
LOGICAL reordered
----
----
----
ordInd = compInd
if(reordered) ordInd = compIndices(compInd) <<<--- reordered
!
IF (name.EQ.'lenvlt') THEN
ival=genTab(ordInd)%lenvlt
ELSEIF.....
-----
```

HTSTR-Component Index: The HTSTR indices start after the end of the indices for the hydrodynamics components:

```
SUBROUTINE htstr1
----
----
----
! first loop over all heat structures
!
DO icmp=1,nhtstr
cci=icmp+ncomp <<<--- add ncomp to obtain HTSTR index
cco=compIndices(cci)
----
----
----
```

Component Modularity—Logic and Data: There is much commonality and modularity in the logic (with respect to both data and code) across all of TRAC's 10 component types. The 1D component types are similar in data and coding organization to the other types, and they are very similar among themselves. We begin by using the PIPE component type as the basic example. Any information specific to the other types is given in the sections immediately following.

The component FLTs and VLTs contain data elements that are used by a specific component as a whole (e.g., a variable having the component's type and other variables having its ID number and total number of mesh cells). The FLT and VLT can include arrays, but we reserve the term "array data" for data elements specific to an individual mesh cell in a component (the volume of the J^{th} cell, its void fraction, pressure, etc.). The

1D component-type arrays are mostly 1D in the Fortran sense (rank 1), but they can be of higher rank as well (such as the NCELLS x NODES wall-temperature arrays `tw` and `tw`). We have tried to be specific when using the term "1D" as to which meaning is intended.

3.2.2.2.1. 1D-Hydrodynamic-Component Types (PIPE, etc.). FLT, array `genTab`:
 The data elements in the FLT are the same for all 11 component types; FLTs for specific components in the input deck, for all 11 component types, are stored in array `genTab`, which is of derived data-type `genTabT` and dimension `maxComps`. All FLT-related logic is treated by module `Flt`:

- definition of derived data-type `genTabT` (declaration of its elements):

```

TYPE genTabT
  REAL(sdk) htlsci
  REAL(sdk) htlsco
  ---
  ---
  ---
  INTEGER(sik) nodes
  INTEGER(sik) num
  REAL(sdk) type
END TYPE genTabT
  
```

- declaration of array `genTab` to be of type `genTabT` and dimension `maxComps`:

```

TYPE(genTabT), DIMENSION(maxComps) :: genTab
  
```

- Parameterization of the total length of data-type `genTabT` (for use by the dump/restart logic):

```

INTEGER(sik) genDumpSize
PARAMETER (genDumpSize=24)
  
```

- Subroutine `GenTableDump` adds an individual component's variables that are stored in array `genTab` to the dump/restart file.
- Subroutine `GenTableRst` reads a component's `genTab` data from the dump/restart file.
- Subroutine `GetGenTable` accesses certain `genTab` data of a component other than the current instantiated component (e.g., data that an `HTSTR` component needs from a hydrodynamics component that is coupled to its surface).

Appendix H gives extensive examples of the data structure and coding in module Flt. Appendix C gives descriptions of the use by TRAC of all the individual data elements in genTabT.

VLT, arrays "comp_type" "comp_type"Tab: Each TRAC component type has a set of data that is used by a given component as a whole (not on a mesh-cell basis), where the variables are *common to all components of a given type*. These data sets are called the component VLTs. They comprise mostly scalar variables that are defined by the elements of one of a set of derived data types; there is a separate derived type for each of the 10 component types. The 11 VLT data types currently defined in TRAC are

```
breakTabT
fillTabT
pipeTabT
plenTabT
prizeTabT
pumpTabT
rodTabT
sepdTabT
teeTabT
valveTabT
vessTabT
```

We will refer to these 11 derived data types as a group by the term

"comp_type"TabT.

Eleven arrays, one for each component type and each of dimension maxComps, are declared to store the "comp_type"TabT (VLT) data for the specific individual components in the input deck. The array for each component type is declared to be of the corresponding "comp_type"TabT derived data type and given the name "comp_type"Tab. For example, the code has the declaration

```
TYPE(pipeTabT), DIMENSION(maxComps) :: pipeTab
```

to store VLT data for all the individual PIPE components in the input deck. All VLT-related logic for a component type is handled by a module that is specific for that type, which has a name of the form

```
MODULE "Comp_type"Vlt.
```

In Version 3.0 there are 11 "comp_type"Vlt modules, i.e.,

```
BreakVlt
FillVlt
PipeVlt
PlenVlt
```

```

PrizeVlt
PumpVlt
RodVlt
SepdVlt
TeeVlt
ValveVlt
VessVlt

```

The logic in each module "comp_type"vlt comprises

- definition of derived data-type "comp_type"TabT (declaration of its elements):

```

TYPE pipeTabT
  REAL(sdk) bsmass
  REAL(sdk) cpow
  REAL(sdk) eninp
  REAL(sdk) epsw
  REAL(sdk) fl(2)
  REAL(sdk) fv(2)
  REAL(sdk) houtl
  ---
  ---
  ---
  INTEGER(sik) js2get
  INTEGER(sik) js2put
END TYPE pipeTabT

```

- declaration of array "comp_type"Tab to be of type "comp_type"TabT and dimension maxComps:

```

TYPE(pipeTabT), DIMENSION(maxComps) :: pipeTab

```

- Parameterization of the total length of data type "comp_type"TabT (for dump/restart):

```

INTEGER(sik) pipeDumpSize
PARAMETER (pipeDumpSize=60)

```

- Subroutine "Comp_type"TableDump adds an individual component's variables that are stored in array "comp_type"Tab to the dump/restart file:

```

Subroutine PipeTableDump(ordInd, caller)

```

- Subroutine "Comp_type"TableRst reads a component's comp_type"Tab data from the dump/restart file:

Subroutine PipeTableRst (ordInd, caller)

There are two additional subroutines that are contained in only some of the "Comp_type"Vlt modules:

- Subroutine Get"Comp_type"Tab accesses certain "comp_type"Tab data of a component other than the current instantiated component (e.g., when adjusting the power for the heat structures in a neutronics calculation group):

Subroutine GetRodTab (name, compInd, ival, rval, reordered)

- Subroutine Set "Comp_type"Tab sets (overwrites) certain "comp_type"Tab data of a component other than the current instantiated component (e.g., when adjusting the power for the heat structures in a neutronics calculation group):

Subroutine SetRodTab (name, compInd, ival, rval, reordered)

In Version 3.0 there are Get"Comp_type"Tab subroutines for the ROD, TEE, VALVE, PUMP, AND VESSEL component types; there is a Set"Comp_type"Tab subroutine only for the ROD type.

The component-type routines for dump and restart, subroutine "Comp_type"TableDump and subroutine "Comp_type"TableRst, are called by generic driver subroutines dmpVLT and rstVLT, respectively, which branch according to the component type. Subroutines dmpVLT and rstVLT pass the component index ordInd to the component-level dump and restart routines; dmpVLT assumes reordering has been done; rstVLT assumes reordering has not been done.

Array data, generic for all 1D-component types (arrays g1DAr, intAr, and heatAr) declaration: Mesh-cell data for the 1D-component arrays that are common to all the 1D-component types are stored in three derived-type arrays:

g1DAr, intAr, and heatAr.

In module Gen1DArray, derived data-type g1DArrayT is defined, and array g1DAr is declared to be of derived-type g1DArrayT and dimension maxComps:

```
TYPE g1DArrayT
  REAL(sdk), POINTER, DIMENSION(:) :: driv
  ---
  ---
  ---
  REAL(sdk), POINTER, DIMENSION(:) :: dx
  REAL(sdk), POINTER, DIMENSION(:) :: fa
  REAL(sdk), POINTER, DIMENSION(:) :: fric
  REAL(sdk), POINTER, DIMENSION(:) :: grav
  ---
  ---
```

```

---
REAL(sdk), POINTER, DIMENSION(:) :: twan
REAL(sdk), POINTER, DIMENSION(:) :: twen
REAL(sdk), POINTER, DIMENSION(:) :: tcen
END TYPE g1DArrayT
!
TYPE (g1DArrayT), DIMENSION(maxComps) :: g1DAR

```

Array g1DAR contains arrays for variables defined at mesh-cell centers (e.g., pressures) and at cell faces (e.g., velocities). The arrays in g1DAR are characterized further by whether their variables are time-independent (in TRAC-P, these are the hydropt arrays) or time-dependent (in TRAC-P, the dualpt arrays), for which typically there are separate old- and new-time arrays (e.g., array pn for new-time pressures and array p for old-time values).

Module Gen1DArray contains subroutines TimeUpGen1D and BackUpGen1D. These routines use old- and new-time arrays for timestep advancement and for special (water-packer-type) backups, respectively. Examples of their use are given in Appendix H.

Similarly, arrays intAr and heatAr are declared in modules IntArray and HeatArray, respectively:

```

TYPE intArrayT
REAL(sdk), POINTER, DIMENSION(:) :: idr
REAL(sdk), POINTER, DIMENSION(:) :: matid
REAL(sdk), POINTER, DIMENSION(:) :: nff
REAL(sdk), POINTER, DIMENSION(:) :: lccfl
END TYPE intArrayT
!
TYPE (intArrayT), DIMENSION(maxComps) :: intAr

TYPE heatArrayT
REAL(sdk), POINTER, DIMENSION(:, :) :: cpw
---
---
REAL(sdk), POINTER, DIMENSION(:) :: tov
END TYPE heatArrayT
!
TYPE (heatArrayT), DIMENSION(maxComps) :: heatAr

```

Array intAr corresponds to TRAC-P's intpt (note: arrays are real here), and heatAr corresponds to TRAC-P's heatpt.

Storage Allocation: In module Gen1DArrayM, the 1D-component-array derived-type elements are declared to be Fortran 90 pointers, using Fortran 90 colon notation for their thermal hydraulics:

```

REAL(sdk), POINTER, DIMENSION(:) :: dx
---
---
```

```

----
REAL(sdk), POINTER, DIMENSION(:,:) :: twN

```

The actual sizes of the various arrays for each individual component are separately allocated at run time, using standard Fortran 90 allocate statements, as the input is read for a specific component (the pointer attribute is required for allocatable arrays that are derived-type elements; also, some of the arrays in g1DAr are used as targets of other pointer variables, and it is not necessary to give these a target attribute if they are pointers themselves). Subroutine AllocGen1D, which is contained in module Gen1DArray, is called by the 1D-component input routines to allocate all storage for the data arrays within derived-type arrays g1DAr, intAr, and heatAr:

```

SUBROUTINE AllocGen1D(ncells,nfaces,nods,inflg,ihtflg)
----
----
----
  USE GlobalDat  <<<--- for value of cci
----
----
----
  USE Alloc
  USE IntArray  <<<--- for access to array  intAr
  USE HeatArray <<<--- for access to array  heatAr
----
----
----
  CALL TRACAllo(g1DAr(cci)%tcen,1,'tcen',0.0d0) <<<--- allocate 1 word
  CALL TRACAllo(g1DAr(cci)%twen,1,'twen',0.0d0)
  CALL TRACAllo(g1DAr(cci)%twan,1,'twan',0.0d0)
  CALL TRACAllo(g1DAr(cci)%vvt,nfaces,'vvt',0.0d0) <<<--- nfaces words
  CALL TRACAllo(g1DAr(cci)%vlt,nfaces,'vlt',0.0d0)
  CALL TRACAllo(g1DAr(cci)%qppc,ncells,'qppc',0.0d0) <<<--- ncells words
----
----
----
  CALL TRACAllo(intAr(cci)%idr,ncells,'idr',0.0d0)
----
----
----
  CALL TRACAllo(heatAr(cci)%cpw,ndm1,ncells,'cpw',0.0d0)
----
----
----

```

Subroutine AllocGen1D calls TRACAllo to do the actual storage allocations, with one call for each data array. TRACAllo is a generic name (Fortran 90 interface) for subroutines AllocRealOneD, AllocRealTwoD, AllocRealThreeD, and AllocIntOneD; the actual routine used is determined by the compiler according to the number and data types of the actual arguments in the particular call TRACAllo statement. TRACAllo also can initialize an array to a single value that is passed from its caller. TRACAllo is in module Alloc:

```

MODULE Alloc
----
----
----
!   Encapsulation of F90 dynamic allocation and diagnostics for TRAC
!
INTERFACE TRACAllo
  MODULE PROCEDURE AllocRealOneD
  MODULE PROCEDURE AllocRealTwoD
  MODULE PROCEDURE AllocRealThreeD
  MODULE PROCEDURE AllocIntOneD
END INTERFACE
----
----
----

CONTAINS

SUBROUTINE AllocRealOneD(pt,n,name,initialValue)
----
----
----
  ALLOCATE(pt(n),STAT=errorStatus) <<<--- allocate real rank-1 array
----
----
----
  SUBROUTINE AllocRealTwoD(pt,n1,n2,name,initialValue)
----
----
----
  ALLOCATE(pt(n1,n2),STAT=errorStatus) <<<--- real rank-2 array
----
----
----
  SUBROUTINE AllocRealThreeD(pt,n1,n2,n3,name,initialValue)
----
----
----
  ALLOCATE(pt(n1,n2,n3),STAT=errorStatus) <<<--- real rank-3 array
----
----
----
  SUBROUTINE AllocIntOneD(pt,n,name,initialValue)
----
----
----
  ALLOCATE(pt(n),STAT=errorStatus) <<<--- integer rank-1 array
----
----
----
  IF (PRESENT(initialValue)) pt=initialValue <<<--- initialize this array
----
----

```

```
---  
END SUBROUTINE AllocIntOneD  
  
END MODULE Alloc
```

All TRAC 1D-component types (and also the BREAK, FILL, and PLENUM) have a module with a name of the form

```
MODULE "Comp_type".
```

These modules contain component-type-specific routines for I/O, array storage allocation, and driving the generic hydrodynamics routines.

For example, module Pipe contains the following routines:

```
SUBROUTINE dpipe -- add this PIPE to dump file  
SUBROUTINE ipipe -- initialize this PIPE after input  
SUBROUTINE pipe1 -- drive prep hydro stage for this PIPE  
SUBROUTINE pipe1x -- obtain analysis data  
SUBROUTINE pipe2 -- drive OUTER hydro stage  
SUBROUTINE pipe3 -- drive POST hydro stage  
SUBROUTINE repipe -- read restart file; call TRACAllo, AllocGen1D  
SUBROUTINE rpipe -- read input; call TRACAllo, AllocGen1D  
SUBROUTINE wpipe -- write text output
```

The driver input routine (*rpipe*, *repipe*, *rtee*, *rete*, etc.) for each 1D component in the input (or restart) deck uses module Gen1DArray, which defines derived data-type *g1DArrayT*. *AllocGen1D* has a call to subroutine *TRACAllo* for each array that is a member of arrays *g1DAr*, *intAr*, and *heatAr*. As shown in the next section, the input routines also have direct calls to *TRACAllo* for each of their component-specific arrays.

Array data, specific for each 1D-component type (arrays "comp_type"Array): For the 1D-component types, TRAC has these modules to define derived-type arrays for component-type-specific data (currently, there is no need for a module for the pressurizer):

```
PipeArray  
  
PumpArray  
  
TeeArray  
  
ValveArray
```

For example:

```
MODULE PipeArray
```

```

---
---
---
! Pipe component specific arrays
!
TYPE pipeArrayT
  REAL(sdk),    POINTER, DIMENSION(:)    :: powrf
  REAL(sdk),    POINTER, DIMENSION(:)    :: powtb
  REAL(sdk),    POINTER, DIMENSION(:)    :: qp3rf
  REAL(sdk),    POINTER, DIMENSION(:)    :: qp3tb
END TYPE pipeArrayT
!
TYPE (pipeArrayT), DIMENSION(maxComps) :: pipeAr
!
END MODULE PipeArray

```

The separator component has a specific array called `sepdAr`, which is defined in module `Sepd` (which also uses module `TeeArray`).

The component-specific input routines allocate storage for these arrays with direct calls to `TRACAllo`:

```

MODULE Pipe
!
! BEGIN MODULE USE
USE PipeArray
!
CONTAINS
---
---
---
SUBROUTINE repipe(jflag, jun, icip)
!
! BEGIN MODULE USE
---
---
---
USE Alloc
---
---
---
CALL TRACAllo(pipeAr(cci)%powrf, iabs(pipeTab(cci)%npowrf)*2      &
&, 'powrf', 0.d0)
CALL TRACAllo(pipeAr(cci)%powtb, iabs(pipeTab(cci)%npowtb)*2      &
&, 'powtb', 0.d0)
CALL TRACAllo(pipeAr(cci)%qp3rf, iabs(pipeTab(cci)%nqp3rf)*2      &
&, 'qp3rf', 0.d0)
CALL TRACAllo(pipeAr(cci)%qp3tb, iabs(pipeTab(cci)%nqp3tb)*i2    &
&, 'qp3tb', 0.d0)
---
---
---

```

There is analogous logic for the PIPE-specific arrays in subroutine rpipe.

3.2.2.2.2. Pseudo-1D Boundary-Condition-Component Types (BREAK, FILL).

FLT, same as for 1D: The BREAK and FILL component types use array genTab for their FLTs in exactly the same manner as the other components.

VLT, specific for each type: The BREAK and FILL VLT data are treated by modules BreakVlt and FillVlt, respectively; their logic is the same as the other VLT modules. VLT data for individual BREAK and FILL components are stored in derived-type arrays

breakTab and fillTab.

Array data generic for all 1D-component types (subset of 1D): BREAK and FILL both use the general 1D-component array g1DAr and allocate storage for their data arrays in it with calls to AllocGen1D from their input routines in modules Break and Fill, respectively. However, the number of mesh cells is hardwired to be one, the number of wall heat-conduction nodes is hardwired to be zero, the allocation of storage for the data arrays in intAr and heatAr is turned off, and there are hardwired assignment statements for the data arrays in rbreak and rfill:

```
SUBROUTINE rbreak(jflag,jun)
----
----
----
USE Gen1DArray
----
----
----
genTab(cci)%ncellt=1      <<<--- one cell
ncpl=genTab(cci)%ncellt+1
genTab(cci)%nodes=0      <<<--- no nodes
!
! initialize general 1-d pointers
!
CALL AllocGen1D(genTab(cci)%ncellt,ncpl,genTab(cci)%nodes,0,0)
                                                                    ^
                                no intAr and heatAr allocations
----
----
----
gldAr(cci)%dx(1)=dxin      <<<--- array assignment
----
----
----
```

Array data specific for each component type: TRAC has modules for derived-type data arrays specific to the BREAK and FILL component types:

```
MODULE BreakArray
```

```
MODULE FillArray
```

Their logic is similar to that for the 1D-component types:

```

MODULE BreakArray
---
---
---
TYPE breakArrayT
  REAL(sdk),    POINTER, DIMENSION(:)  :: alptb
  REAL(sdk),    POINTER, DIMENSION(:)  :: contb
  REAL(sdk),    POINTER, DIMENSION(:)  :: patb
  REAL(sdk),    POINTER, DIMENSION(:)  :: ptb
  REAL(sdk),    POINTER, DIMENSION(:)  :: rftb
  REAL(sdk),    POINTER, DIMENSION(:)  :: tltb
  REAL(sdk),    POINTER, DIMENSION(:)  :: tvtb
!
END TYPE breakArrayT
!
TYPE (breakArrayT), DIMENSION(maxComps) :: breakAr
!
END MODULE BreakArray

```

Storage is allocated for these arrays by calls from the component-type input routines to `AllBreakArrays` and `AllFillArrays`, respectively, which are in modules `Break` and `Fill`. Subroutines `AllBreakArrays` and `AllFillArrays` are designed to handle other common operations on the `BREAK`- and `FILL`-specific arrays; they also are responsible for the dump and restart for these arrays. These routines contain subroutines `AllBOP` and `AllFOP`, respectively:

```

      CALL AllBreakArrays('allocate',breakTab(cci)%ibty      &
&,breakTab(cci)%isat,breakTab(cci)%nbtb,breakTab(cci)%nbrf,isolut  &
&,idum1,inbtb2,idum2)

MODULE Break
!
! BEGIN MODULE USE
USE BreakArray
USE Global
!
CONTAINS

SUBROUTINE AllBreakArrays
& (mode,ibty,isat,ntb,nrf,isolut,ictrl,intb2,words)
---
---
---
IF (ibty.GE.1.AND.ibty.NE.6) THEN
  intb2=iabs(ntb)*2
  CALL AllBOP(breakAr(cci)%ptb,breakAr(cco)%ptb,intb2,'ptb')
!
  IF (ibty.NE.1) THEN
    CALL AllBOP(breakAr(cci)%tltb,breakAr(cco)%tltb,intb2,'tltb')
  ---
  ---

```

```

---
CONTAINS

SUBROUTINE AllBop(initArray, orderedArray, size, name) <<<-- cci, cco
!
! BEGIN MODULE USE
USE Alloc
USE Restart
---
---
---
IF(mode.EQ.'words') THEN
  words=words+size
ELSEIF(mode.EQ.'allocate') THEN
  CALL TRACAllo(initArray, size, name, 0.d0) <<<-- Call TRACAllo
ELSEIF(mode.EQ.'dump') THEN
  CALL bfoutn(orderedArray, size, ictrl)
ELSEIF(mode.EQ.'restart') THEN
  CALL bfinn(initArray, size, ictrl)
ELSE
  STOP 'AllBreakArrays: Unrecognized mode'
ENDIF
END SUBROUTINE AllBop

END SUBROUTINE AllBreakArrays

```

Subroutine AllFillArrays calls and contains the corresponding service routine AllFOp:

```
CALL AllFOp(fillAr(cci)%alptb, fillAr(cco)%alptb, ntb2, 'alptb')
```

Note that these routines can use either reordered (cco) or nonreordered (cci) indexing into the data arrays, according to the situation and task.

3.2.2.2.3. "0D" Multiple-Connection-Component Type (PLENUM). FLT, same as for 1D: The PLENUM-component type uses array genTab for its FLT in exactly the same manner as the other components. The PLENUM is the only component type to use FLT-variable typeIndex (it is set by variable currentPlenumInd in subroutine rplen), but this logic currently is not used by the code.

VLT, specific for PLENUM: The PLENUM VLT data are treated by module PlenVlt, which has the same logic as the other VLT modules. VLT data for individual PLENUM components are stored in derived-type array

```
plenTab.
```

Array data generic for all 1D-component types (subset of 1D): The PLENUM-component type stores some of its array data in array g1dAr, but it does not use a call to subroutine AllocGen1D. Rather, input routines rplen and replen, in module Plenum, employ calls to PLENUM-specific allocation routine AllocPlenum, which is also in module Plenum; rplen and replen have direct calls to loadn and bfinn, which read

the `g1dAr` data arrays. Many of the PLENUM data arrays are hardwired to a size of one word; others have storage allocated according to the user-input number of junctions for the specific PLENUM component. The PLENUM component is the only component type that associates the pointers for the thermodynamic derivatives and enthalpies in data-type `g1DArrayT` with elements of `g1DArrayT` array `driv`. For the 1D components, `driv` is allocated `nthm*nfaces` words; for the PLENUM, it is allocated `nthm` words.

```

MODULE Plenum
---
---
---
CONTAINS
!
SUBROUTINE AllocPlenum
!
BEGIN MODULE USE
USE Gen1DArray
USE Alloc
USE PlenVlt
---
---
---
CALL TRACAllo(g1DAr(cci)%hiv,1,'hiv',0.0d0)
CALL TRACAllo(g1DAr(cci)%hil,1,'hil',0.0d0)
CALL TRACAllo(g1DAr(cci)%bitn,1,'bitn',0.0d0)
CALL TRACAllo(g1DAr(cci)%tvn,1,'tvn',0.0d0)
CALL TRACAllo(g1DAr(cci)%tln,1,'tln',0.0d0) <<<--- allocate one word
---
---
---
CALL TRACAllo(g1DAr(cci)%driv,nthm,'driv',0.0d0) <<<--- allocate driv
!
g1DAr(cci)%dtsdp=>g1DAr(cci)%driv(1:) <<<--- associate pointer to driv
g1DAr(cci)%deldp=>g1DAr(cci)%driv(2:)
---
---
---
CALL TRACAllo(g1DAr(cci)%favol,plenTab(cci)%npljn,'favol',0.0d0)
g1DAr(cci)%fa=>g1DAr(cci)%favol
CALL TRACAllo(g1DAr(cci)%dx,plenTab(cci)%npljn,'dx',0.0d0)
      ^
      allocate npljn words

```

Array data specific for the PLENUM-component type: TRAC has a module for derived-type data arrays specific to the PLENUM-component type:

PlenArray

Its logic is similar to that for the 1D-component types:

```

MODULE PlenArray
---
```

```

---
---
! Plenum component specific arrays
!
TYPE plenumArrayT
  INTEGER(sik), POINTER, DIMENSION(:) :: i0j <<<--- integer, rank 1
---
---
---
  REAL(sdk), POINTER, DIMENSION(:, :) :: dbnd <<<--- real, rank 2
---
---
---
  REAL(sdk), POINTER, DIMENSION(:) :: favul <<<--- real, rank 1

END TYPE plenumArrayT
!
TYPE (plenumArrayT), DIMENSION(maxComps) :: plenAr
!
END MODULE PlenArray

```

Subroutines `rplen` and `replen` have direct calls to `TRACAllo` to allocate storage for the `plenAr` data arrays. `rplen` and `replen` are contained in module `Plenum`, which uses module `PlenArray`.

Adding new variables: These steps are followed when adding a new variable to the 1D hydrodynamic database for an existing component type. Complete details are given in Appendix H. See also Section 3.2.3.1 for guidelines for modification of the system services (when, for example, a new component type is added). At the end of this subsection, we list a summary of steps for adding a new variable for components when System Services for the component `bd` array logic are affected; full details are given in Section 3.2.3.1.

genTabT (FLT):

1. Modify the definition of data-type `genTabT` and the parameterization of the length of data-type `genTabT`.
2. Add the new variable to the dump/restart file.
3. Read the new variable from the dump/restart file.
4. Modify subroutine `GetGenTable` (as needed).
5. Echo new input variable (as needed).
6. Add to edits.

"comp_type"TabT (VLTs):

1. Modify the definition of data type "comp_type"TabT and the parameterization of the length of data-type "comp_type"TabT.
2. Add the new variable to the dump/restart file.
3. Read the new variable from the dump/restart file.
4. Add or modify subroutine Get"Comp_type"Tab (as needed).
5. Add or modify subroutine Set"Comp_type"Tab (as needed).
6. Echo new input variable (as needed).
7. Add to edits.

Array Data:

For "comp_type"-specific arrays:

1. Add declaration of array to TYPE "comp_type"ArrayT in module "Comp_type"Array.
2. Add allocation of storage for array with call to subroutine TRACAllo in "comp_type" input routines, which are in module "Comp_type".
3. Add array to dump file with call to subroutine bfoutn in "comp_type" dump routine in module "Comp_type".
4. Read array from input file tracin with call to subroutine loadn and echo to output file trcout with call to subroutine warray, in "comp_type" input routine in module "Comp_type", after storage allocation.
5. Read array from restart file trcrst with call to subroutine bfinn and echo (restart) array to trcout with call to subroutine warray, in "comp_type" restart routine in module "Comp_type".
6. Write array to large (major) edits in trcout (as needed).

For general data-arrays (g1DAr):

1. Add declaration of array to TYPE g1DArrayT in module Gen1DArray.
2. Add allocation of storage for array with new call to subroutine TRACAllo, inserted in subroutine AllocGen1D, which is in module Gen1DArray.

3. Add array to dump file with call to subroutine `bfoutn` in subroutine `dcomp`.
4. Read array from input file `tracin` with call to subroutine `loadn`, and echo to output file `trcout` with call to subroutine `warray`, in subroutine `rcomp`, after storage allocation.
5. Read array from restart file `trcrst` with call to subroutine `bfinn` and echo (restart) array to `trcout` with call to subroutine `warray`, in subroutine `recomp`.
6. Write array to large (major) edits in `trcout` (as needed) with call to subroutine `wcomp`.
7. If the array is in DUALPT, add assignment statements for it to subroutine `TimeUpGen1D` (module `Gen1DArray`) (in two places).
8. If appropriate, add an assignment statement for the array to subroutine `BackUpGen1D` (module `Gen1DArray`).
9. On an as-needed basis, add a new index variable for the array to the module `Gen1DArray` data interface and add a corresponding array reference to the case construct in subroutine `Get1DArrayPointer` (module `Gen1DArray`).

bd Array and System Services:

1. Add variable to the `bd`-array, derived-type structure in module `Boundary`. (Note: the `bd` derived type is not currently implemented.)
2. Increment variable `nbd` by one.
3. Add call to `SetBDVar` for the new variable in `SetBDJunCell`.
4. Add case statement to `flipSign` logic in `SetBDVar`, if appropriate.
5. Add case statement to the associated logic in `AssignGen1DPtr` (in post-3.0 versions, this routine is called `AssignPtr`).

If the new variable is isolated for use by only one or two components, then pointers should be set up specifically for these components. Remaining components should have this variable pointing to the `null` variable in module `SysService` (in a post-3.0 version, variable `null` will be moved to module `Global`).

3.2.2.2.4. 3D Hydrodynamic-Component Type (VESSEL). FLT, same as for 1D, array `genTab`: The 3D VESSEL component uses the same FLT as the other component types. FLT data for individual VESSEL components are stored in derived-type array

genTab.

VLT, specific 3D-component type: The VESSEL VLT data are treated by module VessVlt; its logic is the same as the other VLT modules. VLT data for individual VESSEL components are stored in derived-type array

vessTab.

Array data for the VESSEL-component type: Array data for a specific 3D VESSEL component fall into two categories:

1. Special array variables provide "general" information for a VESSEL, i.e., data that are not defined at each cell in the 3D mesh. Examples of such data are arrays that carry the physical lengths of the VESSEL's radial segments ("rings"), azimuthal segments ("thetas" or "sectors"), and axial segments ("levels"); these are dimensioned according to the user-input parameters *nrsx*, *ntsx*, and *nasx*, respectively (for a VESSEL modeled in Cartesian coordinates, these variables correspond to the *x*, *y*, and *z* coordinates, respectively). Other examples are arrays with information for source connections to 1D hydrodynamic components, dimensioned by input-parameter *ncsr* (or a multiple thereof), and arrays for vent-valve information, dimensioned by input parameter *nvent*.

All of these arrays are of rank 1; storage for them is allocated at run time. A special case of the special array variables is that which carries 3D VESSEL information that is only 2D in nature; typically, these data specify a different single value for each axial column of mesh cells in the VESSEL and are dimensioned *nrsx* x *ntsx* (the number of mesh cells per level). As shown below, elements of these rank-1 arrays are accessed by composite indices into the 2D plane.

2. 3D mesh-cell variables carry information that is defined for each cell in the VESSEL fluid mesh; they are all of rank 3, and storage for them is allocated at run time. Some of these arrays hold data that are defined at each cell center (e.g., pressure and liquid temperature), and some hold data defined at cell faces (e.g., vapor velocity in the radial direction and liquid velocity in the axial direction). For the face arrays, information for only three faces per cell is needed because the other faces are defined at neighboring cells in the mesh.

Special array variables—declaration: The special array variables for all VESSEL components in a model are stored in derived-type array

vsAr,

which is of type `vessArrayT` and dimension `maxComps`. Module `VessArray` defines the elements of data type `vessArrayT` and declares array `vsAr` (note that `vessArrayT` contains both integer and real arrays):

```

MODULE VessArray
!
! BEGIN MODULE USE
USE IntrType
USE GlobalDim <<<--- parameter maxComps
!
IMPLICIT NONE
----
----
----
! VESSEL component specific arrays
!
TYPE vessArrayT
  REAL(sdk), POINTER, DIMENSION(:) :: z
  REAL(sdk), POINTER, DIMENSION(:) :: dz
  REAL(sdk), POINTER, DIMENSION(:) :: rad
  REAL(sdk), POINTER, DIMENSION(:) :: dr
  REAL(sdk), POINTER, DIMENSION(:) :: th
  ----
  ----
  ----
  REAL(sdk), POINTER, DIMENSION(:) :: zsgrd
  INTEGER(sik), POINTER, DIMENSION(:) :: isrl
  INTEGER(sik), POINTER, DIMENSION(:) :: isrc
  INTEGER(sik), POINTER, DIMENSION(:) :: isrf
  ----
  ----
  ----
  INTEGER(sik), POINTER, DIMENSION(:) :: nsrl
  REAL(sdk), POINTER, DIMENSION(:) :: svc
  REAL(sdk), POINTER, DIMENSION(:) :: sac
  ----
  ----
  ----
  REAL(sdk), POINTER, DIMENSION(:) :: alpcn
  REAL(sdk), POINTER, DIMENSION(:) :: alpntn
  REAL(sdk), POINTER, DIMENSION(:) :: zchfn
  REAL(sdk), POINTER, DIMENSION(:) :: ztbn
!
END TYPE vessArrayT
!
TYPE (vessArrayT), TARGET, DIMENSION(maxComps) :: vsAr <<<--- declare vsAr

```

Because the elements of data-type `vessArrayT` are allocatable arrays, they have the pointer attribute.

Storage allocation: Subroutines `rvssl` and `revssl`, which are contained in module `VessTask`, call subroutine `AllocVess` (module `VessArray`). `AllocVess` calls `TRACAllo` for each array in `vsAr`:

```

MODULE VessArray
----
CONTAINS

SUBROUTINE AllocVess
!
! BEGIN MODULE USE
USE GlobalDat
USE VessCon
USE VessVlt
USE Alloc
----
----
--- Store dimension data in local variables:
nclx=vessTab(cci)%nclx
nrsx=vessTab(cci)%nrsx
nytv=vessTab(cci)%nytv
nasx=vessTab(cci)%nasx
ntsx=vessTab(cci)%ntsx
ncsr=vessTab(cci)%ncsr
nvent=vessTab(cci)%nvent
----
----
CALL TRACallo(vsAr(cci)%ztbn,nclx,'ztbn',0.0d0) <<<--- # cells per level
CALL TRACallo(vsAr(cci)%zchfn,nclx,'zchfn',0.0d0)
----
----
CALL TRACallo(vsAr(cci)%jsn,ncsr,'jsn',0) <<<--- # source connections
CALL TRACallo(vsAr(cci)%juns,ncsr,'juns',0)
----
----
CALL TRACallo(vsAr(cci)%avent,nvent,'avent',0.0d0) <<<--- # vent valves
CALL TRACallo(vsAr(cci)%dth,ntsx,'dth',0.0d0) <<<--- # thetas or y-cells
CALL TRACallo(vsAr(cci)%th,ntsx,'th',0.0d0)
CALL TRACallo(vsAr(cci)%dr,nrsx,'dr',0.0d0) <<<--- # rings or x-cells
CALL TRACallo(vsAr(cci)%rad,nrsx,'rad',0.0d0)
CALL TRACallo(vsAr(cci)%dz,nasx,'dz',0.0d0) <<<--- # levels or z-cells
CALL TRACallo(vsAr(cci)%z,nasx,'z',0.0d0)
!
END SUBROUTINE AllocVess

```

Accessing special-array 2D elements: The special-array variables that carry 2D information for the nclx columns in a VESSEL through the (i,j) plane are accessed by a composite index that is calculated as the VESSEL mesh is looped over:

```

MODULE VessTask
----
----

```

```

---
CONTAINS

SUBROUTINE cif3 <<<--- VESSEL interfacial shear
---
---
---
USE VessVlt <<<--- obtain (i,j,k) ranges for this VESSEL
USE VessArray
---
---
---
! loop over all levels for each radial and azimuthal mesh
!
  ir=0
  DO i=vessTab(cco)%ic0,vessTab(cco)%icx,nv <<<--- radial loop
    ir=ir+1 <<<--- ring counter
    jct=(ir-1)*(vessTab(cco)%jcx-vessTab(cco)%jc0+1)
    it=0
    DO j=vessTab(cco)%jc0,vessTab(cco)%jcx <<<--- azimuthal loop
      it=it+1 <<<--- theta counter
      jct=jct+1 <<<--- index into (i,j) plane
      agalp=vsAr(cco)%alpan(jct)
      chfalp=vsAr(cco)%alpcn(jct)
      rwalp=vsAr(cco)%alprn(jct)
      smalp=vsAr(cco)%alpsn(jct)
      tbalp=vsAr(cco)%alptn(jct)
      agsz=vsAr(cco)%zagsn(jct)
      chfz=vsAr(cco)%zchfn(jct)
      dfsz=vsAr(cco)%zdfsn(jct)
      rwsz=vsAr(cco)%zrwsn(jct)
      smsz=vsAr(cco)%zsmsn(jct)
      tbz=vsAr(cco)%ztbn(jct)
      nrefld=int(vsAr(cco)%refld(jct))
      xfunh=vsAr(cco)%funh(jct)
      DO k=vessTab(cco)%kc0,vessTab(cco)%kcx <<<--- axial loop
        iz=k-nzbcm
      !
    ! set VESSEL location logical variables
    ---
    ---
    ---

```

3D mesh-cell variables: A TRAC 3D VESSEL component may be modeled in either cylindrical (r, q, z) or Cartesian (x, y, z) coordinates. All 3D mesh arrays are indexed by indices (i, j, k) that correspond to either the (r, q, z) or (x, y, z) coordinates, depending on the particular VESSEL's geometry. When we refer to rings and thetas, we are indicating in a more general sense the i and j indices, respectively.

Declaration: The mesh arrays for the 3D VESSEL component are stored in a derived-type array of dimension `maxComps`, in much the same fashion as the generic arrays for the 1D components. Module `VessArray3` defines derived data-type `vessArray3T`, the

elements of which are of dimension (:, :, :) and have the pointer attribute. Array vsAr3 is declared to be of TYPE vessArray3T and dimension (maxComps):

```

MODULE VessArray3
----
----
----
TYPE vessArray3T                                <<<--- define TYPE  vessArray3T
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: h1a
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: hva
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: q3dr1
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: q3drv
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: wat
----
----
----
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: xcv
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: xv1
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: xv2
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: xv3
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: xv4
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: xv5
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: xv6
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: xvs
END TYPE vessArray3T
!
TYPE (vessArray3T), DIMENSION(maxComps) :: vsAr3 <<<--- declare vsAr3

```

Indices for all specific VESSEL components in an input model are stored with the other component indices in array compIndices (module Global). Typically, array data for specific Vessels in array vsAr3 are accessed using the same cci and cco index variables used for the other components.

Storage allocation: Table 3-4 summarizes the variables used in allocating storage for the VESSEL mesh arrays (also included in Table 3-4 are two parameter variables that limit the maximum size of any VESSEL component in an input model; these variables are used only for special purposes).

Each specific VESSEL component has its mesh arrays dynamically allocated at runtime; this storage allocation is based on the user-input number of mesh cells for each of its three dimensions [using input variables nr_{rx} (radial or x-coordinate)], nt_{rx} (azimuthal or y-coordinate), and na_{rx} (axial or z-coordinate), which are stored in the VESSEL's VLT). Each array is dimensioned according to VESSEL VLT variables ni, nj, and nk, which specify the lengths of the i, j, and k subscripts, respectively. Variables ni, nj, and nk include storage for the VESSEL's physical mesh, as input by the user, and also storage for boundary cells that TRAC uses internally along each coordinate. The number of boundary cells is specified by six parameter variables in module VessCon (see Table 3-4 and Fig. 3-1) for the low- and high-numbered ends of each of the three coordinates. The use of the boundary cells is described in the following section.

TABLE 3-4
VESSEL-Array Dimension Variables

Name	Storage Location	Purpose
nrsx	vessTab	The number of physical mesh cells in the radial or x coordinate (user input).
ntsx	vessTab	The number of physical mesh cells in the azimuthal or y coordinate (user input).
nasx	vessTab	The number of physical mesh cells in the axial or z coordinate (user input).
nxbcm	Module VessCon	The number of boundary (phantom) mesh cells next to the radial ring or x-direction cell 1.
nybcm	Module VessCon	The number of boundary (phantom) mesh cells next to the azimuthal sector or y-direction cell 1.
nzbcm	Module VessCon	The number of boundary (phantom) mesh cells next to the axial or z-direction cell 1.
nxbcp	Module VessCon	The number of boundary (phantom) mesh cells next to the radial ring or x-direction cell nrsx.
nybcp	Module VessCon	The number of boundary (phantom) mesh cells next to the azimuthal sector or y-direction cell ntsx.
nzbcp	Module VessCon	The number of boundary (phantom) mesh cells next to the axial or z-direction cell nasx.
ni	vessTab	The total number of computer words allocated for the first subscript, i (radial or x coordinate), of a VESSEL mesh array: $ni = nrsx + nxbcm + nxbcp$
nj	vessTab	The total number of computer words allocated for the second subscript, j (azimuthal or y coordinate), of a VESSEL mesh array: $nj = ntsx + nybcm + nybcp$
nk	vessTab	The total number of computer words allocated for the third subscript, k (axial or z coordinate), of a VESSEL mesh array: $nk = nasx + nzbcm + nzbcp$
nxrmx	Module VessCon	The maximum number of radial rings or x-direction cells in the 2D or 3D mesh for any VESSEL component in the input model. Used only to statically dimensioned global array wp and the HTSTR/VESSEL interface arrays in module RodHtcref1.
nytmx	Module VessCon	The maximum number of azimuthal sectors or y-direction cells in the 2D or 3D mesh for any VESSEL component in the input model. Used only to statically dimensioned global array wp and the HTSTR/VESSEL interface arrays in module RodHtcref1.

```

MODULE VessCon
!
!   BEGIN MODULE USE
!   USE IntrType
!
!   From parset1.h:
!
Parameters nxrmx and nytmx are used only to statically dimension global array
wp and the HTSTR/VESSEL interface arrays in module RodHtcref1 (via
parameter nxryt):
!
!   INTEGER(sik) nxrmx,nytmx
!   PARAMETER (nxrmx=21,nytmx=12)
!   INTEGER(sik) nrfmx,nzfmx
!   PARAMETER (nrfmx=20,nzfmx=250)
!   INTEGER(sik) nms,ndms
!   PARAMETER (nms=10,ndms=7+nms)
!
!   INTEGER(sik) nxbcm,nybcm,nzbcm
!   PARAMETER (nxbcm=2,nybcm=2,nzbcm=2) <<<--- low-boundary cells
!   INTEGER(sik) nxbcp,nybcp,nzbcp
!   PARAMETER (nxbcp=1,nybcp=1,nzbcp=1) <<<--- high-boundary cells
!   INTEGER(sik) nxryt
!   PARAMETER (nxryt=nxrmx*nytmx) <<<--- nxryt, used by module
!                                     RodHtcref1
!
!   INTEGER(sik) nrfmx1,nrzfmx
!   PARAMETER (nrfmx1=nrfmx+1,nrzfmx=nrfmx*nzfmx)
!
!   From parset2.h: <<<---the following is for loop-index logic:
!
!   PARAMETER (jc0p=1+nybcm)
!   PARAMETER (kc0p=1+nzbcm)
!   PARAMETER (jc0mp=jc0p-1,kc0mp=kc0p-1)
!   PARAMETER (jc0mmp=jc0p-nybcm,kc0mmp=kc0p-nzbcm)
!
!   From parset0.h:
!
!   PARAMETER (iseq=1,imfreq=1,idrpeq=0,nfr3eq=2)
!
!   END MODULE VessCon

```

Fig. 3-1. Module VessCon

Subroutine AllocVess3 (module VessArray3) allocates all storage for the VESSEL mesh arrays, using calls to TRACAllo for each array. Subroutines rvssl and revssl (module VessTask) each have a call to AllocVess3. In rvssl, the three input variables that dimension the physical mesh are first stored in local variables nxx, nyt, and nzz. Then, ni, nj, and nk are calculated using the six boundary-cell parameter variables, and TRACAllo is called:

```

MODULE VessTask
---
---
---
CONTAINS
---
---
---
SUBROUTINE rvssl(icompl,jflag,jun)
---
--- read nasx, nrsx, and ntsx; store in local variables:
---
!   read input parameters to be stored in vlt
!
CALL readi('iiii',vessTab(cci)%nasx,vessTab(cci)%nrsx
&,vessTab(cci)%ntsx,vessTab(cci)%ncsr,vessTab(cci)%ivssbf,'nasx'   &
&,'nrsx','ntsx','ncsr','ivssbf')
jflagd=0
nxx=vessTab(cci)%nrsx
nyt=vessTab(cci)%ntsx
nzz=vessTab(cci)%nasx
---
--- calculate ni, nj, and nk; call AllocVess3:
---
vessTab(cci)%ni=nxx+nxbcm+nxbcpl
vessTab(cci)%nj=nyt+nybcm+nybcpl
vessTab(cci)%nk=nzz+nzbcm+nzbcpl
!
CALL AllocVess(); <<<--- for special arrays
CALL AllocVess3(vessTab(cci)%ni,vessTab(cci)%nj,vessTab(cci)%nk   &
&,cci)
---
---
---
MODULE VessArray3
---
--- allocate all VESSEL mesh arrays with arguments   ni, nj, nk:
---
CONTAINS
SUBROUTINE AllocVess3 (ni,nj,nk,ccix)
!
! BEGIN MODULE USE
! USE Alloc
!

```

```

IMPLICIT NONE
INTEGER(sik) ni,nj,nk,ccix

!

CALL TRACAllo(vsAr3(ccix)%hla,ni,nj,nk,'hla',0.0d0)
CALL TRACAllo(vsAr3(ccix)%hva,ni,nj,nk,'hva',0.0d0)
CALL TRACAllo(vsAr3(ccix)%q3drl,ni,nj,nk,'q3drl',0.0d0)
CALL TRACAllo(vsAr3(ccix)%q3drv,ni,nj,nk,'q3drv',0.0d0)
---
---
---
CALL TRACAllo(vsAr3(ccix)%xv6,ni,nj,nk,'xv6',0.0d0)
CALL TRACAllo(vsAr3(ccix)%xvs,ni,nj,nk,'xvs',0.0d0)

!

END SUBROUTINE AllocVess3

```

VESSEL boundary (phantom) cells: The VESSEL mesh in TRAC is constructed with two planes of boundary cells outside the mesh in each of the three lower-numbered directions, with one plane of boundary cells in each of the higher-numbered directions. The use of boundary cells allows all references from cells within the physical mesh to neighboring cells outside the physical mesh to be valid. The extra plane in the lower-numbered directions is necessary to accommodate face-centered data. The number of boundary cells in each direction is determined by parameter constants that are set in module VessCon:

```

INTEGER(sik) nxbcm,nybcm,nzbcm
PARAMETER (nxbcm=2,nybcm=2,nzbcm=2) <<<--- low-boundary cells
INTEGER(sik) nxbcp,nybcp,nzbcp
PARAMETER (nxbcp=1,nybcp=1,nzbcp=1) <<<--- high-boundary cells

```

When using a 3D VESSEL component to model a typical cylindrical-geometry reactor VESSEL with outer-boundary walls, the data in the bottom and top axial-boundary cells and in the outer radial-boundary cells do not affect the calculation. However, the inner radial-boundary cells can be used to incorporate the effect of radial-momentum convection across the center of the VESSEL. Such a model was implemented using a different mechanism in TRAC-PF1/MOD1. This model, which is partially implemented in subroutine vrbd (module VessCrunch), is not currently activated in TRAC. The azimuthal-boundary cells are used to avoid the special logic necessary to indicate that the first physical azimuthal sector is adjacent to the last physical azimuthal sector. This is accomplished by subroutine setbdt (module VessCrunch), which copies the data from the cells in the first and last physical sectors to their appropriate phantom cells.

The boundary-cell implementation makes it simple to include generalized boundary conditions at the bottom-axial, top-axial, and outer-radial boundaries of a cylindrical VESSEL and at all external boundaries of a 3D Cartesian-geometry VESSEL. TRAC contains the appropriate coding in all VESSEL hydrodynamic routines to allow for fixed-pressure (such as a BREAK component) or fixed-velocity (such as a FILL component) boundary conditions independently at any of these boundaries. However, this coding for the radial (or x) and azimuthal (or y) boundaries has not yet been tested. In the currently released version of TRAC, there is no input-data mechanism to activate this coding. Input option ivssbf activates only the generalized boundary conditions at the

lower and upper axial faces. There currently is no coding to allow for the generalized boundary conditions to be time-dependent. However, implementing such a capability should not require major changes to TRAC.

In addition to providing for the new generalized boundary conditions, using phantom cells improves implementation of the standard hydrodynamic algorithms. Without the use of phantom cells, special program logic is required to calculate expressions, including gradients and fluxes for cells at the edge of the physical mesh. Such logic would increase the probability of coding errors and inhibit vectorization on hardware such as a Cray computer.

For typical coarse-mesh 3D VESSEL components, most of the cells are found at the edges of the mesh. For example, a VESSEL component with four radial rings and four azimuthal sectors on each level actually has only 4 of the 16 cells on a level that has neither a radial nor an azimuthal boundary. Because even straightforward vectorization generally reduces computation time by more than a factor of 5, it is clearly desirable to design implementations that are vectorizable for all cells.

As stated previously, if phantom cells are not used, special logic would be required to carry out calculations for cells at the edge of the physical mesh. On the other hand, when phantom cells are used, additional procedures are required to define the values associated with the phantom cells. The amount of code that must be maintained is similar in either case; however, the phantom-cell methodology is more easily modularized.

The major disadvantage in using phantom cells is the potential for significantly increased computer-memory requirements for coarse-mesh VESSEL components. For our previous example, a VESSEL with 4 radial rings, 4 azimuthal sectors, and 10 axial levels has only $4 \times 4 \times 10$, or 160, physical mesh cells. However, it will have $(4 + 3) \times (4 + 3) \times (10 + 3)$, or 637, computational mesh cells when including the boundary cells. Naturally, the percentage of boundary cells is smaller for more finely noded problems. The current VESSEL mesh array data contain about 300 different variables; thus, this example would require about 200,000 words of computer memory for the 48,000 words of physical mesh-cell array data. However, for most modern computer hardware, this is not a large amount of memory, and the cost-benefit ratio of this memory increase is extremely favorable when considering the more efficient coding.

Because both of the lowest-numbered planes of phantom cells in each direction are used only in conjunction with the generalized boundary-condition option associated with a fixed-pressure boundary condition, it should be possible to reduce the memory requirements by changing from 2 to 1 the parameter constants defining the number of lower-numbered phantom cells for the radial or x and azimuthal or y directions. However, this reduction has not been tested.

DO-loop limits: The lower and upper limits of the many Fortran DO loops over the VESSEL mesh arrays in TRAC are stored in variables in each VESSEL component's VLT (array `vessTab`). As shown below, these limits are calculated in module `VessCon` (see

Fig. 3-1) and in subroutine *rvssl* (module *VessTask*) according to the individual input specifications for each VESSEL component and the number of boundary cells specified for each coordinate in module *VessCon* (Fig. 3-1). All array-dimension loop-limit variable names have the same naming convention, with the first letter, i.e., *i*, *j*, and *k*, indicating the first (radial- or *x*-direction), second (azimuthal- or *y*-direction), and third (axial- or *z*-direction) array subscripts, respectively. The letter *c* in a name denotes a limit suitable for looping over cells, and the letter *f* denotes a limit suitable for looping over cell faces. The convention for cell-face variables in the TRAC-M VESSEL is the same as in TRAC-P: the cell-face data at the outer (*r* or *x*), forward (θ or *y*), or upper (*z*) face of a cell have the same index as the data at the cell center. Note that, as indicated above in the section on boundary cells, cell faces at the VESSEL boundaries are included in the cell-face loops only when their velocities need to be calculated as a result of using the generalized boundary-condition *ivssbf* option for a pressure boundary condition.

The numeral 0 in a name denotes a lower limit, and the letter *x* denotes an upper limit. The suffix *m* denotes a lower limit that includes the boundary cell adjacent to the first physical cell, and the suffix *mm* denotes a lower limit that includes all of the low-numbered boundary cells. The suffix *p* denotes an upper limit that includes the boundary cell adjacent to the last physical cell, and the suffix *a11* denotes an upper limit that includes all the high-numbered boundary cells. The variable names for the radial- or *x*-direction are

<i>ic0mm</i>	Lower limit for loop over all radial rings or <i>x</i> -direction cells in the computational mesh.
<i>ic0m</i>	Lower limit for loop over radial rings or <i>x</i> -direction cells in the physical mesh and the adjacent low-numbered phantom or boundary radial ring or <i>x</i> -direction cell.
<i>ic0</i>	Lower limit for loop over all radial rings or <i>x</i> -direction cells in the physical mesh.
<i>if0</i>	Lower limit for loop over all radial-ring faces or <i>x</i> -direction cell faces at which velocities are calculated.
<i>icx</i>	Upper limit for loop over all radial rings or <i>x</i> -direction cells in the physical mesh.
<i>ifx</i>	Upper limit for loop over all radial-ring faces or <i>x</i> -direction cell faces at which velocities are calculated.
<i>icxp</i>	Upper limit for loop over radial rings or <i>x</i> -direction cells in the physical mesh and the adjacent high-numbered phantom or boundary radial ring or <i>x</i> -direction cell.
<i>ia11</i>	Upper limit for loop over all radial rings or <i>x</i> -direction cells in the computational mesh.

The variable names for the azimuthal or y-direction loop limits can be obtained by replacing the leading i with a j and those for the axial or z-direction loops by replacing the leading i with a k. The code developer should not have to change any of the coding of the loop limits in either module *VessCon* or in subroutine *rvssl*. The coding of the loop limits is described here for completeness. Certain loop limits are hard coded with parameter statements, which are defined as follows in module *VessCon* (see Fig. 3-1):

```

jcop = nybcm + 1
jc0mp = jc0p - 1
jcommp = jc0p - nybcm
kc0p = nzbcm + 1
kc0mp = kc0p - 1
kc0mmp = kc0p - nzbcm

```

The "p" in these names stands for parameter because they are parameter constants. These constants are copied to the corresponding *vessTab* variables *jc0*, *jc0m*, *jc0mm*, *kc0*, *kc0m*, and *kc0mm*, using the standard naming convention in subroutine *rvssl*:

```

MODULE VessTask
---
---
CONTAINS
---
---
---
SUBROUTINE rvssl(icom, jflag, jun)
---
---
---
vessTab(cci)%jc0=jc0p
vessTab(cci)%jc0m=jc0mp
vessTab(cci)%jc0mm=jc0mmp
vessTab(cci)%kc0=kc0p
vessTab(cci)%kc0m=kc0mp
vessTab(cci)%kc0mm=kc0mmp
---
---
---

```

Additional radial- or x-direction, azimuthal- or y-direction, and axial or z-direction lower loop limits, as well as all of the upper loop limits, are defined dynamically for each 3D VESSEL component in subroutine *rvssl*, where

nrx is the input number of physical radial rings or x-direction cells,

nyt is the input number of physical azimuthal sectors or y-direction cells,

nzz is the input number of physical axial levels or z-direction cells,

igeom is 0 for cylindrical geometry and 1 for Cartesian geometry,

igbcxr is nonzero for generalized radial- or x-direction boundary conditions,

igbcyt is nonzero for generalized azimuthal- or y-direction boundary conditions,

igbcz is nonzero for generalized axial-direction boundary conditions,

nrxv is the upper limit corresponding to loop-limit ifx,

nytv is the upper limit corresponding to loop-limit jfx, and

nzzv is the upper limit corresponding to loop-limit kfx.

Variables nrx, nyt, and nzz are the same local variables that are used by rvssl to dimension the VESSEL fluid-mesh arrays. In the current version of TRAC, IGBCXR and IGBCYT are always 0 and IGBCZ is only nonzero when the VESSEL outer-boundary-condition input flag, IVSSBF, is nonzero.

```
---
---
---
nrx=vessTab(cci)%nrxx
nyt=vessTab(cci)%ntxy
nzz=vessTab(cci)%nazx
vessTab(cci)%igbcxr=0
vessTab(cci)%igbcyt=0
vessTab(cci)%igbcz=0
IF (vessTab(cci)%ivssbf.NE.0) THEN
    vessTab(cci)%igbcz=1
---
---
---
vessTab(cci)%nrxv=vessTab(cci)%nrxx-1
IF (vessTab(cci)%igeom.EQ.0.AND.vessTab(cci)%igbcxr.NE
&.0) vessTab(cci)%nrxv=vessTab(cci)%nrxx
IF (vessTab(cci)%igeom.NE.0.AND.vessTab(cci)%igbcxr.NE
&.0) vessTab(cci)%nrxv=vessTab(cci)%nrxx+1
vessTab(cci)%nytv=vessTab(cci)%ntxy
IF (vessTab(cci)%igeom.EQ.0.AND.vessTab(cci)%ntxy.EQ
&.1) vessTab(cci)%nytv=0
IF (vessTab(cci)%igeom.NE.0.AND.vessTab(cci)%igbcyt.EQ
&.0) vessTab(cci)%nytv=vessTab(cci)%ntxy-1
IF (vessTab(cci)%igeom.NE.0.AND.vessTab(cci)%igbcyt.NE
&.0) vessTab(cci)%nytv=vessTab(cci)%ntxy+1
vessTab(cci)%nzzv=vessTab(cci)%nazx-1
IF (vessTab(cci)%igbcz.NE.0) vessTab(cci)%nzzv=vessTab(cci)%nazx+1
---
---
---
! Set up the start value for the first (x or r) variable index
```

```

!
vessTab(cci)%ic0mm=1
vessTab(cci)%ic0=vessTab(cci)%ic0mm+nxbcm
vessTab(cci)%ic0m=vessTab(cci)%ic0-1
---
---
---
vessTab(cci)%jf0=vessTab(cci)%jc0
IF (vessTab(cci)%igeom.EQ.1.AND.vessTab(cci)%igbcyt.EQ      &
&.1) vessTab(cci)%jf0=vessTab(cci)%jc0m
vessTab(cci)%jcx=vessTab(cci)%jc0+nyt-1
vessTab(cci)%jcxp=vessTab(cci)%jcx+1
vessTab(cci)%jall=vessTab(cci)%jcx+nybcp
vessTab(cci)%nijt=ni*vessTab(cci)%jall
vessTab(cci)%kf0=vessTab(cci)%kc0
IF (vessTab(cci)%igbcz.EQ.1) vessTab(cci)%kf0=vessTab(cci)%kc0m
vessTab(cci)%kcx=vessTab(cci)%kc0+nzz-1
vessTab(cci)%kcxp=vessTab(cci)%kcx+1
vessTab(cci)%kall=vessTab(cci)%kcx+nzbcpc
vessTab(cci)%if0=vessTab(cci)%ic0
IF (vessTab(cci)%igeom.EQ.1.AND.vessTab(cci)%igbcxr.EQ      &
&.1) vessTab(cci)%if0=vessTab(cci)%ic0m
vessTab(cci)%icx=vessTab(cci)%ic0+(nrx-1)
vessTab(cci)%icxp=vessTab(cci)%icx+1
vessTab(cci)%iall=vessTab(cci)%icx+nxbcp
vessTab(cci)%jfx=vessTab(cci)%jf0+vessTab(cci)%nytv-1
vessTab(cci)%kfx=vessTab(cci)%kf0+vessTab(cci)%nzzv-1
vessTab(cci)%ifx=vessTab(cci)%if0+(vessTab(cci)%nxrv-1)
---
---
---

```

VESSEL interface to 1D/3D service routines: Subroutines *fprop*, *thermo*, *htif*, and *evaldf1d* perform services for the hydrodynamic solution that are common to the 1D and the 3D components. They evaluate fluid properties (*fprop* and *thermo*), interfacial heat transfer (*htif*), and new-time/old-time fluid changes for the timestep-size logic (*evaldf1d*). The VESSEL component passes data to these routines and receives data back from them (with the exception of *evaldf1d*) via interface-module *VessTo1D*, which contains subroutines *Evaldf3D*, *Fprop3D*, *Htif3D*, and *Therm3D*. These interface routines copy required VESSEL 3D mesh data into local rank-1 arrays for a given VESSEL level, according to the standard TRAC numbering convention of

ring 1, thetas 1--ntsx; ring 2, thetas 1--ntsx; ring nrnx, thetas 1--ntsx.

These rank-1 arrays are then passed to the 1D-3D service routine. The interface routines also declare local rank-1 arrays for any data to be returned from the service routines via their argument lists; after the service routine call, these rank-1 arrays are copied into the 3D fluid-mesh arrays.

Note that the service routines expect the thermodynamic-derivative arrays to be in "cell-wise" storage, where each of the various derivatives is stored contiguously for each mesh cell (as opposed to "mesh-wise" storage, where the elements of each derivative

array are stored contiguously). The service routines access a given derivative for a given mesh cell by calculating an offset pointer according to the total number of derivative arrays; currently, there are 18 thermodynamic-derivative arrays, a value which is parameterized in variable `nthm` in module `GlobalDim`:

```
PARAMETER (nthm=18).
```

The interface routines in module `VessTo1D` stack the 18 derivative arrays in a cell-wise sense before the service routines are called. The interface routines are called once for each axial level; the (i, j) start and stop indices, the level index, and the number of cells per level are passed:

```
MODULE VessTask
---
---
CONTAINS
---
---
SUBROUTINE vss12(isrl, isrc, isrf, jsn, lr, jsnget, jsnput)
---
---
DO k=vessTab(cco)%kc0, vessTab(cco)%kcx <<<--- loop over axial levels;
                                         call interface routine:
    iz=k-nzbcm
    jstart=1
!
    CALL Htif3D(vessTab(cco)%ic0, vessTab(cco)%icx           &
&             , vessTab(cco)%jc0, vessTab(cco)%jcx, k, vessTab(cco)%nc1x)
!
ENDDO
```

In this example, the `VESSEL` index `cco` is not passed (it is obtained through use association); for the other interface routines, `cco` also is passed.

```
MODULE VessTo1D
---
---
CONTAINS
---
---
SUBROUTINE Htif3D(istart, iend, jstart, jend, k, ncellx)
---
---
REAL(sdk) dum1(1) <<<--- dummy array, not used by htif for 3D
REAL(sdk) alpv(ncellx) <<<--- number of cells per level
REAL(sdk) alpov(ncellx)
```

```

REAL(sdk)  rovv(ncellx)
REAL(sdk)  rolv(ncellx)
---
---
---
REAL(sdk)  tsnv(ncellx)
REAL(sdk)  spifzv(ncellx)
REAL(sdk)  drivv(ncellx*nthm) <<<--- for derivative arrays
!
dum1(1)=0.0d0
iv=1      <<<--- for ordering mesh cells
ivdr=1    <<<--- for derivative-array stacking
DO i=istart,iend  <<<--- radial loop
  DO j=jstart,jend  <<<--- theta loop
!
    alpv(iv)=vsAr3(cco)%alp(i,j,k)
    alpov(iv)=vsAr3(cco)%alpo(i,j,k)
    rovv(iv)=vsAr3(cco)%rov(i,j,k)
    rolv(iv)=vsAr3(cco)%rol(i,j,k)
    visvv(iv)=vsAr3(cco)%visv(i,j,k)
    vislv(iv)=vsAr3(cco)%visl(i,j,k)
    pv(iv)=vsAr3(cco)%p(i,j,k)
    ---
    ---
    ---
    tsnv(iv)=vsAr3(cco)%tsn(i,j,k)
    spifzv(iv)=vsAr3(cco)%spifz(i,j,k)
!
    drivv(ivdr)=vsAr3(cco)%dtsdp(i,j,k) <<<--- cell-wise stacking
    ivdr=ivdr+1
    drivv(ivdr)=vsAr3(cco)%deldp(i,j,k)
    ivdr=ivdr+1
    drivv(ivdr)=vsAr3(cco)%degdp(i,j,k)
    ivdr=ivdr+1
    ---
    ---
    ---
    drivv(ivdr)=vsAr3(cco)%dradp(i,j,k)
    ivdr=ivdr+1
    drivv(ivdr)=vsAr3(cco)%dradt(i,j,k)
    ivdr=ivdr+1
    iv=iv+1
  ENDDO
ENDDO
!
CALL htif(alpv,alpov,rovv,rolv,visvv,vislv,pv,arvv,arlv,chtinv &
& ,alvnnv,drivv,watv,tlnv,tvnnv,dzzv,volv,ncellx,hlav,clv,roav,cvv &
& ,tssnv,dum1,dum1,sigv,c5p2v,c5p4v,dalvav,hgamv,hfgv,dum1 &
& ,darhsv,dtlrhsv,c5p1v,dtvrhsv,dum1,finanv,bitnv,bitv,dprhsv &
& ,dum1,dparhsv,c5p3v,hlatwv,c5p5v,chtanv,alvenv,tsnv,spifzv &
& ,vsAr(cco)%funh,vsAr(cco)%zchfn,vsAr(cco)%ztbn,vsAr(cco)%zsmsn &
& ,vsAr(cco)%zagsn,vsAr(cco)%alpan,vessTab(cco)%icrl &
& ,vessTab(cco)%icru,vessTab(cco)%icrr,vessTab(cco)%nsgrid &
& ,vsAr(cco)%zsgrd,vsAr(cco)%refld,ncellx,ncellx,1,ncellx*nthm)

```

```

!
  iv=1
  DO i=istart,iend      <<<--- put returned data in mesh arrays
    DO j=jstart,jend
      ---
      ---
      ---
      vsAr3(cco)%chtin(i,j,k)=chtinv(iv)
      vsAr3(cco)%alvn(i,j,k)=alvnv(iv)
      vsAr3(cco)%wat(i,j,k)=watv(iv)
      ---
      ---
      ---
      vsAr3(cco)%spifz(i,j,k)=spifzv(iv)
!
      iv=iv+1
    ENDDO
  ENDDO
!
  END SUBROUTINE Htif3D

```

Unlike this example with `htif`, subroutine `thermo` returns the derivative-array information, and its interface routine does the appropriate unstacking of the derivatives into the corresponding 3D mesh arrays.

The service routine calculates a pointer offset in order to access a specific thermodynamic derivative:

```

  MODULE GenHeat
!
!   BEGIN MODULE USE
  USE IntrType
!
  CONTAINS

  SUBROUTINE htif(alp,alpo,rov,rol,visv,visl,p,arv,arl,cti,alv,dr, &
  ---
  ---
  ---
  DO jj=jstart,jcell <<<--- loop over 1D or 3D cells
    j=jj
    jdr=nthm*(j-1)+1 <<<--- offset for passed derivative-array  dr
  ---
  ---
  ---

```

Arrays `bitn` and `bit`: TRAC uses bit flags to store a variety of yes/no information for all the individual mesh cells of the 1D and 3D hydrodynamic-component types. These bit flags are the individually addressed on/off (1 or 0) bit positions of the computer words in the arrays `bitn` and `bit`. For the 1D components, arrays `bitn` and `bit` are elements of derived-type `g1DArrayT`; for the 3D VESSEL component, `bitn` and `bit` are elements of derived-type `vessArray3T`.

As their storage in both new-time (i.e., bitn) and old-time (bit) arrays indicates, the bit flags can change as the state of a mesh cell changes (e.g., the direction of the vapor velocity at a 1D cell's right edge). One exception to this is the bit position that indicates if the user has chosen to employ the choking model at a cell face.

For the 1D hydrodynamic components, arrays bitn and bit are dimensioned nfaces (which is ncells + 1) by TRACAllo. For the 3D VESSEL component, bitn and bit are dimensioned (ni, nj, nk) by TRACAllo. The bit flags are accessed with the Fortran 90 intrinsic functions btest, ibset, and ibclr:

btest -- return status of requested bit position

ibset -- set requested bit to "on" (1)

ibclr -- set requested bit to "off" (0)

TRAC (Version 2.120) currently uses 30 different bit flags (total for 1D and 3D hydrodynamic components). The bit positions for the Fortran 90 bit-intrinsic functions are accessed from TRAC with parameter variables that have meaningful names. The parameter values of the bit flags are assigned in module Bits, which also has documentation on the use of each bit flag. A complete description of all of TRAC's bit flags is given in Appendix G; this includes the parameter names associated with the bit positions, the purpose of each bit, and the routines in which the bit is set and tested.

3.2.2.2.5. HTSTR-Component Type. FLT, same as for 1D: The HTSTR-component type uses array genTab for its FLT in exactly the same manner as the other components.

VLT, specific for HTSTR-component type: The HTSTR VLT data are treated by module RodVlt; its logic is the same as the other VLT modules. VLT data for individual HTSTR components are stored in derived-type array

rodTab.

Array data for the HTSTR-component-type (array hsAr) declaration: Data for all individual HTSTR arrays are stored in derived-type array

hsAr.

In module HSArray, derived data-type hsArrayT is defined, and array hsAr is declared to be of derived-type hsArrayT and dimension maxComps. A difference from the 1D hydrodynamics array g1DAr is that array hsAr is also given a TARGET attribute, and variable chs is declared to be a pointer, also of type hsArrayT (but not an array):

```
MODULE HSArray
----
----
----
```

```

TYPE hsArrayT
  REAL(sdk), POINTER, DIMENSION(:)      :: rdpwr
  REAL(sdk), POINTER, DIMENSION(:)      :: rs
  REAL(sdk), POINTER, DIMENSION(:)      :: cpowr
  REAL(sdk), POINTER, DIMENSION(:)      :: hs
  REAL(sdk), POINTER, DIMENSION(:)      :: zpwzt
  REAL(sdk), POINTER, DIMENSION(:)      :: rpwrt
  REAL(sdk), POINTER, DIMENSION(:)      :: zpwtb
  REAL(sdk), POINTER, DIMENSION(:)      :: zpwrf
  REAL(sdk), POINTER, DIMENSION(:)      :: zpw
  REAL(sdk), POINTER, DIMENSION(:, :)   :: zpwf
  REAL(sdk), POINTER, DIMENSION(:)      :: zpwfb
  ---
  ---
  ---
  REAL(sdk), POINTER, DIMENSION(:)      :: pslen
!
!   Time-Dependent Data
  REAL(sdk), POINTER, DIMENSION(:)      :: cdg
  REAL(sdk), POINTER, DIMENSION(:)      :: cdh
  REAL(sdk), POINTER, DIMENSION(:)      :: clen
  REAL(sdk), POINTER, DIMENSION(:)      :: cdgn
  REAL(sdk), POINTER, DIMENSION(:)      :: cdhn
  REAL(sdk), POINTER, DIMENSION(:)      :: clennc
!
!   Rod and slab dependent data
  REAL(sdk), POINTER, DIMENSION(:, :)   :: burn
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: cnd
  ---
  ---
  ---
  INTEGER(sik), POINTER, DIMENSION(:)   :: noht
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: cpdr
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: rndr
  REAL(sdk), POINTER, DIMENSION(:, :)   :: rpowf
!
!   Time dependent rod data
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: radr
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: radrn
  REAL(sdk), POINTER, DIMENSION(:, :)   :: drz
  REAL(sdk), POINTER, DIMENSION(:, :)   :: drzn
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: rft
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: rftn
!
!   Surface dependent rod data
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: alpr
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: alvr
  ---
  ---
  ---
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: cepwn
  REAL(sdk), POINTER, DIMENSION(:, :, :) :: cepwo
!
END TYPE hsArrayT
!

```

```

TYPE (hsArrayT), TARGET, DIMENSION(maxComps) :: hsAr <<<--- array hsAr
TYPE (hsArrayT), POINTER :: chs <<<--- pointer chs

```

The data arrays in array `hsAr` include the arrays in the TRAC-P `rodpt` and `rodpt1` pointer tables. The TRAC HTSTR component can calculate 2D heat conduction (radial and axial). Also, information may be needed for wall-to-fluid heat transfer at either one or two surfaces of an HTSTR (inner and outer). The heat conduction and transfer may be either on a coarse or a fine axial mesh. Also, the user may specify that a given HTSTR in the input deck is to be "copied," with the copies in thermal contact with various parts of a VESSEL core. A further distinction among the copies of an HTSTR is whether a given copy is "average" (affecting the temperature of the fluid it is coupled to) or "supplemental" (not affecting fluid temperature). To handle these various requirements for an HTSTR, there are data arrays in `hsAr` of rank 1, rank 2, and rank 3, with several possible dimension sizes. Examples of the allocation of data arrays in `hsAr` are given in the next section.

We emphasize that the optional copies for a given HTSTR component are a user-convenience feature for preparing an input deck; all data for the copies are contained in a single reference to (element of) `hsAr` (`cco` or `cci`) for the specific HTSTR, using the appropriate subscript in the data array to access the desired copy. Typically, the arrays for such data include a dimension of `ncrx` (the number of average rods) or `nrods` (total number of copies, including supplemental rods).

Pointer `chs` is associated with individual HTSTR components for subroutine `core1`'s calls to subroutines `htvss1` and `htcor` and for `core3`'s call to `frod`:

```

SUBROUTINE core1(dt,istd1,ndum2)
---
---
---
chs=>hsAr(cco) <<<--- associate pointer chs
---
---
---
CALL htcor(chs%rft(genTab(cco)%nodes+ins,j,ncr) &
---
---
---
```

The only reason for this pointer association is to allow the call statements to fit within a maximum of 19 continuation lines (for Fortran 77 source-format compatibility).

Three types of data arrays in `hsAr` are duplicated to contain old- and new-time quantities:

1. Time-dependent 1D arrays global to the entire HTSTR component (including its copies), or global to specific copies of an HTSTR

example: arrays `cdg` and `cdgn` for old- and new-time delayed neutron group concentrations, each of dimension (`ndgx`),

2. Time-dependent rod data defined for each copy of the HTSTR and either on the 2D conduction (radial x axial) grid or in the axial direction only

example: arrays `rft` and `rftn` for old- and new-time fine-mesh rod temperatures, each of dimension (`nodes`, `nzmax`, `nrods`)

3. The time-dependent portion of the surface-dependent rod data, defined for each copy of the HTSTR on the axial nodes at the inner and/or outer surface.

example: arrays `hrflo` and `hrfl` for old- and new-time fine mesh liquid heat transfer coefficients, each of dimension `nzmax`, `nsurf`, `nrods`.

Module `HSArray` contains service subroutines `TimeUpHS` and `TimeUpHS1`, which transfer data from new-time to old-time `hsAr` arrays, or vice-versa. Examples of the use of these routines are given in Appendix C.

Storage allocation: The data arrays in array `hsAr` are dynamically allocated at run time in much the same manner as the 1D-component arrays. There is no module `ROD`; rather, all HTSTR data allocation and input are treated in module `RodTask`. HTSTR input subroutines `rehtst` and `rhtstr` call subroutine `pntrod` for each HTSTR in the input; `pntrod` has individual calls to `TRACAllo` for each data array in `hsAr(cci)`. The order of calls to `TRACAllo` matches the order of data-array declarations in the definition of derived-type `hsArrayT`; this is not necessary, but it facilitates maintenance of the data. Some examples of the allocation of the various types of HTSTR arrays follows; the values used in the allocations already have been read from the input and stored in `genTab(cci)` or `rodTab(cci)` or have been calculated from such values [e.g., `ndm1=genTab(cci)%nodes-1` is calculated at the start of `pntrod`].

```
SUBROUTINE pntrod(lbase)
---
---
--- general global arrays:
CALL TRACAllo(hsAr(cci)%rdpwr,genTab(cci)%nodes*(1           &
&-rodTab(cci)%ipwrad), 'rdpwr', 0.0d0)
CALL TRACAllo(hsAr(cci)%rs,genTab(cci)%nodes                 &
&*mod(rodTab(cci)%nfbpwt,2), 'rs', 0.0d0)
CALL TRACAllo(hsAr(cci)%cpowr,rodTab(cci)%ncrx, 'cpowr', 0.0d0)
---
---
--- one value for each average (power) rod
---
CALL TRACAllo(hsAr(cci)%matrd,ndm1, 'matrd', 0.0d0) <<<--- nodes-1
---
---
--- dual-time global arrays:
```

```

CALL TRACAllo(hsAr(cci)%cdg,rodTab(cci)%ndgx,'cdg',0.0d0)
CALL TRACAllo(hsAr(cci)%cdh,rodTab(cci)%ndhx,'cdh',0.0d0)
CALL TRACAllo(hsAr(cci)%clen,rodTab(cci)%ncrx,'clen',0.0d0)
CALL TRACAllo(hsAr(cci)%cdgn,rodTab(cci)%ndgx,'cdgn',0.0d0)
CALL TRACAllo(hsAr(cci)%cdhn,rodTab(cci)%ndhx,'cdhn',0.0d0)
CALL TRACAllo(hsAr(cci)%clenn,rodTab(cci)%ncrx,'clenn',0.0d0)
---
      ^
--- one value for each average (power) rod
---
--- rod- and slab-dependent data:
CALL TRACAllo(hsAr(cci)%burn,ncrzp1,rodTab(cci)%nrods,'burn',0 &
&.0d0)
CALL TRACAllo(hsAr(cci)%cnd,genTab(cci)%nodes,ncrzp1 &
&,rodTab(cci)%nrods,'cnd',0.0d0)
---
---
--- dual-time rod data
CALL TRACAllo(hsAr(cci)%radr,genTab(cci)%nodes,ncrzp1 &
&,rodTab(cci)%nrods,'radr',0.0d0)
CALL TRACAllo(hsAr(cci)%radrn,genTab(cci)%nodes,ncrzp1 &
&,rodTab(cci)%nrods,'radrn',0.0d0)
CALL TRACAllo(hsAr(cci)%drz,ncrzp1,rodTab(cci)%nrods,'drz',0.0d0)
CALL TRACAllo(hsAr(cci)%drzn,ncrzp1,rodTab(cci)%nrods,'drzn',0 &
&.0d0)
CALL TRACAllo(hsAr(cci)%rft,genTab(cci)%nodes,rodTab(cci)%nzmax &
&,rodTab(cci)%nrods,'rft',0.0d0)
CALL TRACAllo(hsAr(cci)%rftn,genTab(cci)%nodes,rodTab(cci)%nzmax &
&,rodTab(cci)%nrods,'rftn',0.0d0)
---
---
--- surface-dependent rod data:
nsurf=1
IF (rodTab(cci)%idbci.GT.1.AND.rodTab(cci)%idbco.GT.1) nsurf=2
!
CALL TRACAllo(hsAr(cci)%alpr,ncrzp2,nsurf,rodTab(cci)%nrods,'alpr' &
&,0.0d0)
CALL TRACAllo(hsAr(cci)%alvr,ncrzp2,nsurf,rodTab(cci)%nrods,'alvr' &
&,0.0d0)
---
---
---
CALL TRACAllo(hsAr(cci)%stnu,rodTab(cci)%nzmax,nsurf &
&,rodTab(cci)%nrods,'stnu',0.0d0)
CALL TRACAllo(hsAr(cci)%tld,rodTab(cci)%nzmax,nsurf &
&,rodTab(cci)%nrods,'tld',0.0d0)
---
---
--- dual-time surface-dependent rod data:
CALL TRACAllo(hsAr(cci)%hrfg,rodTab(cci)%nzmax,nsurf &
&,rodTab(cci)%nrods,'hrfg',0.0d0)
CALL TRACAllo(hsAr(cci)%hrfgo,rodTab(cci)%nzmax,nsurf &
&,rodTab(cci)%nrods,'hrfgo',0.0d0)
---
---

```

3.2.2.3. Control System Databases. We refer to the "Control System data" as those sets of data that implement TRAC's signal variables, trips, and control blocks, as well as the data sets that support these capabilities. There are 14 basic types of Control System data in all:

1. a set of global data that includes 10 integers used to allocate storage based on the input specifications for the Control System [e.g., `ntsv` (the total number of signal variables)], and one REAL variable that specifies problem time (`etime`),
2. data for multipass control-parameter evaluation,
3. signal variable data,
4. control block data,
5. control block tabular data,
6. control block user-specified units labels,
7. trip-user-specified units labels,
8. signal variable user-specified units labels,
9. trip data,
10. trip-signal-expression signal data,
11. trip-controlled-trip signal data,
12. trip-set-point-factor table data,
13. trip-initiated restart dump and problem termination data, and
14. trip-initiated time-domain data.

Declaration: Module `ControlDat` handles the declaration of all Control System data. Thirteen different derived data types are defined, with a data type for all but one of the basic kinds of Control System data (only allocatable arrays are needed for the control block tabular data). The data types have names of the form:

`csNameT`.

The corresponding data sets have names of the form:

`csName`.

(The control block tabular data are in array csCBTD.)

The global Control System data are grouped into derived data-type csGlT, and variable csGl is declared to be of this type. Of the 12 other derived data types, 11 are used to declare allocatable arrays, which will be dimensioned according to the user input (the trip-initiated restart dump and problem termination data are in scalar variable csTDP, which is of type csTDPT). For 13 of the data sets, variables are also declared with a name of the form

csrName.

The csr variables are used only as scratch storage to read the restart data for the corresponding cs variable; they are deallocated after any Control System restart data are read.

Only one of the declarations of the various individual data-type elements in module ControlDat uses the pointer attribute, unlike module Gen1DArray (the pointer is required for allocatable arrays that are derived-type elements):

```
MODULE ControlDat
  ---
  ---
  ---
!   Global Data
!   -----
  TYPE csGlT                <<<--- nt variables determined from input
    INTEGER(sik) :: ntsv
    INTEGER(sik) :: ntcb
    INTEGER(sik) :: ntcf
    INTEGER(sik) :: ntrp
    INTEGER(sik) :: ntcp
    INTEGER(sik) :: ntse
    INTEGER(sik) :: ntct
    INTEGER(sik) :: ntsf
    INTEGER(sik) :: ntdp
    INTEGER(sik) :: ntsd
    REAL(sdk)    :: etime
  END TYPE csGlT
!
  TYPE (csGlT) :: csGl
  TYPE (csGlT) :: csrGl      <<<--- csr variable is used to read restart
!
  ---
  ---
  ---
!   Signal Variable Data
!   -----
  TYPE csSigT
    INTEGER(sik) :: idsv
    INTEGER(sik) :: isvn
    INTEGER(sik) :: ilcn
```

```

    INTEGER(sik) :: icn1
    INTEGER(sik) :: icn2
    REAL(sdk)    :: prevVal
    REAL(sdk)    :: presVal
END TYPE csSigT

! Dynamically dimensioned to csGl%ntsv <<<--- ntsv read from input
TYPE (csSigT),ALLOCATABLE,DIMENSION(:) :: csSig
TYPE (csSigT),ALLOCATABLE,DIMENSION(:) :: csrSig
!
---
---
---
! Control Block Tabular Data <<<--- only an array needed
! - - - - -
! Dynamically dimensioned to csGl%ntcf
REAL(sdk),ALLOCATABLE,DIMENSION(:) :: csCBTD
REAL(sdk),ALLOCATABLE,DIMENSION(:) :: csrCBTD
!
---
---
---
! Trip Set Point Factor Table Data
! - - - - -
TYPE csTSFT
    INTEGER(sik)           :: idft
    INTEGER(sik)           :: idsg
    INTEGER(sik)           :: inft
    REAL(sdk),DIMENSION(2,10) :: setp <<<--- array shape/size known
END TYPE csTSFT

! Dynamically dimensioned to csGl%ntsf <<<--- dataset is allocatable
TYPE (csTSFT),ALLOCATABLE,DIMENSION(:) :: csTSF
TYPE (csTSFT),ALLOCATABLE,DIMENSION(:) :: csrTSF
!
!
!
! Set Point Factor Table <<<--- - - - - -
- - - - -
TYPE csTDPT
    INTEGER(sik)           :: ndmp
! Dynamically dimensioned to csGl%ntdp
    INTEGER(sik),POINTER,DIMENSION(:) :: tripIDs <<<--- type element
END TYPE csTDPT

! Only One of These
TYPE (csTDPT) :: csTDP <<<--- scalar derived-type variable
TYPE (csTDPT) :: csrTDP
!
---
---
---
```

Storage allocation: Storage for the Control System data is allocated dynamically at run time, according to the user input for the signal variables, trips, and control blocks. Subroutine input reads the first five parameters in the Control System global derived-type variable csG1, first into local variables:

```

SUBROUTINE input
---
---
---
CALL readi('iiii',ntsv,ntcb,ntcf,ntrp,ntcp, 'ntsv', 'ntcb', 'ntcf',  &
& 'ntrp', 'ntcp')
---
---
---
```

Subroutine input subsequently adjusts three of these input parameters for internal use by the optional CSS logic; ntsv and ntcb are increased to allow for internally created signal variables and control blocks; and an extra pass is added for input models with more than one evaluation pass:

```

IF (.NOT.((stdyst.NE.2).AND.(stdyst.NE.4))) THEN
---
---
---

    ntsv=ntsv+ncontr+ncontt+nconts
    ntcb=ntcb+ncontr+ncontt
    IF (ntcp.GE.2) ntcp=ntcp+1
---
---
---
```

After the optional CSS logic, subroutine input stores these five local variables in the corresponding elements of csG1 and allocates storage for seven of the Control System arrays before calling the Control System input-driver rcnt1:

```

---
---
---
csG1%ntsv=ntsv
csG1%ntcb=ntcb
csG1%ntrp=ntrp
csG1%ntcf=ntcf
csG1%ntcp=ntcp
!
ALLOCATE(csSig(ntsv))    <<<--- direct use of F90 allocate
ALLOCATE(csCB(ntcb))
ALLOCATE(csCPED(ntcp))
ALLOCATE(csTrip(ntrp))  <<<--- derived-type variable
ALLOCATE(csCBTD(ntcf))  <<<--- simple array
ALLOCATE(csULCB(ntcb))
ALLOCATE(csULTR(ntrp))
```

```

!
  IF ((inlab.EQ.3).AND.(ntsv.GE.1)) WRITE (inlab,395)
395 FORMAT ('*/26('*/'*/' control-parameter data */26('*/'))
!

```

```

nrdy=1
CALL rcntl(jflag)  <<<--- Call rcntl
---
---
---

```

Subroutine rcntl, which is in module Control, reads the remaining Control System data; as part of this, it reads the remaining global data and allocates storage for the remaining Control System arrays. Note that rcntl uses the ntsv and ntcbl elements of csG1 to clear the signal variable and control-block arrays (these now include space for any internally created signal variables and control blocks) but recalculates local variables for the reading of the signal variable and control block user input:

```

MODULE Control
---
---
---
CONTAINS
---
---
---
SUBROUTINE rcntl(jflag)
---
--- recalculate local ntsv to use in reading user input:
---
IF (csG1%ntsv.GE.1) THEN
  ntsv=csG1%ntsv-cssG1%ncntr-cssG1%ncntt-cssG1%ncnts
!
! read and edit the signal-variable data cards
!
  DO nsv=1,csG1%ntsv <<<--- loop over all signal variables
    csSig(nsv)%idsv=0
    csSig(nsv)%isvn=0
    csSig(nsv)%ilcn=0
    csSig(nsv)%icn1=0
    csSig(nsv)%icn2=0
    csSig(nsv)%prevVal=0.0d0
    csSig(nsv)%presVal=0.0d0
  ENDDO
!
  IF (ntsv.GE.1) THEN
    IF (inlab.EQ.3) WRITE (inlab,95)
95  FORMAT ('*/'*/' signal variables')
    WRITE (iout,96)
96  FORMAT ('/ signal-variable data cards'/)
    DO n=1,ntsv <<<--- read user-input signal variables into locals
      CALL readi('iiii',idsv,isvn,ilcn,icn1,icn2,'idsv','isvn', &
& 'ilcn','icn1','icn2')
      IF (idsv.EQ.0) GOTO 103

```

```

IF (idsv.LT.0) idsv=-idsv
csSig(n)%idsv=idsv <<<--- store local variable in type element
csSig(n)%isvn=isvn
csSig(n)%ilcn=ilcn
csSig(n)%icn1=icn1
*
---
---
---
IF (csGl%ntcb.GE.1) THEN <<<--- local ntcb for user input:
  ntcb=csGl%ntcb-cssGl%ncntr-cssGl%ncntt
!
! read and edit the control-block data cards
!
DO n=1,csGl%ntcb <<<--- total number of control blocks
  csULCB(n)%data=0.0d0
  csCB(n)%idcb=0
  csCB(n)%icbn=0
  ---
  ---
  ---
DO n=1,ntcb <<<--- user input
  ra= ' '
  WRITE (iout,106) ra
  CALL readi('iiii',idcb,icbn,icb1,icb2,icb3,'idcb','icbn', &
& 'icb1','icb2','icb3')
  ---
  ---
  ---
  CALL readr('rrrrr',cbgain,cbxmin,cbxmax,cbcon1,cbcon2, &
& 'cbgain','cbxmin','cbxmax','cbcon1','cbcon2')
  ---
  ---
  ---
CALL cbedit(idcb,icbn,icb1,icb2,icb3)
csCB(n)%idcb=idcb <<<--- store in corresponding-type element
csCB(n)%icbn=icbn
csCB(n)%icb(1)=icb1
csCB(n)%icb(2)=icb2
csCB(n)%icb(3)=icb3
csCB(n)%cbgain=cbgain
csCB(n)%cbxmin=cbxmin
csCB(n)%cbxmax=cbxmax
csCB(n)%cbcon1=cbcon1
csCB(n)%cbcon2=cbcon2
csCB(n)%flag1=0.0d0
csCB(n)%flags=transfer('nl',1.0d0)
---
--- Read remaining global storage data and allocate
--- remaining arrays:
! read and edit the trip-DIMENSION variables card
!
IF (inlab.EQ.3) WRITE (inlab,145)
145 FORMAT ('*/* trips')

```

```

WRITE (iout,146)
146  FORMAT ('/' trip-dimension data card')
      CALL readi('iiii',ntse,ntct,ntsf,ntdp,ntsd,'ntse','ntct',
&      'ntsf','ntdp','ntsd')
      IF (ntse.LT.0) ntse=0
      IF (ntct.LT.0) ntct=0
      IF (ntsf.LT.0) ntsf=0
      IF (ntdp.LT.0) ntdp=0
      IF (ntsd.LT.0) ntsd=0
!
      ALLOCATE(csTDP%tripIDs(ntdp))    <<<--- F90 allocate statement
      ALLOCATE(csTSD(ntsd))
      ALLOCATE(csTSF(ntsf))
      ALLOCATE(csTCT(ntct))
      ALLOCATE(csTSE(ntse))
      ALLOCATE(csULSE(ntse))

```

One of the allocated arrays here is a data element of variable `csTDP`, of data-type `csTDPT`, which itself is not an array.

Dump and Restart: The Control System dump/restart logic is in module `ControlDat`, which contains subroutines `CSDump`, `CSRestart`, and `CSFree`, and in module `Control`, which contains subroutine `recntl`. To add a dump of the current Control System data to the dump/restart file, subroutine `dmpit` calls `CSDump`. The following code fragment shows the various ways in which the data in module `ControlDat` are accessed:

```

SUBROUTINE dmpit
---
---
---
  CALL CSDump
---
---
---

MODULE ControlDat
---
---
---
CONTAINS

SUBROUTINE CSDump
!
!  BEGIN MODULE USE
  USE Restart
!
  IMPLICIT NONE
  INTEGER(sik) i,ia
!
  CALL bfoutis(csGl%ntsv,1,ictrld)  <<<--- first, dump global data
  CALL bfoutis(csGl%ntcb,1,ictrld)

```

```

----
----
----
CALL bfoutis(csGl%ntse,1,ictrld)
CALL bfouts(csGl%etime,1,ictrld)
!
CALL bfoutn(csCBTD,csGl%ntcf,ictrld) <<<--- not a derived type
!
CALL bfoutis(csTDP%ndmp,1,ictrld)
CALL bfoutni(csTDP%tripIDs,csGl%ntdp,ictrld)
!
      ^
      type element is array with pointer attribute
DO i=1,csGl%ntsd
  CALL bfoutis(csTSD(i)%ndid,1,ictrld)
----
----
----
DO i=1,csGl%ntse      <<<--- loop over ntse array elements
  CALL bfoutis(csTSE(i)%idse,1,ictrld)
  CALL bfoutis(csTSE(i)%inse,1,ictrld)
  CALL bfoutis(csTSE(i)%incn,1,ictrld)
  DO ia=1,10          <<<--- loop over array within this dataset
    CALL bfoutni(csTSE(i)%ids(1:,ia),3,ictrld)
  ENDDO
  CALL bfoutn(csTSE(i)%constants,5,ictrld)
ENDDO
----
----
----

```

The basic idea of the Control System restart is that any data that are not present in the current text-input file `tracin` are obtained from the binary restart file `trcrst`. Subroutine `input` calls the restart-driver routine `rdrest` after `input` has called `rcntl` to read `tracin`. Subroutine `rdrest` makes an initial pass over the restart file to find the desired (user-specified) dump; as part of this pass, `rdrest` calls `CSRestart` to skip over the Control System portion of the various restart dumps. `CSRestart` reads a Control System dump into the `csr` scratch arrays. First, the global data are read to obtain needed array sizes, then the various arrays are allocated (including the derived-type arrays, the simple allocatable array, and the array that is a derived-type element), and then the remaining data are read. After the `csr` arrays are allocated on the first call to `CSRestart`, logical flag `csAllocate` is set to `.FALSE..`

```

SUBROUTINE rdrest(ifreex)
----
----
----
LOGICAL :: csAllocate = .TRUE.
----
----
----
!
      Control System
      CALL CSRestart(csAllocate)

```


MODULE ControlDat

CONTAINS

SUBROUTINE CSRestart(csAllocate)

!

BEGIN MODULE USE
USE Restart

!

IMPLICIT NONE
LOGICAL csAllocate
INTEGER(sik) i,ia

!

CALL bfinis(csGl%ntsv,1,ictrlr) <<<--- Global data
CALL bfinis(csGl%ntcb,1,ictrlr)
CALL bfinis(csGl%ntrp,1,ictrlr)
CALL bfinis(csGl%ntcf,1,ictrlr)
CALL bfinis(csGl%ntdp,1,ictrlr)
CALL bfinis(csGl%ntsd,1,ictrlr)
CALL bfinis(csGl%ntsf,1,ictrlr)
CALL bfinis(csGl%ntct,1,ictrlr)
CALL bfinis(csGl%ntse,1,ictrlr)
CALL bfinis(csGl%etime,1,ictrlr)

!

IF (csAllocate) THEN <<<--- allocate csr arrays
 ALLOCATE(csrSig(csGl%ntsv))
 ALLOCATE(csrCB(csGl%ntcb))
 ALLOCATE(csrTrip(csGl%ntrp)) <<<--- derived-type array
 ALLOCATE(csrCBTD(csGl%ntcf)) <<<--- simple array
 ALLOCATE(csrTDP%tripIDs(csGl%ntdp)) <<<--- array as type element
 ALLOCATE(csrTSD(csGl%ntsd))
 ALLOCATE(csrTSF(csGl%ntsf))
 ALLOCATE(csrTCT(csGl%ntct))
 ALLOCATE(csrTSE(csGl%ntse))
 ALLOCATE(csrULCB(csGl%ntcb))
 ALLOCATE(csrULTR(csGl%ntrp))
 ALLOCATE(csrULSE(csGl%ntse))
 csAllocate=.FALSE. <<<--- allocate arrays only once

ENDIF
CALL bfinn(csrCBTD,csGl%ntcf,ictrlr)
CALL bfinis(csrTDP%ndmp,1,ictrlr)
CALL bfinni(csrTDP%tripIDs,csGl%ntdp,ictrlr)

!

DO i=1,csGl%ntsd
 CALL bfinis(csrTSD(i)%ndid,1,ictrlr)

```

CALL bfinis(csrTSD(i)%ntid,1,ictrlr)
CALL bfinni(csrTSD(i)%tripIDs,5,ictrlr)
---
```

After the correct dump is found, the Control System data in that dump are first read into the csr arrays with another call to CSRestart (csAllocate is now .FALSE.); then subroutine recntl stores only the needed data (data that are not in text file tracin) from the csr arrays into the regular cs arrays; finally, the storage for the crs arrays is released with a call to CSFree, which uses the Fortran 90 deallocate statement:

```

SUBROUTINE rdrest(ifreex)
---
```

! read control parameter data

```

!
CALL CSRestart(csAllocate)
CALL recntl()
CALL CSFree
---
```

MODULE Control

```

---
```

CONTAINS

```

---
```

SUBROUTINE recntl()

```

---
```

! signal variables from the restart file that were not

```

! input on cards are added to the signal-variable data
!
```

```

jtsv=csG1%ntsv
jsav1=1
IF (jtsv.GE.1) THEN
---
```

951 csSig(jsav1)%idsv=csrSig(i)%idsv <<<--- csr to cs

```

csSig(jsav1)%isvn=csrSig(i)%isvn

MODULE ControlDat
---
```

```

    ---
CONTAINS
    ---
    ---
    ---
SUBROUTINE CSFree
!
    IMPLICIT NONE
!
    DEALLOCATE(csrSig)
    DEALLOCATE(csrCB)
    DEALLOCATE(csrTrip)
    DEALLOCATE(csrCBTD)
    DEALLOCATE(csrTDP%tripIDs)
    DEALLOCATE(csrTSD)
    DEALLOCATE(csrTSF)
    DEALLOCATE(csrTCT)
    DEALLOCATE(csrTSE)
    DEALLOCATE(csrULCB)
    DEALLOCATE(csrULTR)
    DEALLOCATE(csrULSE)
!
    END SUBROUTINE CSFree

    END MODULE ControlDat

```

3.2.2.4. Steady-State Databases. The constrained steady state (CSS): The CSS data are declared in module ControlDat. There are three derived-type variables: `cssG1` holds global data that are used for storage allocation, and `cssDat` and `cssTP` are declared as allocatable arrays. Module ControlDat also declares arrays `cpv` and `dsv`.

Subroutine input uses local variables corresponding to `cssG1` data-elements `ncontr` and `ncontp` to allocate storage for `cssDat` and `cssTP`, respectively. Subroutine input also uses local variables corresponding to `cssG1` data-elements `ncontr`, `ncontt`, and `nconts` to adjust Control System storage variable `ntsv` for internally created CSS signal variables, and it uses `ncontr` and `ncontt` for a similar adjustment to `ntcb` for control block storage. After the allocation of array `cssTP`, `cssG1%ncontp` is set to 0. The CSS data are used by module Control, by the individual component modules and by subroutines `rcomp` and `edit`. The CSS data are not included in the dump/restart file.

The HPSS: The HPSS data are declared in module HpssDat. Derived-type `hpsT` is defined; it consists of 22 arrays, all with the pointer attribute. Scalar variable `hps` is declared to be of TYPE `hpsT`. Module HpssDat also declares five other variables and initializes one of them with a DATA statement.

Subroutine input allocates storage for 18 of the arrays in `hps` with individual calls to `TRACAllo`. Subroutine `icomp` has four calls to `TRACAllo` for the remaining `hps` arrays. Subroutine input reads the HPSS input-data and does some initializing. The HPSS logic is in module Hpss, in subroutines `ihpss1` and `ihpss3`, which are called from subroutines `icomp` and `civss1` (module VessTask), respectively. The HPSS data are not included in the dump/restart file.

3.2.2.5. Radiation Model Databases. The radiation model of TRAC-P is not available in TRAC-M/F90; its database has not been translated yet.

3.2.3. Data Communication

Data Interfaces: TRAC attempts to protect the integrity of its data from inadvertent corruption as the code runs (i.e., from bugs) and to provide an easily maintainable environment for code development. To these ends, TRAC has clean interfaces within and among the 1D and 3D hydrodynamics database, HTSTR database, control system database, steady state databases, and radiation database. These interfaces exist both within the module use associations and subroutine calling chains that primarily calculate with a particular database and exist among the interactions of the various models and databases with each other. Two recent development activities have made further improvements to the code's data interfaces. The first fully separates the evaluation of terms in the flow equations from the solution of the resulting system of linear equations, providing a well-defined location for equation terms and eliminating the need for generation of this data for 1D components before evaluation of the equations in 3D components (this logic is described in Section 2).

The second development deals directly with the problem of intercomponent data communication, requiring only one request at initialization to establish automatic information passing between components. This has been implemented as a system service, with sufficient generality to permit later use by higher-order and more implicit difference methods. The System Service logic is described in Section 3.2.3.1.

Naming Conventions: Module names that end with "Task" contain task manager routines that have access to the global database. Module names that end with "Crunch" contain worker routines where the access to the array database is through their argument interface. This convention has been implemented for the HTSTR and 1D-hydrodynamics modules; for the 3D hydrodynamics, the Crunch routines directly access the 3D-mesh database (the argument lists otherwise would be prohibitively long).

Use Association: A Crunch module is used only by a corresponding Task module.

example -- Task-Crunch use association:

Module RodCrunch is used only by module RodTask and has no global access to the ROD (HTSTR) array database.

3.2.3.1. Intercomponent Communication via System Services. A request-driven communications method has been created based on requests from components for values of specifically named variables beyond end junctions. The request for information is recorded in a derived-type table that contains the address of the information needed, the address to which it must be copied, and a notation on whether a change in sign is required during the copy. Currently, these requests for boundary information are made only during the initialization phase of a calculation, with a call to subroutine InitBDArray near the beginning of subroutine init. Later, we plan to develop a dynamic communication process where the list of variables requested by a given

component can occur at any time during the calculation. This will be useful in interactive simulations or for dynamic linking to other programs.

A form of the TRAC *bd* array (see Section 2.3.1) is still in the current implementation the destination for boundary information transfers. This was selected to minimize changes to existing subroutines, which require information on conditions in an adjacent component. However, the setup and transfer subroutines do not rely heavily on this particular data structure as a destination for information and can be quickly adapted to other data structures. The current boundary data array matches the row content of the TRAC-P *bd* array. Columns of the new storage have been arranged to align with the new junction data array named *junCells* (see the next section), which provides detailed information on junction properties and connectivity in the system. In this arrangement, if *junCells(j)* provides basic information about a junction associated with a given component, then, for example, *bd(7, j)* contains the value of the old-time void fraction in the cell on the other side of that junction. This structure is illustrated in Fig. 3-2 for a simple component configuration.

3.2.3.1.1. Specification of the System Configuration. A component must register its flow connections with the system services to permit correct intercomponent communications. In older versions of TRAC, this was accomplished within input and restart subroutines (RPIPE, REPIPE, etc.) by filling in entries to the JUN array, which were used to define the JSEQ array (which is no longer used). The revised registration involves passing information to a junction cell data structure for each junction in a component with a call to subroutine *Junctions* from a component input or restart subroutine (RPIPE, REPIPE, etc.). In this context, registration is required for both standard intercomponent junction and intracomponent junctions, such as the junction of a TEE side leg to the primary leg. Arguments to *Junctions* are

```
SUBROUTINE Junctions (compNum, cellNum, junNum, compType, vOutSign, theta, phi,  
dist, ncAdj, doEdge, ix, iy, iz)
```

where the following definitions hold:

- compNum* - input-component number for the cell with this junction;
- cellNum* - number for the cell containing the junction to another component (or to the other section of the same TEE);
- junNum* - input-component junction number, or generated junction number for an internal connection;
- vOutSign* - the sign of the velocity associated with flow out from the cell through this junction face (+1 or -1);
- theta* - the angle (degrees) between an inwardly directed normal to the junction face and the primary positive direction of motion within the component;
- phi* - the angle (degrees) between an inwardly directed normal to the junction face and a reference vector perpendicular to the primary positive direction of motion within the component;

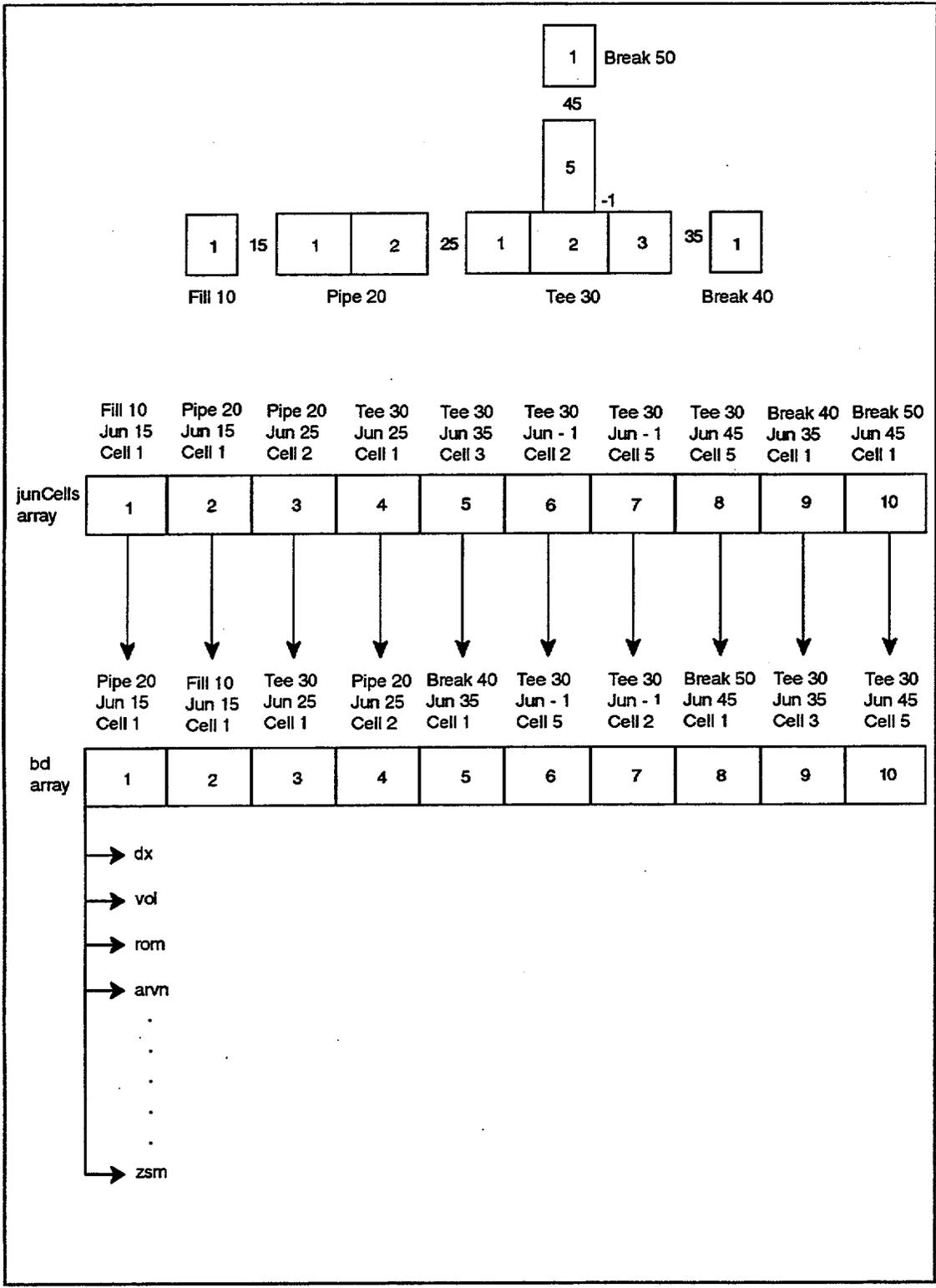


Fig. 3-2. Boundary Array Layout

- dist - the distance between the cell center and the junction face;
- doEdge - optional argument that when set to TRUE gives the component containing this cell control over the evaluation of edge-based quantities;
- ix - optional argument giving the x or radial cell index in a 3D region;
- iy - optional argument giving the y or theta cell index in a 3D region;
- iz - optional argument giving the z (axial) cell index in a 3D region; and
- ncAdj - number of cells in this component adjacent to the junction face in the direction of the inward normal to the junction face.

When calculating theta in a VESSEL, the primary positive direction of motion is taken to be the positive z direction. The reference vector for computing phi is taken to be pointing toward the center of the VESSEL. This results in values of phi of

1. zero for a connection in from the outer radial cell face;
2. 90 degrees for a connection in from the high-numbered cell theta face;
3. 180 degrees for a connection from an inner-radial cell face; and
4. 270 degrees for a connection from the low-numbered cell theta face.

For registration of an intracomponent junction such as a TEE side-leg connection, a unique junction number must be generated. This is accomplished with a reference to the function interiorJunNum, which returns a new unique (and negative) number with each call. For example, in a TEE component, the following coding would be appropriate for registration associated with the side leg:

```

junSide = interiorJunNum()
dist = .5*wjcell(jcell,cost,gldAr(cci)%hd,gldAr(cci)%dx)
angle = acos(cost)*180/pi
CALL junctions (num, jcell, junSide, 1, angle, 0, dist, 1)
CALL junctions (num, ncell1+2, junSide, -1, 0, 0,          &
               .5*gldAr(cci)%dx(ncell1+2), ncellt-ncell1-1 )

```

The subroutine Junctions installs the information from the dummy argument list into the derived-type array junCells for further processing to index and locate boundary information.

```

TYPE junctionCellsT
  INTEGER(sik):: ioc, icmp, compNum, cellNum, junNum, jcTblOrd
  INTEGER(sik):: vOutSign, otherSide, ncAdj, iEndAdj, iSeg
  INTEGER(sik) :: ivarC, ivarE, icDp
  INTEGER(sik) :: ix, iy, iz
  REAL(sdk)   :: compType
  REAL(sdk)   :: theta, phi, cosTheta, dist
  LOGICAL    :: is3D, doEdge, side
END TYPE junctionCellsT

```

```
TYPE (junctionCellsT), ALLOCATABLE, TARGET :: junCells(:)
```

The array `junComp` has been created to permit easy access to the junction information contained within the `junCells` array for any component. It is a derived-type array (of type `RangeT`), with only two components for each element (see Fig. 3-3). One array element exists for each component. Therefore, the maximum size of this array corresponds to the number of components (`ncomp`) within the input deck. As with array `junCells`, `junComp` is loaded in the order in which the components are processed during input/restart processing. The derived-type components serve as pointers (just a means of indirect addressing—they are not declared with the Fortran 90 pointer attribute) to the upper and lower elements, which bound each component's junction cell information in the `junCells` array. For example, `junComp(3)%iLB` is the index of the first entry in `junCells` for junctions in the third component processed during input, and `junComp(3)%iUB` is the index of the last entry in `junCells` for that component (see the example in Fig. 3-4).

Each component also must register general information about the computational mesh segments that it contains. This completes the picture of system connectivity and makes access of connection information simpler in mesh-based calculations. This set of information is stored in the `compSeg` derived-type array (type `SegmentT`). For purposes of this array, a mesh segment is defined as a contiguous set of adjacent cells that are contained entirely within a component. For example, this means that a PIPE, VALVE, PUMP, PRIZER, or PLENUM each contain just one mesh segment. A TEE (and the SEPD component that is based on the TEE) contains two mesh segments (one each for the main leg and side tube). Although its structure might seem to be somewhat discontinuous, a VESSEL is defined as having just one mesh segment. The FILL and BREAK do not contain any mesh segments.

Array `compSeg` clusters such mesh segment information by input component. One element is allocated for each component. Therefore, the maximum size of this array is equal to the total number of components in the system, `ncomp`. Again, the ordering of component information in this array coincides with the order in which components are processed from input. Currently, the four components to the `compSeg` structure are `nseg1D`, `nseg3D`, `seg1D`, and `seg3D` (see Fig. 3-5). The first two variables simply contain the number of 1D and 3D mesh segments owned by the corresponding component, respectively (currently taking on values of either 0, 1, or 2). The remaining two variables are derived-type arrays themselves (type `segment1DT` and `segment3DT`, respectively). The size of each array is allocated dynamically according to the values contained within `nseg1D` or `nseg3D`. Derived-type `segment1DT` stores information on the extent of data segments in 1D regions. Derived-type `segment3DT` stores information on the extent of data segments in 3D regions. It should be noted that there is intentional overlap between the junction-oriented and mesh-segment-oriented data structures to ease other data access and configuration.

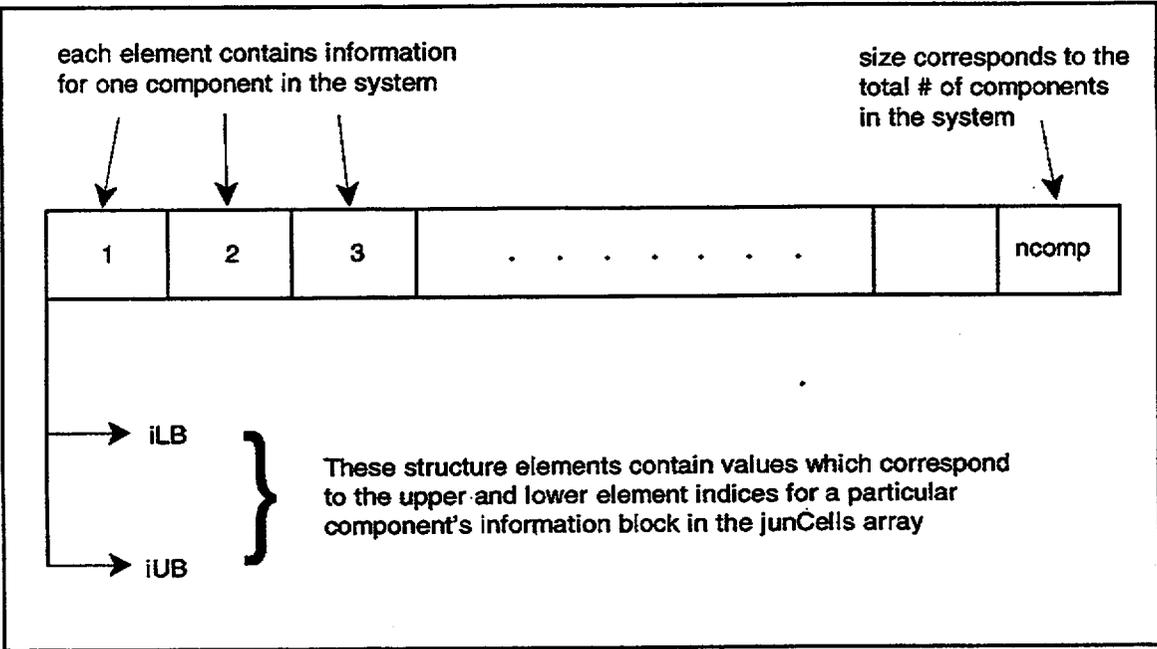


Fig. 3-3. Graphical representation of the junComp array

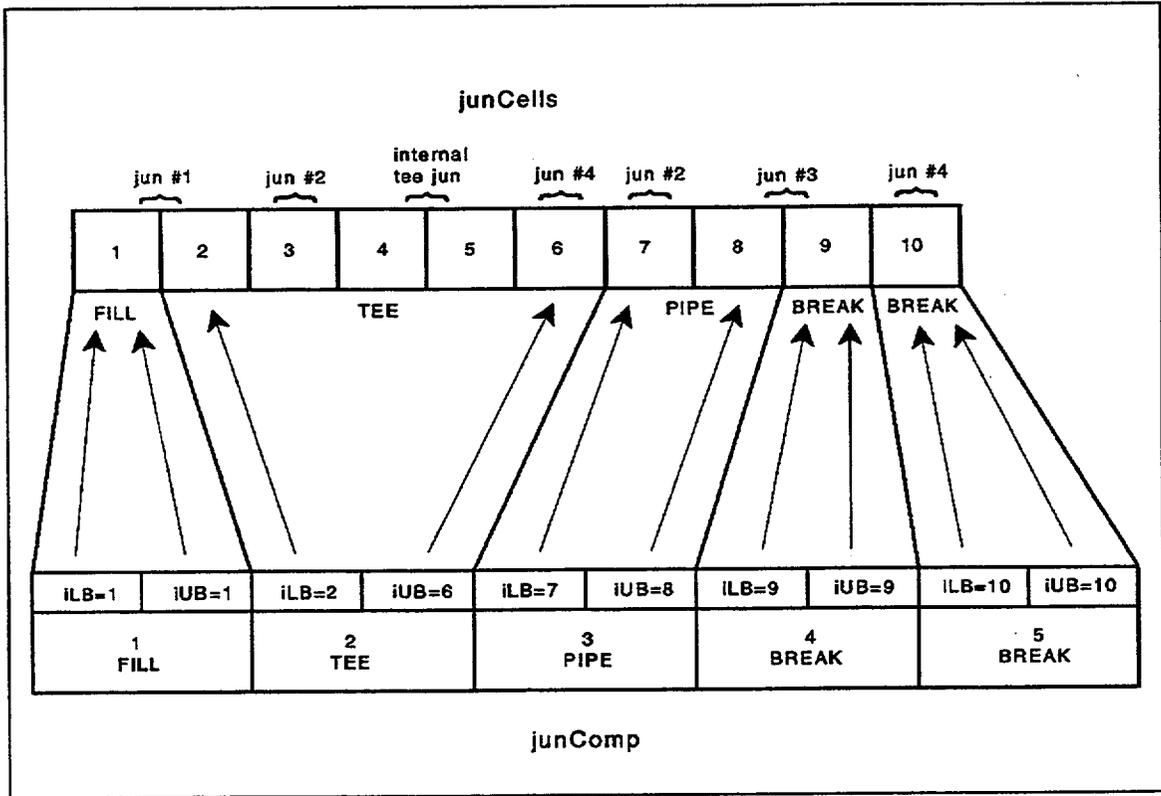


Fig. 3-4. Graphical representation of the coupling between the junCells and junComp arrays for a FILL, TEE, PIPE, BREAK, and BREAK system.

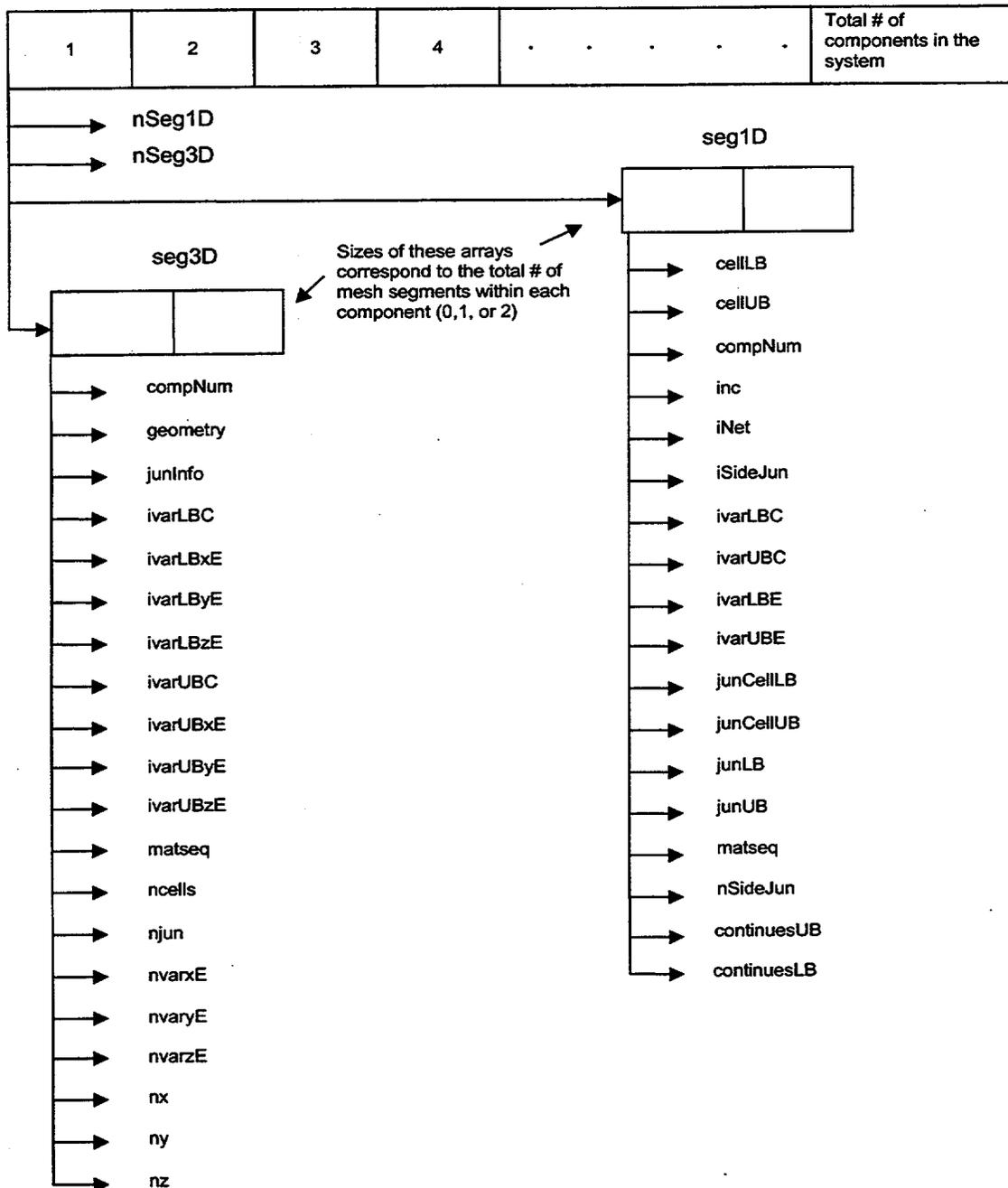


Fig. 3-5. Graphical representation of the compSeg array.

As with junCells, registration of information in the compSeg array occurs during the input/restart stage. In fact, the logic designed to accomplish this task is placed just after the call to Junctions. First, a single call to subroutine SetSegment is used to establish the number of either 1D or 3D mesh segments for the current component. As stated above, this currently can be 0, 1, or 2, depending on the component type. If the number of mesh segments to be registered is > 0, then an appropriate number of calls to either the

AddSegment1D or AddSegment3D routine is made. These routines allocate and register information in the seg1D or seg3D array components of compSeg derived-type structure arrays, respectively. The interface for each of these subroutines is structured as follows:

```
SUBROUTINE SetSegment (nSeg1D, nSeg3D)
```

where the following definitions apply:

- nSeg1D - number of 1D mesh segments in the component
- nSeg3D - number of 3D mesh segments in the component

```
SUBROUTINE AddSegment1D (compNum, iseg, cellLB, cellUB, junLB,      &
&      junUB, nsideJun)
```

where the following definitions apply:

- compNum - input-component number containing this mesh segment;
- iseg - 1D segment identifier ($0 < iseg < nSeg1D$);
- cellLB - component cell number at the lower boundary of the 1D segment;
- cellUB - component cell number at the upper boundary of the 1D segment;
- junLB - junction number at the lower boundary of the 1D segment;
- junUB - junction number at the upper boundary of the 1D segment; and
- nSideJun - number of side junctions connected to this mesh segment.

```
SUBROUTINE AddSegment3D(compNum, iseg, geometry, ncells, nvarxE,      &
&      nvaryE, nvarzE, njun, nx, ny, nz)
```

where the following definitions apply:

- compNum - input-component number containing this mesh segment;
- geometry - mesh geometry (either Cartesian or cylindrical);
- ncells - number of computational volumes in this segment;
- iseg - 3D-mesh segment identifier ($nSeg1D < iseg < nSeg3D$);
- nvarxE - number of variables at radial (x) cell edges in this segment;
- nvaryE - number of variables at theta (y) cell edges in this segment;
- nvarzE - number of variables at axial (z) cell edges in this segment;
- njun - number of junctions to other mesh segments;
- nx - number of radial (r) or x cells;
- ny - number of azimuthal (theta) or y cells; and
- nz - number of cells in the z direction.

Detailed descriptions of derived-types `junctionCellsT`, `segment1DT`, and `segment3DT` are available in Appendix C. Further information on subroutines such as `Junctions`, `AddSegment1D`, and `AddSegment3D` is provided in Appendix B.

3.2.3.1.2. Setup for Boundary Information Transfer. As previously indicated, setup for data transfer is driven by subroutine `InitBDArray`. Computational flow for this setup is summarized in Fig. 3-6. As with `icomp`, `InitBDArray` establishes a loop over each component in the system. For each junction cell within any one component, a call is placed to subroutine `SetBDJunCell`. This subroutine is responsible for establishing the necessary indices to the 1D and 3D array information that will populate the `bd` array and driving the setup of the pointer table for each boundary variable. The interface to this subroutine is

```
SUBROUTINE SetBDJunCell(compNum, junNum, cellNum, offset, bdArray,      &
&                          jindex)
```

The programmer is responsible for providing the component, junction, and cell number for the junction cell currently being processed (`compNum`, `junNum`, and `cellNum`); the number of cells/faces away from the current junction cell, which will provide the necessary boundary information (`offset`); the location in memory where this boundary information will be stored (`bdArray`); and the `junCells` index, which contains the current junction cell information (`jindex`).

One of the first tasks in `SetBDJunCell` is to generate the array indices (corresponding to the `offset` variable) for the component information to be coupled to the `bd` array elements. Indices are generated for cell-centered information (housed in variable `cellIndex`) and face-centered information for the first (the junction itself) and second faces beyond the current junction cell (`face1Index` and `face2Index`). These indices are declared as module variables and, as such, are usable by any routine during the pointer table initialization. Logic has been included to deal with special cases such as `FILL`, `BREAK`, `PLENUM`, `VESSEL`, and `TEE` components. Special geometrical considerations for these components and the associated structure of the `bd` array require that the indices to component information be generated in a unique manner.

The primary work in scheduling boundary information transfer is performed by subroutines `SetBDVar` and `AssignGen1DPtr` (in post-3.0 versions, this routine is called `AssignPtr`). After the appropriate indices are established, a call is placed to `SetBDVar` for each variable in the boundary array. The interface to this routine is given as

```
SUBROUTINE SetBDVar(localTo, offset, varName) ,
```

where

- `localTo` – location to which the boundary information will ultimately reside,
- `offset` – number of cells beyond the current junction cell for which boundary information is required, and
- `varName` – ASCII string denoting the variable to be registered in the system service pointer table.

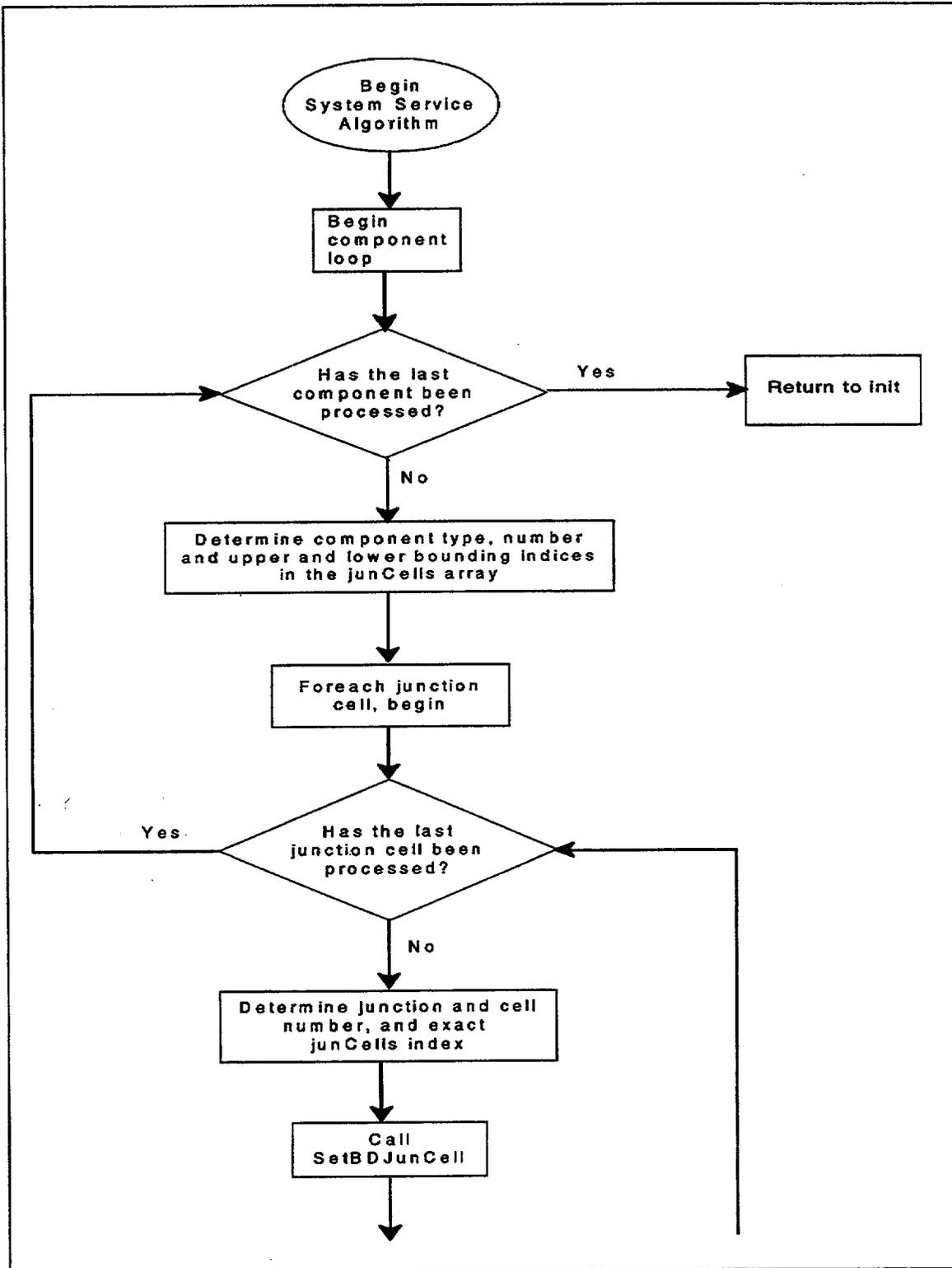


Fig. 3-6. Flow logic for System Service initialization

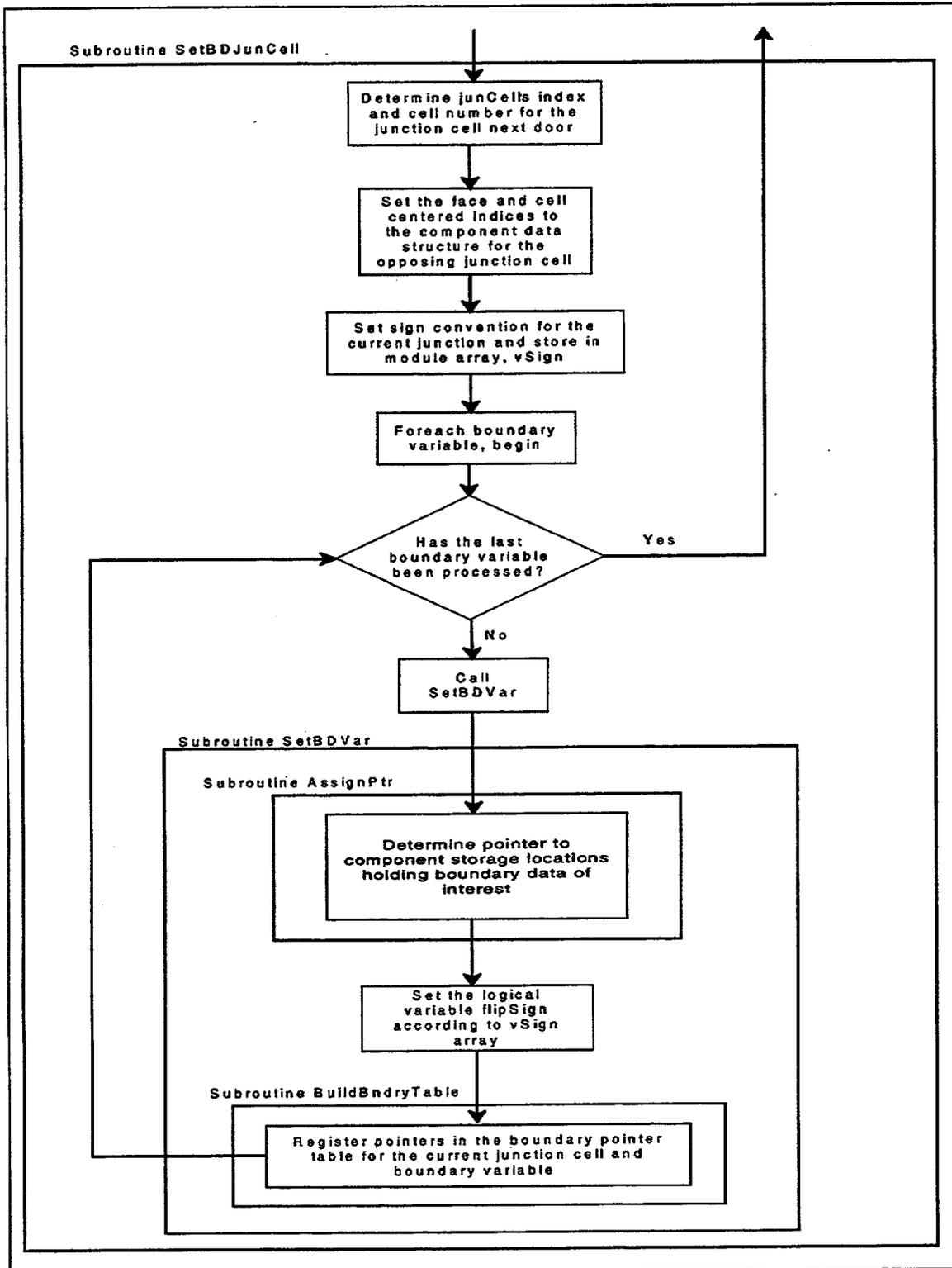


Fig. 3-6. Flow logic for System Service initialization (cont)

This routine establishes the locations to which the pointers in the transfer table will become associated, first through a call to `AssignGen1DPtr` (in post-3.0 versions, this routine is called `AssignPtr`). Using the indices established in `SetBDJunCell` and mnemonic strings for the individual `bd` variables passed through the argument list, `AssignGen1DPtr` provides a series of case statements upon which to establish the target location of data that are to be transferred to the `bd` array. This target location is returned to `SetBDVar` through the argument list.

Once the target location is created, the pointer table location for the current junction cell and variable is created via a call to `BuildBndryTable`. This low-level subroutine has the following syntax:

```
SUBROUTINE BuildBndryTable(localFrom, localTo, flipSign)
```

`LocalFrom` is a pointer to the component information arrays, and `localTo` is the boundary storage location (`bd` array) to which the component information will ultimately be transferred. `FlipSign` is a logical variable indicating whether the sign of the boundary information should be flipped when making the transfer to or from the `bd` array. Because each component maintains its own positive and negative direction vectors, it is possible to have a situation where a velocity in the boundary cell has a sign that is not compatible with the current component's convention. In other words, given two horizontal PIPES that share a common junction, defined as `jun2` in both components, and velocities that move in a left to right direction, these velocities are considered as positive in the left-hand PIPE and negative in the right-hand PIPE. When storing the right-hand PIPE's velocities in the left-hand PIPE's boundary information, it is necessary to flip the sign so that the proper conventions are followed.

3.2.3.1.3. System Service Setup Programming Guidelines. The above routines, although used exclusively for creating a newly structured `bd` array, are certainly general enough that should a programmer require a new set of boundary information for an enhanced numerical scheme or a new type of component, the amount of work required to get to that information is small in comparison to what it used to be. The most daunting task is to ensure that the appropriate indices are determined within the system service data structure and that the proper CASE statement is available within subroutine `SetBDVar`. The following is a list of programming guidelines for using and modifying the System Service:

- It is absolutely forbidden to modify the `bd` array directly in any way. All modification should be through the pointers to component-specific data-structure locations.
- If a programmer needs to add some dynamic functionality to a boundary variable, some local or global storage (depending on whether the functionality is localized to one component or applies to many) should be created and the system service data structure modified so that the appropriate boundary table elements point to this memory location. All dynamic functionality then applies to this new variable, and all appropriate

calls to Table Transfer can be used to update the information as needed. Once parallel capability is built into code, considerable planning and analysis are required before work is attempted, lest errors be introduced.

- The programmer should use `junCells`, `junComp`, and `compSeg` arrays whenever possible. This will help ensure component generality and free the code from logic that is specialized for a given component, thus making it easier to extend the code's capabilities in the future.

The following is a list of steps necessary for registering a new component in the system service data structure:

- Add a new case statement to the component loop in `InitBDArray` for the new component.
- Within this new case logic, add calls to `SetBDJunCell` for each junction cell in the component. Use `junCells` and `junComp` arrays to establish variables that are passed through the argument list.
- A survey should be conducted for each boundary variable to isolate any specialized behavior with respect to this new component. Specialized behavior implies one of several conditions: the corresponding component information in the `g1DAr`, `vsAr3`, or other applicable array is not stored in the same manner as the normal 1D or 3D components; the boundary variable has no equivalent in the new component's data structure; the nature of the solution scheme for the new component requires that the boundary variable be interpreted in a different manner with respect to the other components; or the nature of the boundary variable for the new component requires that there be no static location within the component's data structure with which to associate to the `bd` array location for that variable. If any of these conditions exist, it may be necessary to devise specialized logic to handle the boundary table set up for that boundary variable. This may include the following:
 - adding specialized logic to `SetBDJunCell` to set the indices that are used to establish the "from" pointer locations (i.e., the originating values used to populate the `bd` array). This may require adding new module variables to avoid conflicts with existing indexing variables (`face1Index`, `face2Index`, `cellIndex`, etc.).
 - adding IF THEN ELSE statements within the case statement logic in `SetBDVar` for a certain boundary variable if it does/does not require a sign flip during pointer table transfer.
 - creating a static location within the component's data structure to provide a target location to which the System Service pointer table can point.

- adding IF THEN ELSE statements within the case statement logic in AssignGen1DPtr (in post-3.0 versions, this routine is called AssignPtr) for a certain boundary variable so that the pointer to originating data location is set properly. Such logic would require using the specialized indices set in SetBDJunCell.

The following is a list of steps necessary for adding a new variable:

- add a variable to the bd array derived-type structure in module Boundary (note: the bd derived type is not currently implemented);
- increment the variable nbd by one;
- add a call to SetBDVar for the new variable in SetBDJunCell;
- add a case statement to flipSign logic in SetBDVar, if appropriate; and
- add a case statement to the associated logic in AssignGen1DPtr (AssignPtr).

If the new variable is isolated for use by only one or two components, then pointers should be set up specifically for these components. The remaining components should have this variable pointing to the nul module variable.

3.2.3.1.4. Transfer of Component Boundary Information. Once transfer has been scheduled during initialization, the actual transfer process is very simple. Given a transfer table, data are moved with a simple Fortran loop:

```

DO i = 1, bdIndex
! Copy data from one location to the other
  IF(.NOT.table(i)%flipSign) THEN
    table(i)%to = table(i)%from
  ELSE
!
!           Change the sign of the copied
!           value if such an action is called for
    table(i)%to = -table(i)%from
  ENDIF
ENDDO

```

The communication system intentionally prevents direct access by the requesting component to the storage of the requested information in the adjacent component. This is an attempt to localize errors in new components and limit poor programming practices involving alteration of data by unexpected portions of the program. It also lays a groundwork for parallel processing, providing values of communicating variables that are updated only at well-defined synchronization points in the execution of the program.

When fully implemented, initialization of intercomponent communication establishes information transfer on several different schedules. Transfer is scheduled for calculation setup only, once per timestep or once per cycle through components. Variables

containing fixed-geometry, index, or flag information are transferred only during the initialization phase. This transfer occurs at every pass through components during initialization but does not continue beyond the start of the first timestep. Some variables become "old-time" quantities simply by transfer of the "new-time" value from the previous timestep. These are scheduled for transfer at the beginning of each timestep. Of the remaining variables, some may be generated only once during a specific phase of a timestep. However, modifications to numerical methods may alter the points at which such variables are recalculated. To retain maximum flexibility, this information is transferred after each cycle through all components. Consideration can be given to further scheduling refinement after the consolidated TRAC-B/TRAC-M code reaches a higher level of maturity.

Unfortunately, some peculiarities in data flow have prevented full implementation of this scheme currently. Propagation of information from one component to the next is currently recovered by driving the portion of the data-transfer table associated with a given component whenever the component-specific driver (e.g., pipe2) has completed work. This is accomplished with the subroutine TableTransComp. Work is in progress to eliminate these data flow problems and, when possible, replace a series of calls to TableTransComp with a single call to the subroutine TableTransAll at the completion of the associated loop over all components.

3.2.3.2. Data Access—Instantiated Component with Task-Crunch Association.

The following example is based on the calling tree for the hydrodynamic outer iteration for 1D components. We concentrate here on the flow of data for a single Newton iteration and for a single specific PIPE component; the details of the convergence and backup logic and of the network-solution logic are discussed in Section 2. Examples of other analogous situations that follow the style of data access shown here are found in the 3D-hydrodynamics portion of the OUTER stage, the prep and POST stages for 1D and 3D hydrodynamics, and the ROD power logic (subroutine core1, etc.)

Subroutines trans, hout, and outer are driver routines in the hydrodynamics calling chain before the instantiation of a specific 1D component; they are standalone files and are not contained in any module. Subroutine hout loops over the network loops identified by subroutine srtlp. In turn, subroutine out1d loops over the 1D hydrodynamics components in a specific network loop and instantiates a specific 1D component, which may be of any of the 1D data types; it is not in any module.

```

SUBROUTINE trans                                     (file trans.f)
  CALL hout(oitmax,iofail,nmfail)

SUBROUTINE hout(noitmx,iofail,nmfail)             (file hout.f)
  CALL outer()

SUBROUTINE outer                                   (file outer.f)
  DO il = 1,nloops
  ---
  ---

```

```

----
imin = ig(lloopn+il-1)
imax = ig(lloopn+il)-1
CALL out1d(imin,imax,jflag) <<<--- for ilth loop

```

In subroutine `out1d`, only the type of the component to be calculated is needed; this is in the `FLT` (array `genTab` of derived-type `genTabT`, in module `Flt`). This information becomes available when index-variable `cco`, into array `genTab`, is looked up. Subroutines `pipe2`, `pump2`, `tee2`, etc., which are called by `out1d`, are intermediate-level routines that are specific for a given component type; they call the lower-level general 1D hydrodynamics routines. These routines (`pipe2`, etc.) are each contained in a module that has a name of the form "Comp_type" (e.g., module `Pipe`). The component-type modules (`Pipe`, `Pump`, etc.) are used by `out1d` only for interface checking of the subroutines they contain.

```

SUBROUTINE out1d(imin,imax,jflag)                                (file out1d.f)
!
! BEGIN MODULE USE
! USE Util
! USE CFaces
! USE IntrType
! USE OneDDat
! USE GlobalDat
! USE GlobalPnt
! USE CompTyp
! USE Flt                <<<--- FLT (module FltM.f)
! USE Global
! USE Plenum
! USE Gen1DArray
!   USE Pipe            <<<--- CONTAINS pipe1, pipe2, pipe3, etc.
!   USE Pump            <<<--- CONTAINS pump1, pump2, pump3, etc.
! USE Prizer
! USE Valve
! USE Tee
! USE Sepd
! USE Fill
! USE Break
! USE Boundary
!
! IMPLICIT NONE
!
! Declaration Generated by genImpDecs.pl 5/98
! INTEGER(sik) imax,imin,jflag
!
! controls outer calculation for one-thermal-hydraulical
! components.
!
! INCLUDE 'vellim.h'
! INTEGER(sik) idum
!

```

```

!   read component data
!
DO icmp = imin,imax           <<<---loop over components in this loop
  cco = compIndices(icmp)    <<<--- obtain cco for specific component
  icme=icme+1
!
  IF (.NOT.(genTab(cco)%type.EQ.breakh.OR.genTab(cco)%type.EQ      &
& .fillh.OR.ipakon.NE.1.OR.oitno.GT.1.OR.ibks.EQ.1)) THEN
!
!   back up to start of iteration values due to 1-d packing
!
!
  IF(genTab(cco)%type.eq.plenh) THEN
    CALL BackUpPlen
  ELSE
    CALL BackUpGen1D
  ENDIF
!
  ENDIF
  jvlim=0
  msc=0
  iphsep=0
  nc2=1
  CALL clearfldc
  varer=0.0d0
  qtp=0.d0
!
!   branch to component type
!
  IF (genTab(cco)%type.EQ.pipeh) THEN <<<---access array genTab
    CALL pipe2(jflag)           <<<--- call pipe2 (module Pipe)
  ELSEIF (genTab(cco)%type.EQ.pumph) THEN
    CALL pump2(jflag)
  ELSEIF (genTab(cco)%type.EQ.teeh) THEN
    ntee=ntee+1
    CALL tee2(jflag)
  ELSEIF (genTab(cco)%type.EQ.valveh) THEN
    CALL vlve2(jflag)
  ---
  ---
  ---
  RETURN
  END

```

Subroutine pipe2 is contained in module Pipe; it needs data from the specific PIPE VLT for the now-instantiated PIPE component being calculated, which is in array pipeTab of derived-type pipeTabT and indexed by cco. pipe2 calls general subroutine inner, which is called by all the 1D hydrodynamic components, and is contained in module Gen1DTask. Therefore, pipe2 uses modules PipeVlt (only for this PIPE's data in this case) and Gen1DTask (for subroutine interface checking). Note that component index-variable cco is in module Global; pipe2 has access to cco through module PipeVlt, which uses Global.

bd array: The bd array provides needed information from neighboring components at network junctions to the hydrodynamics Crunch routines for the beginning and end cells of a component. Details on the bd-array data members are given in Section 2.3.1. Use of the bd array in the low-level hydrodynamics routines is similar in TRAC-P and TRAC-M. However, TRAC-M has a "double-sided" bd array, with logic that distinguishes "putting" and "getting" boundary data. Also, the actual data transfers are handled in TRAC-M by the System Services.

TRAC-P bd array: In TRAC-P, the bd array is stored in the A array, and a pointer to it is set in subroutine input:

```
lvsi=ljseq+njun
lbd=lvsi+njun
lmatb = lbd + lenbd*njun
lptbln= lmatb + nmat
```

SUBROUTINE out1d passes this reference into the A array, for example, to pipe2:

```
call pipe2(a(lbd),a(lvsi),lenbd,jflag)
```

and pipe2 has a corresponding dummy argument:

```
subroutine pipe2(bd,vsi,lenbd,jflag)
---
---
---
dimension vsi(1),bd(lenbd,1)
```

TRAC-M bd array: TRAC-M declares the bd array and allocates storage for it, with a call to TRACAllo, in module Boundary. pipe2 has direct access to the bd array by using Boundary and passes the appropriate bd vector to subroutine inner via an argument list (as in TRAC-P).

```
SUBROUTINE pipe2(jflag)
!
! BEGIN MODULE USE
! USE IntrType
! USE OneDDat
! USE GlobalDat
! USE Gen1DTask <<<--- use module Gen1DTask for call to inner
! USE PipeVlt <<<--- use module PipeVlt for pipeTab;
! USE Boundary also, PipeVlt uses Global
! USE SysService <<<--- Velocity sign convention (vSign)
!
! IMPLICIT NONE
!
! Declaration Generated by genImpDecs.pl 5/98
```


Note the comments in the source code here (Version 3.0) concerning parallelism.

```

IF (bd1get(37).NE.real(genTab(cco)%num)) THEN
  ism1=istrt-1
  vmol=g1dAr(cco)%vln(ism1+1)
  ---
  ---
  ---
  g1dAr(cco)%vln(ism1+1)=g1dAr(ccoAdj)%vln(faceNum)*vs1
  ---
  ---
  ---
  CALL tf1d(bd1get,bd2get,istop) <<<--- bd columns for left and right
junctions
  ---
  ---
  ---
  END SUBROUTINE inner

```

Subroutine `tf1d` is also in module `Gen1DTask`; it is the driver for the 1D hydrodynamics Crunch routines `tf1ds1`, `tf1ds`, and `tf1ds3`, which are in module `Gen1DCrunch`. `tf1d` also calls subroutines `thermo` (module `EosNoInline`), `htif` (module `GenHeat`), and `cellav` (which is contained in `tf1d` itself). `tf1d` also performs special-case pointer associations between members of array `g1dAr`. `tf1d` is at the Task-Crunch interface for the 1D hydrodynamics; it uses modules `Gen1DArray` (for its data) and `Gen1DCrunch` (for subroutine interface checking). The Crunch routines `tf1ds1`, `tf1ds`, and `tf1ds3` have access only to `g1dAr` data through their argument lists, which are passed from `tf1d`.

```

SUBROUTINE tf1d(bd1,bd2,nc1) <<<--- bd column vectors through arg list
!
! BEGIN MODULE USE
USE IntrType
USE Gen1DArray <<<--- array g1dAr
USE Gen1DCrunch <<<--- 1D hydrodynamic Crunch routines
USE IntArray
USE OneDDat
USE Xvol
USE GlobalDat
USE GlobalPnt <<<--- pointers into array ig
USE Bad
USE Global <<<--- array ig
USE Eos <<<--- uses EosNoInline (for thermo)
USE Network <<<--- array rnet
USE GenHeat <<<--- tf1d, htif
USE JunTerms
USE SemiSolver
---
---
---
```

```

REAL(sdk) bd1(:),bd2(:),dum1(1),dum2(1)
---
---
---
IF (ipakon.EQ.1) CALL thermo(gldAr(cco)%pn,gldAr(cco)%eln &
---
---
---
CALL cellav(gldAr(cco)%alp,gldAr(cco)%vl,gldAr(cco)%fa &
---
---
---
CALL htif(gldAr(cco)%alp,gldAr(cco)%alpo,gldAr(cco)%rov &
---
---
---
CALL tflds1(gldAr(cco)%alpo,gldAr(cco)%alp,gldAr(cco)%rov &
---
---
---
& ,gldAr(cco)%bitn,bd1,bd2,ncl,gldAr(cco)%vlto,gldAr(cco)%vvto &
---
---
---
gldAr(cco)%vvx=>gldAr(cco)%vv <<<--- pointer assignment
gldAr(cco)%vlx=>gldAr(cco)%vl
---
---
---
CALL CellFluxes(gldAr(cco)%vlx,gldAr(cco)%vvx, &
---
---
---
CALL tflds(gldAr(cco)%alp,gldAr(cco)%p &
---
---
---
CALL tflds3(gldAr(cco)%alp,gldAr(cco)%p,gldAr(cco)%vlt &
---
---
---
CONTAINS
---
---
---
SUBROUTINE cellav(alp,vl,fa,grav,bd1,bd2,ncells,favol,gravol, &
& alpnm,alpmx,vlvc,fasmlt,vlalp)
---
---
---
END SUBROUTINE cellav
END SUBROUTINE tfld

```

At the Crunch level, subroutine tf1ds1 receives all of its array information about the specific 1D component through its argument list (it does access the FLT through cco):

```

SUBROUTINE tf1ds1(alpo,alp,rov,rol,vl,vv,p,vlt,vvt,vln,vvn,dr,      &
& tln, tvn, dx, hd, fa, vol, dfvdp, dfldp, wfv, wfl, cif, grav, bit, bitn, bd1,  &
& bd2, ncells, vlto, vvto, pa, dhldz, favol, sigma, vvvol, vlvol, gam, lccfl,  &
& rar1, rarv, arl, arv, xv1r, nfcvsm)

!
! BEGIN MODULE USE
USE CFaces
USE OneDDat
USE GlobalDat
USE GlobalDim
USE Ccfl
USE CompTyp
USE Bad
USE Bits
USE Flt
USE Util

!
! IMPLICIT NONE      <<<-- Note IMPLICIT NONE

!
! Declaration Generated by genImpDecs.pl 5/98
INTEGER(sik) ibit11, ibit12, ic, ichoke, ichpak, ifrcr, j, jdr, jm, jp,      &
& jside2, lccflf, msc0, msc1, msc2, ncells, ncp

!
! Declaration Generated by genImpDecs.pl 5/98
REAL(sdk) aldc, alpg, alpj, alpjm, alpl, alplm, alplp, alpm, alpmm, alpom,  &
& alppp, alpv, alpvm, alpvp, altm, aratio, beta, bond, capc, cc, cv11, cv12,  &
& cvv1, cvv2, dadt, dadx, delrho, det, dp, dprdx, dvl, dvlo, dvv, dvvo, dxdc,  &
& fact, fact1, fact11, fact12, fadc, fiht, fluxx, g, g1, g2, gamm, gamp, hfmaz,  &
& hg, omega, padc, pdc, rac, racm, racp, racpsv, ralp, rarlm, rarm, rarvm, rat,  &
& rdet, rdx, rfa, rhs1, rhsv, rl, rldc, rolm, rolp, rovm, rovp, rrlm, rrvm, rv,  &
& rvdc, sigdc, sigmab, sigmap, tldc, tvdc, vlc, vlj, vlnj, vlnmax, vlr, vltj,  &
& vltoj, vmaxtt, vmxj, vold, volj, voljm, vvc, vvj, vvnj, vvr, vvtj,  &
& vvtoj, wfm, wfp, wl, w11, wln, wln1, wm, wm1, wv, wv1, wvn, wvn1, x1, x2, x3,  &
& xfc, xfcl, xjf, xm, xwfl, xwflo, xwfv, xwfv0

---
---
---
INCLUDE 'vdvmod.h'
INCLUDE 'cflow.h'
INCLUDE 'vellim.h'
INCLUDE 'constant.h'
INCLUDE 'tst3d.h'
INCLUDE 'dtinfo.h'
REAL(sdk) lccfl(:)
REAL(sdk) alpo(:), alp(:), rov(:), rol(:), vl(:), vv(:), p(:), vlt(:),      &
& vvt(:), vln(:), vvn(:), dr(:), tln(:), tvn(:), dx(:), hd(:), fa(:), vol  &
& (:), dfvdp(:), dfldp(:), wfv(:), wfl(:), cif(:), grav(:), bit(:), bitn  &
& (:), bd1(:), bd2(:), vlto(:), vvto(:), pa(:), dhldz(:), favol(:),      &
& sigma(:), vvvol(:), vlvol(:), gam(:), rar1(:), rarv(:), arl(:), arv(:)
REAL(sdk) xv1r(:), nfcvsm(:)
LOGICAL lt1
LOGICAL lpak1, lpakr, ltees1

```

```

DATA fiht,ifrcr/1.0d0,1/
!
!
vmaxtt= max(vmaxt,vmact3)
alpmm=bd1(7) <<<--- bd array reference
ncp=ncells+1
msc0=0
msc1=0
msc2=0
!-----JSIDE2 holds index of 2nd face of tee side tube.
jside2=0
IF (genTab(cco)%type.NE.pumph) msc0=msc <<<--- genTab (FLT) reference
IF (msc0.NE.0) THEN
  msc1=msc+1
  msc2=msc+2
ENDIF
IF (islb.EQ.0) THEN
  dfvdp(jstart)=0.d0
  dfldp(jstart)=0.d0
ENDIF
IF (isrb.EQ.0) THEN
  dfvdp(ncp)=0.d0
  dfldp(ncp)=0.d0
ENDIF
!
! explicit calculation of new time velocities
!
DO j=jstart,ncp
! if(nwf.ne.0) wfl(j)=wflx
! if(nwf.ne.0) wfv(j)=wfvx
  jdr=nthm*(j-1)+1
  jp=j+1
  jm=j-1
  IF (j.NE.jstart) THEN
    voljm=vol(jm)
    alpj=alp(jm)
  ENDIF
  volj=vol(j)
  alpj=alp(j)
  vvj=vv(j)
  vlj=vl(j)
  vvnj=vvn(j)
  vlnj=vln(j) <<<--- array reference
  ---
  ---
  ---
  vln(j)=(cvv1*rhsl-cvv2*rhsv)*rdet <<<--- array reference
  ---
  ---
  ---

```

3.2.3.3. Data Access—Instantiated Component—No Task-Crunch Association.

Here we use as an example the calling chain that adds the data for a specific 1D component to the dump file, again using the PIPE:

dmpit

dpipe (dtee, dpump, etc.)

bfoutn (PIPE-specific arrays)

dcomp

bfoutn (general arrays)

GenTableDump dmpVLT

PipeTableDump

SUBROUTINE dmpit (file dmpit.f)

```
!
!
! BEGIN MODULE USE
! USE IntrType
! USE Io
! USE EngUnits
! USE GlobalDat
! USE GlobalPnt
! USE Ccfl
! USE CompTyp
! USE Flt      <<<--- genTab
! USE Control
! USE Global
! USE Temp
! USE SysTime
! USE Eos
! USE Rad
! USE Plenum
! USE Pipe    <<<--- dmpit calls dpipe
! USE Pump
! USE Valve
! USE Tee
! USE Fill
! USE Break
! USE RodTask
! USE VessTask
! USE Restart
! ---
! ---
! ---
!
! loop over components
!
!   DO icom=1, ncomp
!
!     cci=icom
!     cco=compIndices(icom) <<<--- cco
!
! 
```

```

!      branch on component type
!
      IF (genTab(cco)%type.EQ.pipeh) THEN
          CALL dpipe(icom)          <<<--- pass icomp
      ELSEIF (genTab(cco)%type.EQ.teeh) THEN
          CALL dtee(icom)
      ---
      ---
      ---

```

Subroutine dpipe calls the general 1D dump routine dcomp. dpipe also dumps arrays specific to the PIPE component, which are in module PipeArray; to do this, dpipe needs information from module PipeVlt (from pipeTab(cco)).

```

MODULE Pipe          <<<--- dpipe is in module Pipe
!
!      BEGIN MODULE USE
!      USE PipeArray  <<<--- module PipeArray
!
!      CONTAINS
!
!      SUBROUTINE dpipe(icom)  <<<--- dpipe
!
!      BEGIN MODULE USE
!      USE IntrType
!      USE PipeVlt    <<<--- array pipeTab
!      USE Restart    <<<--- bfoutn
!
!      IMPLICIT REAL(sdk) (a-h,o-z)
!
!      dumps pipe data
!
!      CALL dcomp(icom)      <<<--- pass icomp to dcomp
!      CALL bfoutn(pipeAr(cco)%powtb,iabs(pipeTab(cco)%npowtb)*2,ictrld)
!      CALL bfoutn(pipeAr(cco)%powrf,iabs(pipeTab(cco)%npowrf)*2,ictrld)
!      i2=2
!      IF (pipeTab(cco)%qp3in.LT.0.0d0) i2=1+pipeTab(cco)%ncells
!      CALL bfoutn(pipeAr(cco)%qp3tb,iabs(pipeTab(cco)%nqp3tb)*i2,ictrld)
!      CALL bfoutn(pipeAr(cco)%qp3rf,iabs(pipeTab(cco)%nqp3rf)*2,ictrld)
!      RETURN
!      END SUBROUTINE dpipe
!      ---
!      ---
!      ---

```

Subroutine bfoutn is a service routine for dumping real array data; in this case, dpipe passes references to (derived-type) elements of pipeAr(cco) to it for dumping, as well as the number of words to dump, from references to pipeTab(cco).

```

MODULE Restart
!
!
!

```

```

---
CONTAINS
---
---
---
SUBROUTINE bfoutn(aa,nwrk,ictrl) <<<--- from pipeAr and pipeTab
!
! BEGIN MODULE USE
USE Global <<<--- array Buffer
---
---
---
WRITE (ioc) (Buffer(i),i=istrt,istop) <<<--- array Buffer
---
---
---
```

Subroutine dcomp is a general routine for dumping 1D component data from the FLT, VLT, and array data from arrays g1DAr and intAr for component cco. dcomp uses a local array called aVct for reshaping array qppp (wall heat) before it is dumped. dcomp calls subroutines GenTableDump and dmpVLT to dump the FLT and VLT, respectively, and dumps the array data directly with calls to bfoutn. dcomp also needs information from the specific component's genTab. Only icomp needs to be passed to GenTableDump; dmpVLT is a driver routine for all 1D components and needs the component type also. dcomp uses module Flt both for its genTab data and for an interface to subroutine GenTableDump.

```

SUBROUTINE dcomp(icom) (file dcomp.f)
!
! BEGIN MODULE USE
USE IntrType
USE Gen1DArray <<<--- array g1DAr
USE IntArray <<<--- array intAr
USE GlobalDat
USE GlobalDim
USE CompTyp
USE Flt <<<--- genTab and
USE Global
USE Restart <<<--- contains bfoutn
---
---
---
REAL(sdk), DIMENSION(genTab(cco)%nodes*genTab(cco)%ncellt) :: aVct
---
---
---
CALL GenTableDump(icom,.TRUE.) <<<--- pass icomp & reordered flag
CALL dmpVLT(ictrl,genTab(cco)%type,icom,'dcomp') <<<--- type & icomp
---
---
---
CALL bfoutn(g1DAr(cco)%dx,genTab(cco)%ncellt,ictrl)
```

```

CALL bfoutn(gldAr(cco)%vol,genTab(cco)%ncellt,ictrld)
CALL bfoutn(gldAr(cco)%fa,genTab(cco)%ncellt+1,ictrld)
CALL bfoutn(gldAr(cco)%fric,nfrcl*(genTab(cco)%ncellt+1),ictrld)
CALL bfoutn(gldAr(cco)%grav,genTab(cco)%ncellt+1,ictrld)
CALL bfoutn(gldAr(cco)%hd,ndial*(genTab(cco)%ncellt+1),ictrld)
CALL bfoutn(intAr(cco)%nff,genTab(cco)%ncellt+1,ictrld)
CALL bfoutn(intAr(cco)%lccfl,genTab(cco)%ncellt+1,ictrld)
CALL bfoutn(gldAr(cco)%wa,genTab(cco)%ncellt,ictrld)
aVct=reshape(gldAr(cco)%qppp,shape(aVct))
CALL bfoutn(aVct,nods*genTab(cco)%ncellt,ictrld)
CALL bfoutn(intAr(cco)%matid,ndml,ictrld)
---
---
---

```

Subroutine GenTableDump has logic to access specific genTab array elements in either a reordered or nonreordered sense, depending on the value of its second input argument:

```

MODULE Flt
---
---
---
CONTAINS
!
!
SUBROUTINE GenTableDump(compInd,reordered)
!
! BEGIN MODULE USE
! USE Restart
! ---
! ---
! ---
!
LOGICAL reordered
INTEGER(sik) compInd
INTEGER(sik) ordInd
!
ordInd = compInd
if(reordered) ordInd = compIndices(compInd)
!
! ---
! ---
! ---
CALL bfoutn(genTab(ordInd)%title,4,ictrld) <<<--- dump array
! ---
! ---
! ---

```

Subroutine dmpVLT uses all the component VLT modules for interfaces to their various component-type-specific dump routines; it assumes component reordering:

```

SUBROUTINE dmpVLT(ictrl,typex,compInd,caller) (file dmpvlt.f)
!
! BEGIN MODULE USE

```

```

USE IntrType
USE CompTyp
USE Global
USE PlenVlt
USE PipeVlt <<<--- contains subroutine PipeTableDump
USE PumpVlt
USE TeeVlt
USE BreakVlt
USE FillVlt
USE ValveVlt
USE PrizeVlt
USE RodVlt
USE VessVlt
---
---
---
ordInd = compIndices(compInd)
---
---
---
IF (typex.EQ.pipeh) THEN
  CALL PipeTableDump(ordInd,caller)
ELSEIF (typex.EQ.teeh) THEN
  CALL TeeTableDump(ordInd,caller)
ELSEIF (typex.EQ.valveh) THEN
  CALL ValveTableDump(ordInd,caller)
---
---
---
ELSE
  CALL error(1,'*dmpvlt* component type not recognized ',4)
ENDIF
---
---
---
```

Subroutine PipeTableDump is contained in module PipeVlt, along with all of the PIPE-component VLTs (array pipeTab). All it needs from its caller is an index into pipeTab (the second argument is for diagnostic use).

```

MODULE PipeVlt <<<--- PIPE VLTs and related routines
!
! BEGIN MODULE USE
USE IntrType
USE Global
---
---
---
  INTEGER(sik) js2get
  INTEGER(sik) js2put
END TYPE pipeTabT
!
TYPE(pipeTabT),DIMENSION(maxComps) :: pipeTab
---
```

```

----
----
CONTAINS
!
SUBROUTINE PipeTableDump(ordInd, caller)
!
BEGIN MODULE USE
USE Restart
----
----
----
CALL bfoutn(pipeTab(ordInd)%f1, 2, ictrl) <<<--- dump array f1
CALL bfoutn(pipeTab(ordInd)%fv, 2, ictrl)
----
----
----

```

3.2.3.4. Data Access—Noninstantiated Component. Often in TRAC one of the code's databases will need information from another database. A typical case of this is in the Control System's need to access data from the component database. TRAC has a suite of service routines that is designed to provide a uniform interface to the component database; we refer to these routines as "data-access routines."

Table 3-5 lists all of TRAC's component data-access routines. For each of them, Table 3-5 indicates the module that contains the routine, the routine's name (and whether it is a function or a subroutine), the module(s) or subroutine(s) it is called from, its read/overwrite function, and its purpose.

There are data-access routines for the component FLTs (array genTab), specific hydrodynamic component-type VLTs, the general array for 1D components, the 3D VESSEL fluid mesh array, and HTSTRs.

A typical example of the use of these routines is in module Control's subroutine svset1, which determines the values of signal variables that are defined in the 1D component database:

```

SUBROUTINE svset1(isvf, isv1, isv2, icomp, ncelltx)
----
----
----
! isvn=64 : valve hydraulic diameter (m)
!
CALL GetValveTab ('ivps', icomp, ivpsx, rdum1, .TRUE.)
csSig(n)%presVal=GetGen1D(icomp, hdInd, ivpsx)
GOTO 980
ELSEIF (nsvn.GE.5.AND.nsvn.LE.7) THEN
----
----
----

```

In this example, there is first a call to GetValveTab to obtain the location (index) of a particular VALVE component's adjustable cell face; then there is a call to function GetGen1D to obtain the hydraulic diameter at that cell face; the value returned is stored

TABLE 3-5
TRAC Component Data-Access Routines^a

<u>Module</u>	<u>Routine</u>	<u>Called by</u>	<u>R/W</u>	<u>Purpose</u>
Flt	S GetGenTable	M Hpss M RodCrunch	R	Provides certain values needed from the component's genTab (FLT).
PumpVlt	S GetPumpTab	M Control	R	Provides certain values needed from PUMP component's pumpTab (VLT).
RodVlt	S GetRodTab	M Control M Rodtask	R	Provides certain values needed from the HTSTR component's rodTab (VLT).
	S SetRodTab	M RodTask	W	Overwrites the HTSTR component's rodTab (VLT) element rpowrn for coupled neutronics group.
TeeVlt	S GetTeeTab	S icomp	R	Provides certain values needed from TEE component's teeTab (VLT).
ValveVlt	S GetValveTab	M Control S input	R	Provides certain values needed from the VALVE component's valveTab (VLT).
VessVlt	S GetVessTab	M RodCrunch	R	Provides certain values needed from the 3D VESSEL component's vessTab (VLT).
Gen1DArray	F GetEosDriv1d	M RodTask	R	Returns EOS data from the 1D-component database.
	F GetGen1D	M Control M Hpss M RodTask	R	Returns the value of the desired component 1D-array element.
	F GetGen1D2D	M Control	R	Returns the value of the desired 1D-component, 2D-array element.
	S GetGen1DArray	M Control	R (see Coding Std. below)	Returns the value of a pointer that is associated with a desired component 1D array.

TABLE 3-5—TRAC Component Data-Access Routines^a (cont)

<u>Module</u>	<u>Routine</u>	<u>Called by</u>	<u>R/W</u>	<u>Purpose</u>
	S Get1DArrayPointer	M Gen1DArray (see Purpose)	---	Associates the pointer with the desired 1D-component 1D array. <u>Service routine for:</u> GetGen1D GetGen1DArray CopyGen1DArray IncrementGen1D
	S Get2DArrayPointer	M Gen1DArray (see Purpose)	---	Associates the pointer with desired component 2D array. Only set up for twm array (uses array name). <u>Service routine for:</u> GetGen1D2D
	S IncrementGen1D	M RodTask	W	Adds the passed value to the value of the specified component 1D-array element. The new value replaces the original value in the array.
	S CopyGen1DArray	M Control	R	Copies a specified number of 1D-component array elements into an array in the calling routine.
HSArray	F GetHS	M Control	R	Returns the value of the desired HTSTR component surface-array element (assumes outer surface).
	F GetHSSurf	M Control	R	Returns the value of the desired HTSTR-component surface-array element (assumes outer surface).
	F GetNoht	M Control	R	Returns the value of noht (number of rows of heat-transfer nodes) for a specified copy (ROD) of the desired HTSTR.
	F GetHS2d	M Control	R	Returns the value of the desired HTSTR-component, 2D-array element.
	F GetHS3d	M Control	R	Returns the value of the desired HTSTR-component, 3D-array element.

TABLE 3-5—TRAC Component Data-Access Routines^a (cont)

<u>Module</u>	<u>Routine</u>	<u>Called by</u>	<u>R/W</u>	<u>Purpose</u>
	S GetHS1DPtr	M HSArray (see Purpose)	---	Associates the pointer with the desired HTSTR-component 1D array. <u>Service routine for:</u> GetHS
	S GetHS2DPtr	M HSArray (see Purpose)	---	Associates the pointer with the desired HTSTR-component 2D array. <u>Service routine for:</u> GetHS2d
	S GetHS3DPtr	M HSArray (see Purpose)	---	Associates the pointer with the desired HTSTR-component 3D array. <u>Service routine for:</u> GetHSSurf GetHS3d
HeatArray	S GetHeatArray	M Control	R (see Coding Std. below)	Returns the value of a pointer that is associated with a desired data array in heatAr (part of the 1D-component database).
VessArray3	F GetVSAR	M Control	R	Returns the value of the element (i,j,k) of the specified VESSEL 3D mesh array.
VessCrunch	S copya	M VessCrunch	R	Copies data for one level from one array to another.

^a In columns 2 and 3, "S" = a subroutine, "F" = a function, and "M" = a module.

in the Control System database. Another important use of data-access routines is found in the transfer of data between the hydrodynamic and HTSTR databases. An example is given in a separate section below.

Some of TRAC's data-access routines assume that the components have been reordered. Typically, the routines in this category are called with an `icmp` loop-index actual argument, contain a `compInd` dummy argument, and have a

```
ordInd = compIndices(compInd)
```

statement.

Some of the data-access routines have reordering logic that is driven by a `.TRUE./ .FALSE.` actual argument and a "reordered" dummy argument and have these statements:

```

ordInd = compInd
if(reordered) ordInd = compIndices(compInd)

```

Subroutine `GetGen1DArray` returns the value of a pointer variable that is associated with the beginning of a specific desired array in the component 1D hydrodynamic-array database. The pointer array in the caller's actual argument list can be subsequently used in arithmetic statements in the caller, typically on the right-hand side, but potentially on the left-hand side. Similarly, subroutine `GetHeatArray` returns a pointer value from array `heatAr`, which is also part of the 1D database. Subroutine `CopyGen1DArray` copies the desired component 1D array data into an array (not a pointer array) in the calling routine.

Coding Standard: Pointer values returned by subroutines such as `GetGen1DArray` and `GetHeatArray` should be used only on the right-hand side of assignment statements.

Subroutines `Get1DArrayPointer` and `Get2DArrayPointer` are service routines in module `Gen1DArray`, which associate an array pointer variable with a desired 1D-component array. For computational efficiency, `Get1DArrayPointer` operates with a select case construct, using array-index numbers that are parameterized in module `Gen1DArray`. `Get2DArrayPointer` only has to associate the wall temperature array `twm` and uses an IF statement on the array name. Both routines will fall through to an error message if an array is called for which the routines are not set up to handle.

Function `GetEosDriv1d` is specially set up to return one of four EOS variables from the 1D component database. The 1D EOS data are stored in "inverted" form and are accessed by appropriate offsets into array `driv`, which is in `g1DAr`.

3D VESSEL Data Access—scratch storage use of old-time arrays: Real function `GetVSAR`, contained in module `VessArray3`, returns a value from a subset of the VESSEL 3D mesh arrays. `GetVSAR` takes as input arguments a character string specifying the desired array name, the VESSEL component index `cco`, and the (i, j, k) indices into the array.

Subroutine `svset3`, in module `control`, is responsible for evaluating all signal variables that are defined in a 3D VESSEL component. `svset3` calls `GetVSAR` to obtain all needed information from a VESSEL. Note that `svset3` uses certain old-time VESSEL arrays as scratch storage for intermediate calculations. The following code fragments show use of the old-time VESSEL arrays `vlyt`, `vlz`, and `vlxr` as scratch storage for determining liquid-mass-flow signal variables; also shown are signal variables that do not need the scratch storage.

```

SUBROUTINE svset3(isvf, isv1, isv2, icomp)
---
---
---
!   isvn=31, 32 or 33 : cell lower interface liquid mass flow (kg/s)
!

```



```

        vsvName='vlnyt  '
    ELSEIF (m.eq.1) THEN
        vsvName='vlz  '
    ELSEIF (m.eq.2) THEN
        vsvName='vlx  '
    ENDIF
    GOTO 850

```

```

---
---

```

--- **calling GetVSAR (may be accessing scratch array):**

```

!   isvn is negative : signal value is the parameter difference
!

```

```

        vsv=GetVSAR(vsvName,cco,i1,j1,k1)
        csSig(n)%presVal=vsv
        vsv=GetVSAR(vsvName,cco,i2,j2,k2)
        csSig(n)%presVal=csSig(n)%presVal-vsv
        GOTO 980
    ENDIF

```

```

!
!   the signal value is from cell1 when cell2 is zero
!   or from cell2 when cell1 is zero
!

```

```

        vsv=GetVSAR(vsvName,cco,i1,j1,k1)
        csSig(n)%presVal=vsv
    ENDIF

```

Examples of Data-Access-Routine Coding: Examples of the coding for the hydrodynamic data-access routines, including their argument lists, are given in Appendix G.

HTSTR Data Access Routines: The HTSTR data-access routines are contained in module HSArray, thus providing information to the Control System. They work in much the same way as the data-access routines for the general 1D hydrodynamic array. However, the lowest-level service routines GetHS1DPtr, GetHS2DPtr, and GetHS3DPtr all operate with IF statements on array name strings that are passed to them (they do not use SELECT CASE).

The HTSTR data-access routines GetHS2d and GetHS3d assume that the requested array thermal hydraulics are organized according to

```

REAL(sdkx) FUNCTION GetHS2d(compInd,arrayName,rod,cell)
---
---
---
CALL GetHS2DPtr(arrayName,compInd,arPtr)
GetHS2D=arPtr(cell,rod) <<<--- cell, rod (i.e., copy)
---
---
---
REAL(sdkx) FUNCTION GetHS3d(compInd,arrayName,rod,cell,node)
---
---

```

```

---
CALL GetHS3DPtr(arrayName, compInd, arPtr)
GetHS3D=arPtr(node, cell, rod) <<<--- node, cell, rod
---
---
---
```

These functions will return a value from any rank-2 or rank-3 array (assuming it is made available in the Ptr routines); it is up to the calling routine to know the ordering of information in the array columns. All of the "surface" HTSTR data arrays are in rank-3 arrays, organized according to

(axial node-row, inner/outer surface index, rod index)

Function GetHSSurf currently needs to return only outer-surface data; therefore, only a rod index and a node-row index are passed to it:

```

REAL(sdkx) FUNCTION GetHSSurf(compInd, arrayName, rod, cell)
---
---
---
! This was implicit in the old code:
! (for isvn=91 or 92, only the outer surface is accessed,
! irrespective of input)
! is = 1
!
CALL GetHS3DPtr(arrayName, compInd, arPtr)
GetHSSurf=arPtr(cell, is, rod)
!
END FUNCTION GetHSSurf
```

Function GetNoht is hardwired to return a value from the noht array for a specified copy (ROD) of a specified HTSTR (array noht carries the number of rows of heat-transfer nodes for the rod in question).

3.2.3.5. HTSTR to Fluid Data Communication.

Note: HTSTR to Fluid Data Communication. In future code versions, this logic will be replaced.

HTSTR arrays lchci and lchco: The HTSTR data array hsAr includes the rank-two arrays lchci and lchco, which carry information about the hydrodynamic components to which the inner and outer surfaces of an HTSTR component are coupled. The first subscript of lchci and lchco contains the cell number (for 1D hydrodynamic components) or the reordered component index (for 3D VESSEL components) and the type of the hydrodynamic component. The second subscript contains indices of the coarse-node, heat-conduction rows. Arrays lchci and lchco are initialized in the INIT stage by subroutines irod1, lchpip, and lchvss (module RodCrunch).

Data-copy subroutines fltom, piprod, and vssrod: Currently in TRAC, HTSTR components may be thermally coupled to the 1D and 3D hydrodynamic components

(but not the PLENUM, BREAK, or FILL components). In the TRAC Prep stage, HTSTR components need fluid information from the 1D and 3D hydrodynamic component databases to calculate heat-transfer coefficients and related quantities. The results from the HTSTR metal-to-fluid calculations subsequently are needed by the 1D and 3D hydrodynamic components in the Outer stage. In the Post stage, the HTSTRs will need the new-time fluid temperatures as boundary conditions for their internal-heat-conduction solution. The required data are transferred (copied) between the HTSTR and the 1D and 3D hydrodynamic databases by service subroutines piprod (module RodTask for the 1D hydrodynamics components) and vssrod (module RodTask for the 3D VESSEL component). piprod and vssrod are driven by subroutine fltom (module RodTask):

```

SUBROUTINE htstr1
---
---
---
DO icmp=1,nhtstr
  cci=icmp+ncomp
  cco=compIndices(cci)
  CALL fltom (hsAr(cco)%lchci,hsAr(cco)%lchco,hsAr(cco)%idrod,1)
  ^
transfer hydrodynamic data to HTSTR
---
---
---
SUBROUTINE fltom (lchci,lchco,idrod,imfl) <<<--- lchci and lchco passed
---
---
---
!       IMFL - FLAG INDICATING OPERATION TO BE PERFORMED
!       = 1, MOVE HYDRO INFO INTO ROD DATA DURING PRE-PASS
!       =-1, MOVE ROD DATA INTO HYDRO DATA DURING PRE-PASS
!       = 2, MOVE NEW HYDRO FLUID TEMP'S INTO ROD DATA
!           DURING POST-PASS
---
---
--- inner HTSTR surface:
      ctyp = lchci(2,nzz)      <<<--- component type
      IF (ctyp.EQ.vsslh) THEN
        nz1=nz1+1
        idum=hsAr(cco)%ntsxx(mrd)
        CALL vssrod(int(lchci(1,nz)),int(idrod(nrd)),idum      &
&      ,int(hsAr(cco)%hceli(nz)),nz1,imfl,ncr,rodTab(cco)%iis,nz)
      ELSEIF (ctyp.NE.plenh) THEN
        CALL piprod(int(idrod(nrd)),ig(lorder)      &
&      ,int(hsAr(cco)%hceli(nzz)),int(hsAr(cco)%hcomi(nzz)),imfl &
&      ,ncr,rodTab(cco)%iis,nz)
      ENDIF
---
---
--- outer HTSTR surface:

```

```

        ctyp = lchco(2,nzz)      <<<--- component type
        IF (ctyp.EQ.vsslh) THEN
            nz2=nz2+1
            idum=hsAr(cco)%ntsxx(mrd)
            CALL vssrod(int(lchco(1,nz)),int(idrod(nrd)),idum      &
&            ,int(hsAr(cco)%hcelo(nz)),nz2,imfl,ncr,isurf,nz)
        ELSEIF (ctyp.NE.plenh) THEN
            CALL piprod(int(idrod(nrd)),ig(lorder)                &
&            ,int(hsAr(cco)%hcelo(nzz)),int(hsAr(cco)%hcomo(nzz)) &
&            ,imfl,ncr,isurf,nz)
        ENDIF
    ---
    ---
    ---

```

In subroutine piprod, component-index cco accesses the current HTSTR component. piprod uses data-access routines IncrementGen1D, GetGen1D, and GetEosDriv1d (the index i that is passed to these routines is obtained from comparison of the iorder array with ihcom):

```

    SUBROUTINE piprod (idrod,iorder,ihcel,ihcom,imfl,ncr,isurf,kz)
    ---
    ---
    ---
    DO i=1,ncomp
        IF (iorder(i).EQ.ihcom) GOTO 20      <<<--- obtain hydrodynamic-
component index
        ENDDO
    20 CONTINUE
    ---
    ---
    ---
    !   move rod data to hydro data      <<<--- Prep-stage call
    !
        IF (idrod.GE.0) THEN
            CALL IncrementGen1D (i,hgamInd,aihcel,                &
&            hsAr(cco)%hgamr(kz-1,isurf,ncr))
            ---
            ---
            ---
            CALL IncrementGen1D(i,finanInd,aihcel,                &
&            hsAr(cco)%finar(kz-1,isurf,ncr))
            ENDIF
        !
        !   move hydro data to rod data
        !
        ELSEIF (imfl.GE.2) THEN      <<<--- Post-stage call
        !
            hsAr(cco)%tlnr(kz,isurf,ncr)=GetGen1D(i,tlnInd,aihcel)
            hsAr(cco)%tvnr(kz,isurf,ncr)=GetGen1D(i,tvnInd,aihcel)
            ELSE      <<<--- Prep-stage call
        !
            xalp=GetGen1D(i,alpInd,aihcel)

```

```

hsAr(cco)%alpr(kz, isurf, ncr)=xalp
hsAr(cco)%alvr(kz, isurf, ncr)=GetGen1D(i, alvInd, aihcel)
hsAr(cco)%cplr(kz, isurf, ncr)=GetGen1D(i, cplInd, aihcel)
---
!
hsAr(cco)%sr(kz, isurf, ncr)=GetGen1D(i, sInd, aihcel)
!
hsAr(cco)%drvdt(kz, isurf, ncr)= GetEosDriv1d(i, 'drvdt  ', aihcel)
hsAr(cco)%drltd(kz, isurf, ncr)= GetEosDriv1d(i, 'drltd  ', aihcel)

```

Subroutine `vssrod` performs a function analogous to that of `piprod` for the 3D VESSEL component: it is also called by subroutine `fltom` and handles all HTSTR-to-VESSEL and VESSEL-to-HTSTR data copies. `vssrod` also uses component Index `cco` to access the current HTSTR component. Unlike `piprod`, `vssrod` receives component Index `ccov` through its argument list to access the required hydrodynamic (VESSEL) component (from the `lchci` and `lchco` arrays in subroutine `fltom`). `vssrod` operates directly on the VESSEL arrays. In addition to the VESSEL 3D mesh data, `vssrod` treats some of the VESSEL Special Array Data.

```

SUBROUTINE vssrod(ccov, idrod, ntsxx, iz, nz, imfl, ncr, isurf, kz)
---
! move rod data to hydro data
!
data in module RodHtcref1 copied to VESSEL special arrays:
  IF (idrod.GE.0) THEN
    IF ((newrfd.EQ.1).AND.(nz.EQ.1)) THEN
      ij=ias+(ir-1)*ntsxx
      IF (nrefld(ij).EQ.1) vsAr(cco)%refld(ij)=1
      IF (int(nhsca(ij)).EQ.genTab(cco)%num) THEN
        vsAr(ccov)%alpan(ij)=alpag2(ij)
        vsAr(ccov)%alpcn(ij)=alpcf2(ij)
        ---
        ---
        vsAr(ccov)%ztbn(ij)=ztb(ij)
      ENDIF
    ENDIF
  ENDIF
VESSEL mesh arrays:
  vsAr3(ccov)%hgam(i, j, k)=hsAr(cco)%hgamr(kz-1, isurf, ncr)      &
& +vsAr3(ccov)%hgam(i, j, k)
  vsAr3(ccov)%hla(i, j, k)=hsAr(cco)%hlar(kz-1, isurf, ncr)      &
& +vsAr3(ccov)%hla(i, j, k)
  ---
  ---
  vsAr3(ccov)%finan(i, j, k)=hsAr(cco)%finar(kz-1, isurf, ncr)   &
& +vsAr3(ccov)%finan(i, j, k)
  ENDIF
!

```

```

!   move hydro data to rod data
!
      ELSEIF (imfl.GE.2) THEN
!
      hsAr(cco)%tlnr(kz, isurf, ncr)=vsAr3(ccov)%tln(i, j, k)
      hsAr(cco)%tvnr(kz, isurf, ncr)=vsAr3(ccov)%tvn(i, j, k)
      ELSE
!

```

copy into module RodHtcref1 arrays:

```

      IF ((newrfd.EQ.1).AND.(nz.EQ.1)) THEN
        ij=ias+(ir-1)*ntsxx
        funh(ij)=vsAr(ccov)%funh(ij)
        nhsca(ij)=vsAr(ccov)%nhsca(ij)
        alpag2(ij)=vsAr(ccov)%alpag(ij)
        alprw(ij)=vsAr(ccov)%alprw(ij)
        alpsm(ij)=vsAr(ccov)%alpsm(ij)
        zags(ij)=vsAr(ccov)%zags(ij)
        zdfs(ij)=vsAr(ccov)%zdfs(ij)
        zrws(ij)=vsAr(ccov)%zrws(ij)
        zsms(ij)=vsAr(ccov)%zsms(ij)
      ENDIF

```

copy into hsAr arrays:

```

      hsAr(cco)%alpr(kz, isurf, ncr)=vsAr3(ccov)%alpn(i, j, k)
      hsAr(cco)%alvr(kz, isurf, ncr)=vsAr3(ccov)%alvn(i, j, k)
      hsAr(cco)%cplr(kz, isurf, ncr)=vsAr3(ccov)%cpl(i, j, k)
      hsAr(cco)%cpvr(kz, isurf, ncr)=vsAr3(ccov)%cpv(i, j, k)

```