Title: SOFTWARE DESIGN IMPLEMENTATION DOCUMENT FOR TRAC-M DATA STRUCTURES

Author(s): Susan J. Jolly-Woodruff, Ogden
John Mahaffy
Paul T. Giguere
James F. Dearing
Brent E. Boyack

Submitted to: Nuclear Regulatory Commission

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
SOFTWARE DESIGN IMPLEMENTATION DOCUMENT
FOR
TRAC-M DATA STRUCTURES

by

S. Jolly-Woodruff
Ogden Environmental and Energy Services

J. Mahaffy
Pennsylvania State University

P. Giguere, J. Dearing, and B. Boyack
Los Alamos National Laboratory

EXECUTIVE SUMMARY

The Transient Reactor Analysis Code (TRAC)-M system-wide and component data structures are to be reimplemented by using the new features of Fortran 90 (F90). There will be no changes to the conceptual design, data flow, or computational flow with respect to the current TRAC-P, except that readability, maintainability, and extensibility will be improved. However, the task described here is a basic step that does not meet all future needs of the code, especially regarding extensibility. TRAC-M will be fully functional and will produce null computational changes with respect to TRAC-P, Version 5.4.25; computational efficiency will not be degraded significantly. The existing component and functional modularity and possibilities for coarse-grained parallelism will be retained.

F90 modules will replace common blocks. All container arrays and integer offset pointers will be eliminated. All arrays will be individually dynamically allocated, with their F90 pointers or arrays placed in modules. The fixed-length, variable-length, and pointer tables comprising the component data structures will be replaced with arrays of F90-derived types. The first two will contain the same data elements as before, whereas the pointer tables will contain F90 pointers. Variable names will remain the same to the extent possible; F90 pointers will use the same name as the corresponding integer pointers, with omission of the leading letter "L." All data structures will be documented internally and in the updated Programmer's Guide.

Because of the complexity of the current coding and some F90 implementation details that will become clear only with in-place prototyping, some features of the final implementation may differ from that described in this document. Consequently, this Software Design Implementation
Document (SDID) must be treated as a living document and updated quarterly until work is completed.

Symbolic names are included in the SDID for heuristic purposes only and may be changed in the final implementation.

All sample code is included for heuristic purposes only and may not represent actual coding in the final implementation.

**Related Documentation.** This SDID is one of several pieces of new documentation, some portions of which will be incorporated into the updated version of the Programmer’s Guide to be delivered at the end of this task. Additional new documentation includes, but is not limited to the

- **Application Programming Interface (API):**

  The API describes how programmers are to make all levels of extensions to TRAC. Details on such high-level changes as adding a new component to the TRAC network will be documented thoroughly. Information also will include the consistent use of specialized low-level routines, such as the TRAC interface for dynamic memory allocation, procedures for information exchange between data structures, and required data management such as time update and consistent dump/restart. The estimated effort to prepare this document is 4 staff months.

- **documentation of new subroutines:**

  This documentation will describe the functionality of all new routines. It will explain the purpose of new functionality and/or the basis for reorganization of current functionality and any resulting changes to data flow. It will include an explanation for deletion of original routines, such as those used to implement the former graphics dump and nonstandard bit manipulations. A brief and initial presentation of this information appears in App. E of this SDID.

- **coding standards and programming practices:**

  This documentation will address details of programming practices, including the use of F90 KIND attribute, IMPLICIT NONE declaration, use of labels for DO loops and other control constructs, guidelines for use of CASE construct vs IF-THEN-ELSE, and similar generic issues. A brief and initial presentation of this information appears in App. F of this SDID.
• TRAC architecture and computational flow:

This documentation will describe and explain the rationale for the conceptual design of the TRAC architecture. It will document the computational flow and relate this to the code architecture. The success metric for this documentation is that by its organization and level of detail, it can be used by a competent code developer to become sufficient familiar with TRAC to modify or add to the code.

This is an additional document, separate from both the SDID and the Programmer’s Guide. Preparation of this document will be scheduled immediately following preparation of the Programmer’s Guide. One reason for this conclusion is that much of the essential information, e.g., variable definitions, final computational flow, and code architecture, will not be fully available until the database modernization effort approaches completion. A second reason is the unavailability of staff to perform this effort concurrently with the database modernization effort. This activity currently is unfunded.
I. INTRODUCTION

This is the Software Design Implementation Document (SDID) for the re-implemented data structures in the first major version of the Transient Reactor Analysis Code (TRAC) to use the new features of Fortran 90 (F90). For the purposes of this document, we will refer to this forthcoming version of TRAC simply as TRAC-M. The starting point for TRAC-M will be the existing TRAC-M, Version 2.0, which is the first version of TRAC to be compiled under F90. TRAC-M, Version 2.0, which makes only minimal use of new F90 features, is a portable, restructured version of TRAC-P, Version 5.4.25.

Data structure has a direct and significant impact on three of the major activities in any software project: new development, maintenance, and verification. This leads to three requirements on the data structure, its interaction with the code structure, and its associated documentation:

1. Implementation must result in a program that can be extended during development activities that add new capabilities requiring new variables or new data structures.

2. Implementation must result in a program that is relatively easy to maintain. The data structure is amenable to minor changes and is designed so that those changes have a low chance of introducing errors into existing code functions.

3. Implementation must result in a readable program to accelerate new development, maintenance, and verification activities. In this instance, “readable” means that an individual with experience in
two-phase thermal-hydraulics and F90 programming will be able to recognize and use data describing the physical state of the system.

The data structure must be compatible with the existing TRAC program structure that is modular by system component as well as by code function. In addition, to avoid a cascade of additional code modifications, it must permit existing low-level TRAC subroutines to continue operating on arrays of physical data (calculate properties for multiple computational cells) and, where appropriate, to assume validly that these arrays correspond to a single physical component.

Starting from an existing program imposes additional practical constraints on functionality, testing, and code performance. All deliverable\(^1\) versions between TRAC-M, Version 2.0 and TRAC-M will be fully functional and will produce null computational changes with respect to TRAC-P, Version 5.4.25. Moreover, we will impose the requirement that the reimplemented data structures not significantly degrade the computational efficiency of TRAC-M with respect to TRAC-M, Version 2.0, which is already more efficient than TRAC-P, Version 5.4.25.

The Nuclear Regulatory Commission (NRC) also has asked that the new data structure permit the creation of a coarse-grained parallel version of the code. Past work at Los Alamos National Laboratory (LANL) [D. R. Liles, J. H. Mahaffy, and P. T. Giguere, "An Approach to Fluid Mechanics Calculations on Serial and Parallel Computer Architectures," in *Large Scale Scientific Computation*, p. 141 (Academic Press, Orlando, Florida, 1984)] and recent discussions between NRC staff, Lori Liebrock, and John Mahaffy (May 14-15, 1997) have determined that the current TRAC code and data structure support this implementation. Because neither TRAC's code architecture nor data structure design is altered by the data structure proposed here and, in particular, the same component modularity is maintained, the same parallel strategy used for the current version could be adapted easily for TRAC-M.

The modularity currently seen in TRAC (Fig. 1), including the dynamic, component-based (objected-oriented) data structure, was designed into the first version of the code. Unfortunately, Fortran 66 (F66) and 77 (F77) did not have features to implement this structure directly. As a result, all data were stored in a single container or "A" array (Fig. 2) and referenced at the highest levels through integer offsets ("pointers"). As an example of the awkward

---

\(^1\)Our current version control agreement for TRAC-M is that versions with version numbers using a single digit after the decimal point, e. g., 2.0 and 2.1, are deliverable, null-tested versions. However, we have the option of producing intermediate versions with version numbers using a different convention, such as two digits after the decimal point, e. g., 2.01, that are not null-tested versions. This option will be used where we feel it is more cost-effective to move forward in the development process while resolving non-null issues rather than delaying parallel efforts.
Fig. 1. TRAC-P data structure and architecture.
Fig. 2. Container array used in TRAC-P, Version 5.4.25.
nature of the data access with this implementation, consider obtaining information on the state of fluid in a given component. First, the location of the pointers must be obtained by copying the component's fixed-length table from the container array to the "fltab" common block. (This copying was necessary because there was no analog of an array of a derived type, which can be thought of as providing the functionality of an array of identical common blocks.) After obtaining their location, the integer pointers to the properties are copied from the container array to the "ptab" common block. Finally, after these two copies are made, the actual information is accessible from the container array with references such as "a(lpn)" for the beginning of the new time pressure information or "a(lpn+j-1)" for the value in the j-th cell of the component. This reliance on a container array to support the modular data structure fails to meet the requirements of maintainability, extensibility, and readability. It causes obvious difficulties in reading the higher-level coding and maintaining the chain of access to information as new arrays are added. This reliance has less obvious problems because of the considerable incidence of coding errors associated with the addition of integer pointers and the associated expansion of the pointer tables. These pointer-related errors often have been very difficult to trace and have required much debugging effort.

The new data structures, detailed in the following sections, will utilize the new features of F90. F90 modules will replace common blocks. All container arrays and integer offset pointers will be eliminated. All arrays will be individually and dynamically allocated, with their F90 pointers or arrays placed in modules. The fixed-length, variable-length, and pointer tables comprising the component data structures will be replaced with arrays of F90-derived types. The first two will contain the same data elements as before, whereas the pointer tables will contain F90 pointers.

These new data structures achieve a high level of maintainability, extensibility, and readability by completely eliminating the use of the current container arrays (Fig. 3). During initialization and time integration, the system-wide arrays and component data will continue to be used directly in all subroutines down through the property and solution drivers, one or two levels below the component routines (e.g., PIPE1, PIPE2, and PIPE3). For computational kernel routines below this level, information on the physical state of fluids and structures will continue to be obtained through subroutine argument lists (Fig. 4). During this restructuring, coding in the lowest-level subroutines will not change from its preconversion functionality. Argument lists will follow those of the existing version of TRAC-P 5.4.25, thus permitting the continued use of simple arrays for referencing state information in low-level subroutines (e.g., p(j), tl(j), tv(j) ).
Fig. 3. Database transformation from TRAC-P, Version 5.4.25 to TRAC-M.
Fig. 4. Computational flow and data passing.
II. DESIGN DESCRIPTION

II.1. Data Structures vs Data Layout

The basic goal of computational efficiency in the context of hierarchical memory is achieved by a data layout that minimizes data movement. However, laying out data in memory to minimize data movement is a highly technical issue that obviously varies from machine to machine.

The use of pointers makes it possible to hide the data layout from the data structures and will provide us with the flexibility to change the data layout in the future without affecting the data structures used by model developers; consequently, we will base the TRAC-M data structures on pointers. Further discussion of this issue is addressed in App. D. Special coding intended to control data layout, such as that illustrated in App. D, will not be utilized as part of this task; we simply will accept the data layout as generated by the compiler.

II.2. Dynamic Memory Allocation

There are two types of memory allocation: static and dynamic. Static allocation occurs at compile time and requires that array sizes be known before execution, whereas dynamic allocation occurs during execution. Because the required sizes of most TRAC arrays depend on the particular input model and there can be hundreds of different array sizes, dynamic allocation is most appropriate.²

In principle, static allocation could be achieved by one of two alternate tactics: (1) dimensioning all arrays with parameters chosen to represent the “worst” case or (2) use of an input preprocessor to read the input deck and determine parameter sizes. (This then would be followed by some complex editing process to customize the source code.) In the case of the current TRAC, neither of these tactics would be feasible; we propose an alternative at the end of this section. An input preprocessor developed for the existing TRAC code—which uses the container array approach to achieve the F77 analog of dynamic memory allocation—would require duplication of thousands of lines of TRAC’s input and initialization functionality to implement all of the logic associated with computing array sizes (although the sizes of some arrays may be obvious from the input model, the arrays sizes for internal data structures are calculated on a case-by-case basis). Either of the two tactics described here also would add considerable code maintenance and computational overhead with no apparent advantages. Moreover, no static

²The only situation where static allocation might be used would be for a simulator that utilized a single input model. The data structure design proposed here would not preclude such a future extension.
allocation scheme can address the need for additional time-dependent memory requirements arising from the particular course of a transient, such as the trip-controlled activation of reactor kinetics or dynamic rezoning.

F90 provides a very simple protocol for dynamic memory allocation and deallocation that supports all anticipated TRAC needs. This protocol uses the two built-in generic functions: allocate and deallocate. We will provide our own generic memory allocation functions, to be called TRACAllo and TRACDeallo, that utilize these two. This additional layer, which is detailed further in App. C, provides for a modular implementation of related functionality, such as addressing error conditions, providing initial values, and providing the output of array sizes.

This last feature—which will be implemented into TRAC-M—would make possible a straightforward means of implementing static memory allocation in customized versions of TRAC. The idea is that one would first use the standard TRAC-M code as its own input preprocessor by executing the code and using TRACAllo to generate a report giving the names and sizes of all of the dynamic arrays for the particular input model. This report then would be the basis for automated (script-based) source-code editing to create the custom version by replacing the dynamic allocation with fixed-array sizes, as illustrated in Fig. D-2. (Of course, this would be useful only for transients where static memory allocation was consistent with the numerical methods.)

Any computational overhead from using an additional layer is not a concern because essentially all TRAC memory allocation occurs only once during a computation. However, use of the additional layer does have one restriction: it will work only for allocating pointers and not for allocating arrays. Arrays, in contrast to pointers, can be dynamically allocated only in two cases: (1) if they are local to the routine doing the allocation or (2) if they are public elements of modules "used" by the routine. We will make direct use of allocate/deallocate for temporary or scratch arrays (compare with Example 2.2). Any need for direct allocation of permanent arrays, rather than pointers to arrays, will be addressed on a case-by-case basis.

---

3 One new feature of F90 is that programmers now can create user-defined generic functions by using the INTERFACE statement. This allows a set of different functions that are distinguished by having differing types and/or numbers of arguments to be referenced with the same function or subroutine name. (This is sometimes called operator overloading.) This is much more efficient than embedding conditional logic in the code because with generic functions, the compiler inserts the proper actual function at compile time (compare with App. C).

4 This report could be considered a virtual memory map by making the assumption that memory is sequentially allocated from the heap; however, this is actually compiler-dependent and outside the control of the programmer using standard constructs.
II.3. TRAC Data Structures

TRAC has two kinds of data structures: system-wide and component-based. The proposed reimplementations of these are discussed in turn.

II.3.1. System-Wide Data Structures

TRAC has many system-wide data structures that are used primarily to control the computational flow and to support models that are not implemented as components. Such models include the control system, constrained steady state, hydraulic-path steady-state initialization, and radiation model. In addition, constants defining the various fluid equations of state and tables used for structural material properties are system-wide data structures.

The system-wide data structures include scalars in common blocks and both real and integer arrays. We will make minimal changes to the former. We will convert the common blocks to data modules and integrate appropriate documentation from the "TRAC-PF1/MOD2 Code Manual Programmer's Guide."

Most of the system-wide arrays are one-dimensional (1D) (rank one); however, some are used as rank-two or rank-three arrays. These will be allocated and used consistently. There are also rank-one arrays, with supporting arrays that represent "ragged 2-D arrays"; these and other special system-wide data structures are discussed in Apps. A and B.

The container-array locations for the system-wide data structures currently are defined primarily in subroutines INPUT and INIT. The new coding generally will be a straightforward allocation of individual arrays, as shown in Example 2.1. However, there will be some nonobvious case-by-case changes because of stricter coding standards and provisions for extensibility. The pre-use of certain arrays as scratch space no longer will be permitted. Scratch arrays simply will be allocated and deallocated in the same routine, as shown in Example 2.2. Also, the re-use of the same memory location for more than one purpose no longer will be permitted.9

A second issue not illustrated by Example 2.1 is the data flow for these data structures. They currently are represented by offset pointers to various container arrays. Most of these pointers are found in the common block "PTRS" (see p. D-50 in the July 1992 release of the TRAC-PF1/MOD2 Programmer's Guide). As a first cut, the new allocable arrays corresponding to these pointers will be placed in a module of the same name.

---

9 This was done to save memory where the code developer was familiar enough with the data flow to be certain this was safe. However, readability is more important than saving memory. A good example is in TFIDS, where the thermodynamic variables TSN, ROVN, and ROLN also are used to store matrix elements in the network solution.
There is one subtle problem with the current data flow for the system-level arrays. Because these currently are stored in container arrays, they often are passed as subroutine arguments, e.g., IG(LORDER), so that the called routine can treat the array in the usual fashion. It may be that the called routine "includes" the PTRS common block. With the reimplementation, the compiler would generate an error because of the same name appearing as a dummy argument and in the module replacing the common block. We will resolve this by removing the variable name from the affected argument lists.

11.3.2. Component-Based Data Structures

TRAC has 10 separate object-oriented-like data structures for the 10 possible physical component types that can make up a reactor system. These components are the five 1D hydrodynamic components: the Pipe, Tee, Valve, Prizer (pressurizer), and Pump; the two pseudo-1D boundary condition components: the Break and the Fill; the Heat Structure; the "zero"-dimensional (0D) Plenum; and the three-dimensional (3D) Vessel.

The TRAC component data structure approach is analogous to the data structure aspects of the concept of a class in object-oriented languages. That is, the data structure for each component is essentially a template that is allocated dynamically to the appropriate size (instantiated) as each instance of that component is encountered in an input model. Thus, each actual component is represented as an object of its class. This is a very powerful approach that allows the input model to contain an arbitrary number of different types of differently sized components. Moreover, the component modularity aspects of the code architecture reflect the data structure in an intuitive manner that enhances code comprehension and communication between code users and code developers. This is primarily because the code and data organization are consistent with the user's concept of an input model. For example, an error message might report "out-of-bounds pressure in cell 2 of Valve component number 8." This would be meaningful to the user because it reflects the user's view of the model; it might even be sufficient information to uncover a modeling error. Moreover, the same information would be directly useful to the code developer because of the component modularity.

---

6There is actually a sixth 1D component, the Turbine. This component has a data structure slightly different from the other 1D ones and has not been tested or maintained for many years (compare with Task 5 in App. D).

7There is an alternate name for the Tee component: "sepd" for separator. This is not a different component but simply the (mis)use of the component type name to activate a special process model option within the Tee component (compare with Task 6 in App. D).
USE kindModule
USE TRACAllocateModule

!Component numbers in computational order
INTEGER (int_kind), POINTER, DIMENSION(:) :: iorder

!Loop-by-loop count of network junctions.
INTEGER (int_kind), POINTER, DIMENSION(:) :: njn

TYPE twodee
  REAL (real_kind), POINTER, DIMENSION(:, :) :: pt
END TYPE twodee

!Network matrices
!AOU is the pressure matrix, AOV is for vapor-associated
!stabilizer matrices, AOL for liquid matrices
TYPE (twodee), DIMENSION(:) :: aou, aov, aol

!SETS option for 1-D
LOGICAL, PARAMETER :: OneDTwoStep = .true.

INTEGER (int_kind) :: ncompp, nloops, il

CALL TRACAllo (iorder, ncompp)
CALL TRACAllo (njn, nloops)

! (Define njn array prior to use.)
DO il = 1, nloops
  CALL TRACAllo (aou(il)%pt, njn(il), njn(il))
  IF (OneDTwoStep) THEN
    aov(il)%pt => aou(il)%pt
    CALL TRACAllo (aol(il)%pt, njn(il), njn(il))
  ENDIF
ENDDO

Example 2.1. Allocation of system-wide arrays.

\(8\)In current versions of TRAC, the network matrices are solved sequentially, loop by loop. This allows the code developer to reduce memory requirements by allocating a single set of network matrices dimensioned for the largest case. This obviously prevents parallelization of this aspect of the computation. More significantly, the TRAC coding utilizing this array relies on the ability of F77 to allow the extent of the first dimension of a rank-two array to vary from subroutine call to subroutine call. This is nonstandard and should not be used in F90. What we show here is allocation of a separate set of network matrices for each loop.

\(9\)In the current version of TRAC, SETS vs semi-implicit is an option for the 3D but not the 1D or plenum features. We have put this option in Example 3 to indicate the simple case-by-case provisions for possible future extensions that we will incorporate as we modify the code.
The component data structure is hierarchical and consists of two aspects (the Fixed-Length Table and the Variable-Length Table) that are common in the abstract sense to the different component types, and a third aspect, the Pointer Table(s) and associated arrays, that varies among the types (Fig. 3). These are discussed in order.

11.3.2.1. Fixed-Length Table. There will be a dynamically allocated array, called FLTABS, of an F90-derived type, called FLTAB, for each of the NCOMP components in an input model. The original TRAC name, "fixed length table," refers to the fact that the data elements are the same (fixed) for each type of component. This is the scalar data that are common to all components. The fixed-length table is the way that the components are accessed and become part of the computational flow. There will be a system-wide data structure array, COMPTR(1:NCOMP), paralleling the array IORDER of Example 2.1. The COMPTR array will contain the index into the FLTABS array for each component in an input model.

Example 2.1 shows how the FLTABS will be defined and used, as illustrated with a subset of the actual data. The elements of the FLTAB-derived type will comprise the variables in the current FLTAB common block plus additional variables, such as IVLTAB described in Example 2.3, needed to

---

10The various pieces of data in an F90-derived type are more correctly called the components of the type rather than elements. The latter term is used here to avoid confusion with the TRAC concept of the component.
support the reimplementation (see pp. D-31 and D-32 in the previously cited MOD2 Programmer's Guide). This data structure will comprise its own F90 module.

This data structure requires no change to accommodate new types of components. This is one of the few data structures that is appropriate to use at all levels of the computational flow.

II.3.2.2. Variable-Length Table. There will be 10 dynamically allocated arrays, called PIPEVLTAB, VESSELVLTAB, etc., of F90-derived types, called PIPEVLTAB, VESSELVLTAB, etc., respectively, for each of the 10 possible

```
MODULE fltabModule
  TYPE fltab
    INTEGER :: num        !User component number.
    CHARACTER*8 :: type   !Component type
    INTEGER :: ivltab     !Index into type-specific variable-length table.
  END TYPE fltab
  TYPE (fltab), ALLOCTABLE, DIMENSION(:) :: fltabs
END MODULE fltabModule

USE fltabModule

CALL TRACAllo (fltabs, ncompp)
npipes = 0
nvalves = 0

DO i = 1, ncompp
  READ (*, '(A8,i5)') fltabs(i)%type, fltabs(i)%num
  SELECT CASE (type)
    CASE ('pipe')
      npipes = npipes + 1
      fltabs(i)%ivltab = npipes
      nld = nld + 1
      CALL rpipe (fltabs(i)%ivltab, nld)
    CASE ('valve')
      nvalves = nvalves + 1
      fltabs(i)%ivltab = nvalves
      nld = nld + 1
      CALL rvalve (fltabs(i)%ivltab, nld)
  END SELECT
END DO
```

Example 2.3. Fixed-length table.

---

11 Of course, we will remove unused variables and those such as common block start and end marker variables made obsolete by the new data structures.
types of components in an input model. The original TRAC name, "variable length table," refers to the fact that the data elements are different for each type of component. This data structure stores the scalar data that are common to all components of a given type.

The variable-length table is also the way that the components of a given type are accessed and become part of the computational flow. As shown in Example 2.3, the fixed-length table for a component contains the index into the corresponding variable-length table for that component.

The data elements of the various "VLTAB" derived types will comprise the variables in the corresponding current VLTAB common blocks, with the same caveat as for the FLTAB data (compare with App. C in the previously cited MOD2 Programmer's Guide). Each of the 10 VLTABS arrays will be part of an F90 module that is accessible only in component-specific subroutines such as PIPE1 or HTSTR3.

Extension of this data structure to accommodate new types of components is straightforward. An appropriate VLTAB-derived type must be defined, an array of that type declared, and provision for its allocation made in parallel to that of existing the VLTABS.

II.3.2.3. Pointer Tables and Arrays. As we have seen, each of the 10 component types have an identical fixed-length table data structure and analogous variable-length table data structures. However, there is significant variation among the data structures for the additional data required for each object of each component type. It is easiest to understand this in terms of four basic categories: (1) 1D and pseudo-1D component types, (2) Heat Structure, (3) Plenum, and (4) Vessel. These are described in turn.

II.3.2.3.1. 1D data structures. The 1D data structures comprise a hierarchical data structure with all of the 1D components, including the basic Pipe data structure. This commonality supports functional modularity by allowing all of the 1D hydrodynamic component types to utilize the same computational kernels as the subroutines FEMOM and TFIDS. The basic Pipe data structure is described in the next section. This is followed by additional details for the non-Pipe 1D component types.

II.3.2.3.1.1. The Basic Pipe Pointer Tables and Arrays. Each component requires several arrays associated with the hydrodynamic finite-difference mesh to characterize the time-dependent state of the component. TRAC uses a staggered grid with some quantities located at cell centers and others at cell faces; the corresponding arrays are dimensioned to the number of cells and

\[12\] The size of these arrays will be determined by a prepass through the current TRAC input deck TRACIN.
the number of cell faces for that component, respectively. The two basic categories of hydrodynamic mesh variables are dual and single time. The dual-time variables are those for which it is necessary to maintain both the start-of-cycle and the working or end-of-cycle values to support a backup capability and/or to compute time derivatives. The remaining mesh variables are single time (these variables are in the dualpt, hydropt, and intpt commons in the MOD2 Programmer's Guide).

An implementation of the basic Pipe data structure is shown in Example 2.4. The array comp1d will be dimensioned dynamically to the total number of 1D components at the start of the computation. The individual arrays for each component then need to be allocated before the input for that component so that the input can be read directly into these data structures. At the higher levels of the computational flow, the hydro1DModule will be used in the subroutines so that data elements are accessed using derived-type notation, e.g., comp1d(i)%vol or comp1d(i)%new%vol. The derived type notation will be used in the arguments to the computational kernels at the lowest level of the computational flow so that these routines are not cluttered with data structure details (Fig. 4).

The 1D hydrodynamic components include a simple heat structure option that is distinct from the Heat Structure component. This option provides for

```
MODULE dualTimeModule
    TYPE dual
        ! Cell-center arrays
        REAL, POINTER, DIMENSION(:) :: p, alp, tv, tl, ...
        integer, pointer, dimension :: bitc, ...
        ! Cell-face arrays
        REAL, POINTER, DIMENSION(:) :: vl, vv, ...
        INTEGER, POINTER, DIMENSION :: bit, bitf, ...
    END TYPE dual
END MODULE dualTimeModule

MODULE ptablDModule
    USE dualTimeModule
    TYPE hydrolD
        TYPE (dual) :: old, new
        ! Single-time arrays
        REAL, POINTER, DIMENSION(:) :: vol, rol, rov, ...
        INTEGER, POINTER, DIMENSION(:) :: nff, ...
    END TYPE hydrolD
END MODULE ptablDModule

MODULE hydro1DModule
    USE ptablDModule
    TYPE (hydrolD), DIMENSION(:) :: comp1d
END MODULE hydro1DModule
```

Example 2.4. Basic Pipe data structure.
1D radial conduction and is intended primarily to model the thermal mass of pipe walls. Users have requested that we retain this option because it is much more convenient than using a Heat Structure component where its simple functionality is adequate; therefore, we have no long-term plans for its removal. In any case, it needs to be retained now to keep TRAC-M fully functional.

The special arrays that support the built-in heat structures are primarily in the "heatpt" section of the "ptab" common block. However, some of these arrays are in the "dualpt" section. Both sets of arrays will be included within the basic Pipe data structure shown in Example 2.4. These arrays will be allocated only for components in the input model that actually use the option.

II.3.2.3.1.2. Arrays supporting special 1D component models. Arrays in addition to those basic mesh variables described in the previous section are required to support the various special flow process models for each of the seven 1D and pseudo-1D components. These arrays are currently in the A array, with their integer offset pointers in the seven specially named sections of the Pointer Table, e.g., the Break Pointer Table or Valve Pointer Table as defined in App. C of the previously cited MOD2 Programmer's Guide. They include such data as tables supporting time-dependent boundary conditions or valve-flow-area adjustment. Each of these seven data blocks will be replaced with corresponding modules containing dynamically allocated pointers to arrays. The array names will be unchanged. The data flow for these modules will be the same as for the corresponding Variable-Length Tables.

An implementation of special flow-process data for the Valve component is shown in Example 2.5. The array valve will be dimensioned dynamically to the total number of Valve components. The individual arrays for each Valve component will be allocated dynamically based on the corresponding Valve scalar input.

```fortran
MODULE valvePtabModule
  TYPE valvePtabT
    REAL, POINTER, DIMENSION(:) :: vtbl, vtb2, ...
  END TYPE valvePtabT
END MODULE valvePtabModule

MODULE valveModule
  USE valvePtabModule
  TYPE (valvePtabT), DIMENSION(:) :: valve
END MODULE valveModule
```

Example 2.5. Special Valve-model array data structure.
II.3.2.3.2. Vessel data structures. There are two data structures for Vessel array data. Most of the data are associated with the fluid mesh; this is discussed in the following section. The additional data structure that supports other arrays is discussed in Section II.3.2.3.2.2.

II.3.2.3.2.1. Data Structure for Vessel 3D fluid mesh arrays. To understand our choice for the Vessel 3D mesh data structure, it is useful to understand the history of this data structure.

Early versions of TRAC had the 3D data structure organized in terms of the two-dimensional (2D) axial slices. This was difficult to read but was necessary because the entire Vessel database could not fit into the computer memory of the CDC 7600. When the Vessel structure was completely rewritten for MOD2 in 1985, it utilized standard 3D arrays. However, to employ the A-array mechanism for dynamic memory management while avoiding the complex use of offset pointers and long argument lists, the mesh data were “inverted” (made variable-wise) and made directly accessible at all levels of the computational flow. This required that the first two dimensions of the 3D mesh arrays be defined by parameters to the maximum size necessary. This results in wasted space for multiple Vessel models where either of the first two dimensions are different for the different Vessels. (There is no wasted space for single Vessel models if the parameters are set properly.) For example, in the case of the large AP600 model, as shown in Fig. 5, as run with the current TRAC-P Version 5.4 code series, there is ~60% wasted memory, roughly 2000 words per array, or 600,000 words for the ~300 Vessel mesh arrays. This can be seen by a comparison of Figs. 6 and 7.

<table>
<thead>
<tr>
<th>NRSX</th>
<th>NTSX</th>
<th>NASX</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

Fig. 5. AP600 Vessel input specifications.\(^{13}\)

<table>
<thead>
<tr>
<th>IDIM</th>
<th>JDIM</th>
<th>KDIM</th>
<th>SIZE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>11</td>
<td>20</td>
<td>1540</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>17</td>
<td>1275</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>11</td>
<td>308</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td></td>
<td>3123</td>
</tr>
</tbody>
</table>

Fig. 6. AP600 Vessel 3D array minimum size, including ghost cells.

\(^{13}\) We thank Jim Lime for supplying us with this figure.
In the current TRAC-M, Version 2.0 code, the Vessel 3D data have been uninverted (made mesh-wise) and also have been removed from the A array. To minimize change during this transition period, these data were placed in a new common block, vssArCom, with each of the mesh arrays hard-dimensioned with parameters for the worst case. Again, this results in wasted space for multiple Vessel models, where either of the last two dimensions is different for the different Vessels. (However, as before, there is no wasted space for single Vessel models if the parameters are set properly.) If these parameters are chosen for the large AP600 model, there is ~85% wasted memory, roughly 2500 words per array, or 750,000 words for the ~300 Vessel mesh arrays. This can be seen by comparison of Figs. 6 and 8. This is not significantly different from the wasted space associated with the original implementation, which has been in use for >10 yr.

Based on user experience here at LANL, the main drawback to both TRAC-P, Version 5.4 and TRAC-M, Version 2.0 implementations is not the wasted space but the need to recompile because of the parameterization of the array extents. However, both of these implementations have software engineering advantages in that they avoid the need for long argument lists and overhead for managing multiple data structures for multiple Vessels.

Consequently, we will reimplement the 3D mesh arrays by making them allocable arrays in a module, as shown in Example 2.6. Here there will be no wasted memory for the typical single Vessel model, but rather some model-dependent memory overhead for multiple Vessel problems. The advantage of this approach is that because the hard work of uninverting the Vessel mesh data has already been done, reimplementation with the same design requires simply replacing the current references to the vssArCom common block with references to a module and adding the coding to allocate the arrays dynamically.
MODULE vssArCom
    REAL, ALLOCATABLE, DIMENSION(:,:,:): p, alp, vol
END MODULE vssArCom

SUBROUTINE Allo3DMesh
    USE vssArCom
    ...
    lasti = 0
    jmax = 0
    kmax = 0
    ! Loop through all Vessels in input model.
    DO n = 1, nmvssl
        lasti = lasti + idim(n)
        jmax = max (jmax, jdim(n))
        kmax = max (kmax, kdim(n))
    ENDDO
    CALL TRACAllo (p, lasti, jmax, kmax)
    CALL TRACAllo (alp, lasti, jmax, kmax)
    CALL TRACAllo (vol, lasti, jmax, kmax)

Example 2.6. Proposed dynamic allocation of current 3D mesh arrays.

II.3.2.3.2.2. Data Structure for Vessel Special Arrays. In addition to the basic 3D mesh arrays, the Vessel data structure contains 1D and 2D arrays for various purposes, such as information about the source connections and geometry. These arrays currently are stored in the A array with their integer offset pointers in the Vessel PTAB (Vessel Pointer Table) common block (compare with Section C.14.2 in the MOD2 Programmer's Guide). Some of these pointers were associated with the generation of TRCGRF and no longer are used anywhere in the code; these will be deleted. The remaining arrays will be replaced with dynamically allocated F90 array pointers in a ptab3DModule in exactly the same manner as the 1D arrays are handled in Example 2.4.

II.3.2.3.3. The basic heat structure arrays. The Heat Structure component introduces a significant increase in data structure complexity beyond the 1D hydrodynamic component. Divisions into new time, old time, cell center, and cell edge are present. However, there is also a distinction between coarse- and fine-mesh variables and the need to separate and allocate space as needed for conditions at the inner and/or outer surfaces of the structure. In addition, a higher level of data splitting exists between information generic to the structure group and array information specific to the optional copies of that structure. Arrays in the current data structure containing one element of information for each copy of the heat structure (e.g., pgapt and plvol) are preserved as arrays in the generic data rather than distributed as scalars in the copy-derived data type (these variables are in the ptabr, htcref1, htcref2, and htcref3 commons described in the previously cited MOD2 Programmer's Guide).
An implementation of the basic Heat Structure data structure is shown in Example 2.7. Selected variables from the full Heat Structure database are included to indicate the breakdown by category. Space for individual pointer variables is provided.

```fortran
MODULE heatStructureModule

  TYPE structureTimeDependent
    ! Coarse mesh
    REAL, POINTER :: drz(:), radr(:,:)
    ! Fine mesh
    REAL, POINTER :: rft(:,:)
  END TYPE StructureTimeDependent

  TYPE structureSurface
    ! Coarse mesh
    ! Hydro cell centered
    REAL, POINTER, DIMENSION(:) :: alpr, alvr, clr, concr
    INTEGER, POINTER, DIMENSION(:) :: hydrocomp, hydrocell
    ! Hydro cell face
    REAL, POINTER, DIMENSION(:) :: hlar, hvar
    ! Fine mesh
    REAL, POINTER :: tchff, hrfg, hrfl, hrfv
  END TYPE structureSurface

  TYPE heatStrucCopy
    ! Data Unique to each copy of the heat structure
    INTEGER ncrz
    TYPE (structureTimeDependent) new, old
    TYPE (structureSurface) inner, outer
    ! Coarse mesh single time or time independent
    REAL, POINTER, DIMENSION(:) :: hgap, pgap, pint,&
      pldv, qwrx
    REAL, POINTER, DIMENSION(:,:) :: cnd, cpnd
    ! Fine Mesh single time or time independent
    INTEGER, POINTER, DIMENSION(:) :: IDHT
    REAL, POINTER, DIMENSION (:) :: zht
  END TYPE HeatStrucCopy

  TYPE heatStruc
    ! Data Specific to copies of the structure
    TYPE (heatStrucCopy), POINTER, DIMENSION(:) :: copy
    ! Data used by all copies
    REAL, POINTER, DIMENSION(:) :: gmix, xgmles, pgapt
    INTEGER, POINTER, DIMENSION(:) :: matrd
  END TYPE heatStruc

END Module heatStructureModule
```

Example 2.7. Basic Heat Structure data structure.
arrays needs to be allocated before the input of the heat structure to permit direct input of values to these data structures. Space for the inner and outer surfaces will be allocated as needed, eliminating the current hidden dual surface structure, which is generated by shifting integer pointers in the container array. Cell-edge and cell-center quantities are segregated within the same derived type for clarity, but no need was identified to segregate them into their own derived types.

II.3.2.3.4. Plenum data structures. The Plenum component differs from the other thermal-hydraulic components in TRAC by having only a single cell (for the solution of mass and energy conservation equations), with an unlimited number of interfaces to other thermal-hydraulic components—implemented with TRAC “Junctions”—for the solution of momentum equations and specification of flow into and out of the cell. The implementation of the Plenum data structure in the base code is probably the most confusing and unnecessarily obfuscated in TRAC, involving the declaration of 110 single-element arrays for cell variables, the EQUIVALENCING of these arrays to offsets from the beginning of the container array (see plendata.h), and the aliasing of these arrays with pointers used in the 1D components (see ptrsp1.f) for access by other parts of the code.

As a first step in reimplemention, the “inverted database” was removed in Version 1.10 and replaced by using integer pointers consistent with the 1D components. In F90, there will be an array of dynamically allocated Plenum derived types, each of which contains scalars for the cell variables and dynamic arrays for the junction variables. Plenum worker routines may be placed together in one or more modules to simplify access to the database.
III. DESIGN SELECTION

The design selection has two aspects. The first aspect, discussed in the following section, resulted from considerations in achieving a reasonable staging for TRAC modernization. The second aspect, discussed in Section III.2 and the following subsections, arises from the technical requirements for the various data structures in TRAC.

III.1. General Considerations

The first decision in the design selection was to use a reengineering approach; reengineering TRAC is a cost-effective method of providing a modern thermal-hydraulic systems code that can have multiple uses. One anticipated use is testing new models and methods in a realistic setting that will provide a basis for an understanding of feedback between models.

The second decision that affected the design selection was the need to stage the modernization effort so that there would be a useable product following each stage. The first stage obviously had to involve the data structure because all other possibilities for modernization require data structure cleanup as a precedent activity. For example, reimplementing the data structures is the only way we can provide accurate, detailed, and understandable documentation. This is because the TRAC data structures contain numerous artifacts that are the inevitable result of hardware, pre-F90 Fortran limitations, and tens of code developers working over the 22-yr history of the source code.

The next decision was that the scope of the current (first stage) project simply be the reimplementation in F90 of the current TRAC data structures, not significant redesign.\[14\] [Whether there will be a need for redesign to support major extensions or computational efficiency is not yet clear (compare with App. D, Task 8)]. This decision to reimplement was made for four reasons: (1) it is the most straightforward approach, (2) the robustness of the TRAC data structure supports maintainability and some level of extensibility, (3) the conceptual design remains unchanged, and (4) the design embodies many of the advantages that have made object-oriented programming (OOP) so popular.

The first reason, simplicity, requires no further justification. The second reason, robustness, is supported by the fact the basic data structures have remained useful for the entire life of TRAC. The third reason, leaving the design unchanged, is important because we know that the design works and that we will not discover major problems partway through the development

\[14\] Business decisions outside the scope of this document led to substaging because the initial customer wanted to treat the 3D reimplementation as a separate task.
process. The final reason, the relationship to OOP, deserves some additional detail. This discussion is based on a recent book on C++.\(^\text{15}\) One reason that OOP is so popular is that this approach allows physical objects to be used as the basis for data structures; this was discussed above in Section II.3.2. A second advantage to OOP is that it supports modularity. A third advantage is that it provides a maintainable and extensible code structure. Thus, reimplementation of TRAC's data structure will lead to one consistent with the latest thinking in software engineering.

III.2. Data Structure Hierarchy

We have identified two general data categories within TRAC: system-wide and component. The principal system-wide structures are associated with the control system, the constrained steady state, the hydraulic-path steady-state initialization, the radiation model, user-defined structural material properties, the Network solution procedure, and general bookkeeping for computational flow. The component data cover the seven 1D hydrodynamic components [Pipe, Tee, Valve, Prizer (pressurizer), Pump, Fill, and Break], the 0D Plenum, the 3D Vessel, and the Heat Structure component. Within the context of object-oriented programming, each of these can be considered an object class.

Clear data groups exist within the component data structure. The first group is data that must exist for every component, currently placed in the fixed-length table (see Section II.3.2.1). The second is information (mostly scalar) that is specific to a given component and is placed in the variable-length table (see Section II.3.2.2). The third is array information for the particular flow and/or heat-transfer model associated with the component. In this third category, the seven 1D components contain copies of one such structure, the 0D and 3D components have related but distinct structures, and the heat structure has its own distinctive set of arrays (see Section II.3.2.3).

As stated previously, we staged TRAC modernization to provide a useable product after each stage. Providing a completely accurate detailed description of design decisions for the old TRAC data structures and their implementation is beyond the scope of this effort. We will provide documentation on specific decisions of F90 constructs used to modernize the underlying data structure. Also, we will clearly document decisions that modify or extend the nature of the old structure.

Although the general form of the TRAC data structure, especially its object-oriented aspects, was found to be maintainable, extensible, and readable, certain details of the structure do not meet these criteria, and some alterations

of details were both permissible and desirable within the scope of work. These changes to the details are described in Sections II and IV; the reasoning behind their selection is provided in the following subsections.

### III.2.1. System-Wide Data Structure

This material consists largely of simple arrays. The three practical options available are to (1) leave this information in a smaller container array, (2) create simple allocated arrays for each, or (3) create pointer arrays for each. The first option has the attraction of creating the minimal amount of work but fails the tests of being maintainable, extensible, and readable. The last two options meet these tests. Option 2 was selected because of relative simplicity, tests indicating a slight speed advantage, and flexibility to later convert to pointers if advantages develop for that option.

Section II.3.1 illustrates one important exception to the “simple array” structure within the system-wide data. The existing method of storing coefficient arrays for the Network solution method is very complex, with information overwritten as each “loop” in the system is processed. Continuing to follow this pattern preserves one of the least readable or maintainable sections of the code. The solution illustrated was obtained while looking for simple continuity in existing naming conventions and permitting simultaneous storage of all loop information. The need for different array sizes in each loop drove a selection of a derived-type array, with an array pointer as the component of each array element. A separate element of the array for each loop makes future coarse-grain parallel processing efforts much simpler.

### III.2.2. Component Data Structure

#### III.2.2.1. The Fixed-Length Table

The selection of this data structure is driven primarily by the requirements of no change in the general TRAC structure and of minimal change to basic variable names. The old TRAC process was to keep components’ copies of the fixed-length table in the container array and copy them to the FLTAB common block when needed. This was in essence an old Fortran trick to implement a feature now available with a derived-type array. The example in Section II.3.2.1 is an example of this direct translation. It improves the code by marking the location of all such information clearly and eliminating the need to worry about location in the container array. It also eliminates the need for separate tracking of the length of the fixed-length table, thus eliminating one possible point of error as information is added or subtracted from the table.

#### III.2.2.2. The Variable-Length Table

Logic for the selection of this group of derived types (one for each of the 10 existing components) followed that for the fixed-length table. The elimination of separate tracking of length of the variable-length table is even more important than for the fixed-length table.
Historically, TRAC has had more errors resulting from changing the contents of this data structure.

III.2.2.3. Component Array Data. The vast majority of TRAC’s computational effort goes into computations involving data from this portion of the data structure. Any selection here must add the consideration of not degrading the performance of the code.

Preservation of the general code structure requires consideration that these data are clustered by component. The two methods identified to accomplish this goal were to follow a derived-type strategy analogous to that developed for the fixed- and variable-length tables or to mark subsections of larger arrays containing all values of a given property within the system. Section IV demonstrates that under some constraints, neither of these options has a significant time advantage. The derived type was selected based on consistency with the existing TRAC component grouping. It is more readable, clearly calling out the component relationship within the structure of the derived-type variable name. It also has a high degree of flexibility, producing distinctive naming patterns that are very amenable to the use of global meta-text replacements for later transformation into other structures.

Once a decision was made on a derived-type structure, the need for arbitrary array lengths required that array components of the derived type be array pointers. Beyond this, several options were available for the exact arrangement of these array pointers. One question was whether to follow the existing TRAC practice and simply list the arrays as follows:

```fortran
  type hydro_1D
    real, pointer, dimension(:) :: p, pn, tl, tln, vol, dx
    integer, pointer, dimension(:) :: nff
  end type hydro_1D
```

or to group variables by cell in an additional derived type of the form:

```fortran
  type cellType
    ! Cell-center Variables
    real(db1) p, pn, tln tln, vol, dx
    integer nff
  end type cellType
```

and use it as an array for the component:

```fortran
  type hydro_1D
    real(cellType), pointer, dimension(:) :: cell
  end type hydro_1D
```
The latter cell-based data structure shows timing disadvantages for simple calculations, but none for the more complex calculations typical of TRAC. Both forms contain the desirable features of readability and flexibility for meta-text replacement. The cell-based form has the advantage of permitting very clear, low-level subroutines based on scalars and provides a convenient clustering of data that could prove useful in future parallel adaptations. The most significant low-level TRAC subroutines assume an array structure for the incoming information. Rewriting these routines is not possible within the scope of the current effort. This drives the selection of the first of the two above options (an array for each component). A future shift to the second option would not be precluded because the vast majority of that transition would be in the necessary rewrites of the low-level subroutines (see the transition process between test cases th4a.f and th4b.f in Section V).

The actual data structure represents a variation on the simple array option just discussed. To make the code more maintainable and extensible, a special subgroup has been created for variables requiring storage for more than one time level (see Example 2.4 of Section II.3.2.3.1.1). This was found to have no speed penalty. In addition, the old version of TRAC stores thermodynamic derivatives (and saturation enthalpies) in a single array (dr). This has always been a very nonreadable portion of the code, and improvement has been highly desirable. Options were simply to convert it to a derived data type with clear names for components or to place derivatives in individual arrays. Both produced the desired readability, but the derived type was selected based upon ease of conversion of the existing code and speed advantages seen in test problems (see Section V).
IV. DESIGN IMPLEMENTATION

We decided to approach reengineering in a phased way so that the code was always fully functional and debugging could be accomplished by comparing results with the baseline. We felt that this would be an effective approach, and our experience to date in developing TRAC-M, Version 2 has confirmed this. This approach allows in-place prototyping of various implementation options, with a quick diagnosis of their effects on the whole code, always with the constraint of NULL results. Many implementation questions can be resolved only in this manner. An example of this was the early decision to extract the control system into separate static arrays for each data type, rather than storing all variables as REAL*8.

Moreover, we successfully tested this approach in October 1996 on the Cray using two prototypes. The first prototype was simply a subset of existing TRAC F77 routines tied together with driver routines that mimic the data structures and data flow in TRAC-P, Version 5.4.25. The second prototype was hand-modified from the first to utilize F90 data structures similar to those described here. This experience of walking through the modification process convinced us that it was straightforward and obvious. We also performed timing tests and determined that the F90 version was marginally faster.

Currently, TRAC’s dynamic database is stored in a single-container array (named “A”) in unnamed (blank) COMMON. Information is referenced in two fundamentally different ways: directly, using a “global” reference via an integer index that may have been calculated nonlocally, and indirectly, using a “local” reference to a dummy argument where the address has been passed to a “worker” routine. Although the conceptual design of TRAC’s data flow mandates a clean interface between these two types of reference, the practical situation is more complex, i.e., many procedures contain both kinds. Most of the work in reimplemention will be replacing global-level container array references with references to an instance of an F90-derived type and replacing worker-level container array references with references to new local procedure arguments. The number of routines that do not use global information will be reduced, thereby increasing modularity, at the expense of passing more information via procedure argument lists. Most information still will be passed as arrays, but the interfaces will be strongly typed (the current F77 interfaces are untyped), either implicitly by MODULE USE association or explicitly by INTERFACE blocks. Some changes in the procedure-calling tree may be necessary to facilitate this cleanup of the hierarchical data structure.

The mechanics of reimplemention are fundamentally manual, with manual-assist Perl scripts used where automated context-sensitive pattern-replacement and reformatting is possible. Manual-only operations are
sometimes necessary to make the source consistent enough for automated operations, and manual changes are often necessary to pick up changes that were not included in the automatic search context. In many cases, Perl scripts can be (1) used to extract information (usually variable lists) out of the old source, (2) reformatted, and (3) written out using F90 syntax and semantics into code fragments. These code fragments then are pasted manually into the source. However, because of the high entropy of the old source, the framework for all operations must remain manual, i.e., subject to line-by-line inspection and change by the developer.

The remainder of this section parallels discussions in Section II, which covers the system-wide and component-based data structures. More detail is provided, including full listings of the modules containing the particular data structures, lists of subprograms requiring that portion of the data structure, and tree charts indicating the position of these subprograms in the calling structure of the program. Replacement of the old container array structure with the new allocated arrays and array pointers required some changes in the previous flow of information. Where appropriate, we have provided lists of subprograms that require argument list changes to accommodate the new data structure.

(Note: Section headers below presently are placeholders for future details. Some details have been provided as examples of future content).

IV.1. System-Wide Data Structures

System-wide data structures are used in routines working with the global system structure and those establishing links between individual components. Table IV.1 provides a list of all subprograms using one or more of the allocated arrays in this module. The location of these subprograms in the calling structure of the program is illustrated in figures [tree charts not yet created].

<table>
<thead>
<tr>
<th>TABLE IV.1</th>
<th>SUBPROGRAMS USING THE SYSTEM-WIDE DATA STRUCTURE.</th>
</tr>
</thead>
<tbody>
<tr>
<td>bkmom</td>
<td>evltab</td>
</tr>
<tr>
<td>break1</td>
<td>fbcss</td>
</tr>
<tr>
<td>cbset</td>
<td>fillx</td>
</tr>
<tr>
<td>cihstst</td>
<td>findnum</td>
</tr>
<tr>
<td>civssl</td>
<td>fitom</td>
</tr>
<tr>
<td>constb</td>
<td>hout</td>
</tr>
<tr>
<td>core1</td>
<td>hstr1</td>
</tr>
<tr>
<td>core3</td>
<td>hstr3</td>
</tr>
<tr>
<td>dhtstr</td>
<td>hstrv</td>
</tr>
<tr>
<td>dmpit</td>
<td>icomp</td>
</tr>
<tr>
<td>edit</td>
<td>ihpss1</td>
</tr>
</tbody>
</table>
In older versions of TRAC, many of these subprograms used some of the contents of the system-wide data structure, both directly from the container array and indirectly via their argument lists. Others have name conflicts with the new data structure because they moved container information into a local variable with a name identical to that used for an allocated array in the new structure. These subprograms are listed in Table IV.2.

[Details of required changes]

IV.2. Component-Based Data Structures

IV.2.1. Fixed-Length Table

IV.2.2. Variable-Length Table

IV.2.3. Pointer Tables and Arrays. As mentioned in Section II, this portion of the data structure currently is divided into four categories: (1) the 1D and pseudo-1D component types, (2) the Heat Structure, (3) the Plenum, and (4) the Vessel. These are described in turn in the following subsections.

IV.2.3.1. 1D Data Structures.

IV.2.3.1.1. The basic Pipe pointer tables and arrays.

IV.2.3.1.2. Arrays supporting special 1D component models.

IV.2.3.2. Vessel Data Structures.

IV.2.3.3. The Basic Heat Structure Arrays.

IV.2.3.4. Plenum Data Structures.

<table>
<thead>
<tr>
<th>TABLE IV.2</th>
<th>SUBPROGRAMS REQUIRING DATA CHANGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>break1</td>
<td>dhtstr</td>
</tr>
<tr>
<td>clihtst</td>
<td>illx</td>
</tr>
<tr>
<td>civssl</td>
<td>htstr1</td>
</tr>
<tr>
<td>constb</td>
<td>htstr3</td>
</tr>
</tbody>
</table>
V. DESIGN TESTING

Several prototyping activities have been performed at LANL, the NRC, and Pennsylvania State University (PSU) as part of the data structure design and selection. The candidate data structures fall into three general categories. The first simply involves single allocated arrays for each state variable (pressure, temperature, velocity, etc.) and the use of stored bounding indices to extract values in the array specific to a given component. The second employs a component-based array of a derived type that contains array pointers for each state variable in that component. The third is a variation on the second, in which the component-derived type contains a cell array also of a derived type. The cell-derived type contains a scalar variable for each state variable within that cell. Examples of all of these structures are provided in Section V.2.

The first of these three general categories was dropped from consideration because it did not exhibit an overall speed advantage and had the disadvantage of hiding the code's modular structure. The third also was dropped from immediate consideration because of incompatibility with existing low-level subroutines, which operate on arrays of state variables. It showed some speed disadvantages on simple subroutines but was as good as or better than the others for problems performing the complex calculations typical of two-phase thermal-hydraulics. In the process of benchmarking these options, we demonstrated that conversion from the second- to third-type data structure was very simple. Although the second option has been selected for the current stage of modernization, the third remains viable for later stages.

Two general methods of passing data to subroutines were explored for all of the above three structures. The first was to pass the lowest levels of the data structures through the argument list (state variable arrays for data structures one and two and scalar state variables for data structure three). The second was to provide the highest level of the structure through its module and to provide information on the current component via the argument list. This second approach had only a significant speed advantage on a Cray. Except when used with the first data structure, it is less readable than the first passing option and violates the general principles of data hiding and modularity. We elected to continue TRAC's use of the first option for passing data to low-level subroutines.
Studies were performed at three levels. Section V.1 discusses simple timings performed on F90 features relevant to the construction of data structures. Section V.2 covers the application of the three data structure classes and two passing methods to subroutines performing very simple operations. Section V.3 presents results of applying these data structures to an actual subroutine from TRAC. Results relevant to the selection of the data structure are reviewed in Section V.4.

V.1. General F90 Tests

A series of simple benchmark programs was created to look for potential pitfalls in the implementation of an F90-based data structure. The last of these were timarr2.f, exploring options for treatment of rank 1 arrays, and timarr3.f, which contains a subset of the timarr2.f tests applied to rank-3 arrays. Tests were based on a simple product of array elements in the form:

\[
\text{do } i=1, \text{ncells} \\
\quad c(i) = a(i) * b(i) \\
\text{end do}
\]

Because this is a single-vector operation, it actually can be implemented as the F90 vector construct \( c = a * b \) with no change in performance. All arrays contain 3000 elements.

The rank-1 benchmark showed only minor variations for using fixed arrays, allocated arrays, or allocated pointers to arrays passed through either arguments or modules. However, a significant disadvantage was seen on both IBM RS6000 and DEC Alpha in the direct use of arrays of a derived type or pointers to arrays used directly within a derived type.

For the following use of a derived-type array and operation:

\[
\begin{align*}
\text{type nine-scalars} \\
\quad \text{real :: } x1, x2, x3, x4, x5, x6, x7, x8, x9 \\
\quad \text{end type nine-scalars} \\
\text{type(nine_scalars) :: } xa9(nc), xb9(nc), xc9(nc) \\
\quad xc9\%x2=xa9\%x2*xb9\%x2
\end{align*}
\]

the speed was dropped by a factor of 5 on an IBM RS6000 and by a factor of 20 on a DEC Alpha by using a standard array. The major contributing factors were probably the effective stride introduced by the structure and questions of cache performance. This degradation would be strongly mitigated in a DO loop with other operations to keep the arithmetic unit busy.
Direct use of a pointer out of a derived type is illustrated next:

```fortran
type misc
    integer i1,i2,i3,i4,i5,i6
    real r1,r2,r3,r4,r5
    real a(nc),b(nc),c(nc)
    real, pointer :: ap(:),bp(:), cp(:)
end type misc

type (misc) :: xm
    xm%cp= xm%ap*xm%bp
```

This loop ran at 50% of the speed of a standard construct on a DEC Alpha and at 60% of the base speed on an RS6000.

The study of the use of rank-3 arrays was intended to look at options for a TRAC Vessel data structure. The one important result in that study was that indirect addressing within an equivalent rank-1 array was faster than computation within a rank-3 array (~10% faster on an IBM RS6000 and ~25% faster on a DEC Alpha).

V.2. Tests with Simple Subroutines

The basic tests created by John Turner of LANL compare calculations in a slightly different environment than the simple studies described above. In this series of tests, the idea of multiple components is introduced, slightly more computation is performed per cell, and some temporary array allocation is included. Two alternative data structures are explored in his tests. The first stores values for a given property (pressure, temperature, etc.) in a single array covering all components in the system. Component-specific data are accessed by providing start and end positions in the array for each component. The global data are defined in a module:

```fortran
module data_module
    use kind_module
    real(real_kind), dimension(:), allocatable :: pressure, temp, velc, density,
    velocity, mass
end module data_module
```

The arrays are allocated to the total number of cells in the system. For the ith component, the pressures are obtained from “pressure(jstart(i):jend(i))”. In this description, this data structure is designated Data Structure 1.

The second data type clearly recognizes a component structure. Property arrays are stored for each component via a derived data type:
In the test problem, all pointers are allocated directly so that the total number of array elements in all components matches that of Data Structure 1. This structure will be designated Data Structure 2.

Turner examines three methods of passing these data structures to a Subroutine. The first is to provide the basic arrays as arguments. For Data Structure 1, this takes the form:

```fortran
  call crunch1 (pressure(jstart(i):jend(i)), &
                 temp(jstart(i):jend(i)), &
                 velc(jstart(i):jend(i)), &
                 velocity(jstart(i):jend(i)+1), &
                 density(jstart(i):jend(i)+1), &
                 mass(jstart(i):jend(i)))
```

For Data Structure 2 this takes the form:

```fortran
  call crunch1 (comp(i)%pressure, comp(i)%temp, comp(i)%velc, &
                & comp(i)%velocity, comp(i)%density, comp(i)%mass)
```

In both instances, a loop over all components is performed around the call, and a loop over all cells in the component is performed within the subroutine. This method of passing information is designated Passing Method 1. One subtlety that should be noted about this method is that array segments are being passed in the first instance and pointers to full arrays in the second. Full information on these arguments is supplied by wrapping the subroutine in a module to provide an implied interface. This wrapping procedure is used for all subroutines in the test.

Passing Method 2 uses the direct approach of providing the full data structure to the subroutine via the use of a module and communicating the necessary index information via the argument lists. For Data Structure 1, the starting and ending array index for the ith component is passed.

```fortran
  call crunch2 jstart(i), jend(i)]
```
For Data Structure 2, the index of the component being computed is passed in the argument (call crunch4(i)), and the Subroutine performs the calculations for all cells contained in that component.

The dual-array nature of Data Structure 2 gives rise to one other possible method of information passing. The element of the derived type for the ith component is provided as an argument:

\[
\text{call crunch3(comp(i))}
\]

and the subroutine treats the structure as a scalar-derived type (pipe_type), computing with all elements of all array components of that derived type. This will be referred to as Passing Method 3.

The benchmark was extended by Graydon Uyeda (NRC), with the addition of a data type that treats state variables as scalars at the lowest level.

```fortran

  type cell_type
    real(real_kind) :: spressure, stemp, sdensity, svelc, svelocity, smass
  end type cell_type

  type node_type
    integer(int_kind) :: ncells
    type(cell_type), dimension(1000) :: node
  end type node_type

!basic pipe structure with node array and scalar values
  type(node_type), dimension(:), allocatable, target :: pipe

The use of "dimension(1000)" was not necessary. The attributes "pointer, dimension(:)" should be used in a dynamic data structure. This structure is designated Data Structure 3.

Passing Methods 1 and 2 were tested for Data Structure 3, along with some variants that were slower. Passing Method 1 takes the form:

```fortran

    do igsu=l,ncells
      call crunch8 (pipe(i)%node(igsu)%spressure, pipe(i)%node(igsu)%stemp,
        &
        pipe(i)%node(igsu)%smass, pipe(l)%node(igsu)%sdensity, &
        pipe(i)%node(igsu)%svelocity,&
        pipe(i)%node(igsu+1)%svelocity)
    enddo
```

Because the lowest level of the derived type is scalar, the DO loop over cells is located outside the subroutine call.
Passing Method 2 has the same calling sequence, as seen in Data Structure 2 [call crunch5 (i)]. Implementation of the structure within the subroutine looks like this:

\[ \text{ncells = pipe(icomp)%ncells} \]

\[ \text{do igsu=1,ncells} \]

\[ \begin{align*}
\text{pipe(icomp)%node(igsu)%stemp} &= \text{pipe(icomp)%node(igsu)%stemp*pipe(icomp)%node(igsu)%sdensity} \\
\text{pipe(icomp)%node(igsu)%spressure} &= \text{pipe(icomp)%node(igsu)%spressure*pipe(icomp)%node(igsu)%stemp} \\
\text{pipe(icomp)%node(igsu)%svelc} &= \text{pipe(icomp)%node(igsu)%svelc = half*(pipe(icomp)%node(igsu)%svelc + pipe(icomp)%node(igsu+1)%svelc)} \\
\text{pipe(icomp)%node(igsu)%smass} &= \text{pipe(icomp)%node(igsu)%smass = pipe(icomp)%node(igsu)%sdensity*} \\
\end{align*} \]

\[ \text{endo} \]

When considering data layout, Data Structures 1, 2, and 3 are analogous to rank 1, 2, and 3 arrays respectively. Thus, when the full extent of the associated data type is used (as above), we would expect Data Structure 1 to be the fastest, Data Structure 2 to be next in performance, and Data Structure 3 to be the slowest of the three. The benchmark as modified at the NRC did not show this behavior for Passing Method 2, leading to a careful study and eventual modifications to the benchmark.

Basic functionality of the benchmark on PSU machines required two changes to the code received from the NRC. The function “etime” that was used originally by LANL is not standard to all UNIX systems, and so it was replaced with the F90 “system_clock” intrinsic subroutine. This is a wall clock routine that required the use of machines when no other users were present and several runs to check repeatability. The second change made to the benchmark at PSU was simply to extend the allocated size of some arrays to stop the use of undefined numbers (which caused a floating-point error on DEC Alpha).

\[ \text{allocate (pressure(ntot+1), temp(ntot+1), density(ntot+1), mass(ntot+1), velocity(ntot+1))} \]

\[ \text{allocate (pressure_init(ntot+1), temp_init(ntot+1), density_init(ntot+1), mass_init(ntot+1), &} \]

\[ \text{velocity_init(ntot+1))} \]

\[ \text{allocate (pressure_ref(ntot+1), temp_ref(ntot+1), density_ref(ntot+1), mass_ref(ntot+1), &} \]

\[ \text{velocity_ref(ntot+1))} \]

\[ \text{allocate (pressure_diff(ntot+1), temp_diff(ntot+1), density_diff(ntot+1), mass_diff(ntot+1), &} \]

\[ \text{velocity_diff(ntot+1))} \]
In looking at timing anomalies, we noted a serious distortion in the sample calculation. Most, but not all test problems were allocating a temporary cell center velocity array. This nonuniformity was corrected by allocating the space for the cell center velocity once at the beginning of the program as reflected in the test program “nrc1.f”. However, this program still showed Data Structure 3 running ~25% faster than the best of the others for an IBM RS6000 (but ~40% slower for a DEC Alpha). This was traced to the fact that the test calculations are implemented with DO loops for Data Structure 3 and with F90 vector constructs for the other two Data Structures. The vector constructs inhibit some optimization available to these machines within a DO construct and give a particularly large disadvantage to a super-scalar machine like the RS6000 relative to the short-vector Alpha. All test cases were reimplemented using DO loops in the test calculations. This program can be found in the file nrc2.f. Significant speed improvements were seen by shifting to the DO loops, suggesting the need for some continued study of vector constructs and programming guidelines to prevent their overuse.

Table V.1 presents the results of the final benchmark problem (nrc2.f) for the three Data Structures and three Passing Methods listed above. Relative speed (a larger number is better) is given for runs on an IBM RS6000, DEC Alpha, Sun, and Cray. Speed is normalized on each machine to the results for Data Structure 1 (simple allocated arrays in a module) and Passing Method 1 (everything through the argument list). Results for the RS6000 and DEC Alpha were obtained with the vendor F90 compiler using the O3 level of optimization.

Relative speeds on the IBM RS6000 were found to be reproducible to within 0.01. Those for the DEC Alpha were repeatable to within 0.02. However, numbers should be viewed as trends because they will vary with compiler and CPU chip versions.

The remarkably poor results for Data Structure 2, Passing Method 2 (component based via a module) on a DEC Alpha can be attributed to failure

### Table V.1

#### RESULTS OF FINAL BENCHMARK PROBLEM

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Data Structure</th>
<th>Passing Method</th>
<th>IBM RS6000</th>
<th>DEC Alpha</th>
<th>Sun UltraSparc</th>
<th>DEC Pentium Pro</th>
<th>Cray</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.04</td>
<td>1.04</td>
<td>1.05</td>
<td>1.45</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0.96</td>
<td>1.16</td>
<td>0.85</td>
<td>1.03</td>
<td>1.07</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.69</td>
<td>0.08</td>
<td>0.56</td>
<td>0.49</td>
<td>0.05</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0.69</td>
<td>0.73</td>
<td>0.61</td>
<td>1.0</td>
<td>0.05</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.76</td>
<td>0.35</td>
<td>0.83</td>
<td>0.82</td>
<td>0.04</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0.85</td>
<td>0.13</td>
<td>1.03</td>
<td>1.04</td>
<td>1.43</td>
<td></td>
</tr>
</tbody>
</table>
of optimization because of the complexity of expressions. A similar situation exists for Data Structure 3, Passing Method 2.

As expected, Data Structure 3 is at a significant disadvantage on any "vector" machine. Because pipelined arithmetic units are not likely to disappear from future machines, this is a significant consideration in the choice of a Data Structure (and Code Structure). However, tests in the next section show that this disadvantage exists only in routines performing very simple computations. For the complex operation mix found in many thermal-hydraulic property routines, the performance of Data Structure 3 is at the same level as the others.

V.3. THERMO Benchmark

The previous benchmarks focused on situations with short or nonexistent argument lists. TRAC contains a significant number of subroutines with a relatively large number of arguments (>10). One such routine, THERMO, was selected as a basis for a third set of benchmark problems. This approach also introduces the effect of a larger and more varied set of operations per argument. As will be seen, this expansion of machine operations results in relatively uniform performance among the three Data Structures covered by the previous test set.

A total of eight variations of the data structure were tested. The simplest test was the use of statically dimensioned arrays. This was followed by the use of simple allocated arrays. The next step placed THERMO and all subsidiary routines into individual modules (wrapping), providing them with an implicit interface. For long arrays (>10), timing of these first three options did not vary significantly.

In TRAC-PF1/MOD2, all thermodynamic derivatives and saturation enthalpies are stored cell-wise in the "dr" array. Tests were generated for two more readable options: (1) converting "dr" to a derived type with clear component names or (2) rearranging the data structure to place each derivative in a separate array. The second option has the disadvantage of approximately doubling the length of THERMO's argument list, which is reflected to a small degree in some timing results. The option of using a derived type produced no major changes in run time, as should be expected from previous results.

The last of the eight variations explored Data Structures 2 and 3. Data Structure 2 was tested for both of the above options for replacement of the "dr" array and ran at roughly the same speeds as the corresponding tests using Data Structure 1. Data Structure 3 was simply tested for the case of a derived type replacement for "dr". Speeds of this test were within 1% of the fastest obtained for other tests.
V.3.1. Test Program th1.f
The array data structure is established with the statements:

```
parameter (nc=100, nrepeat=200, ntest=5)
real(dbl) dr(18, nc), p(nc), tl(nc), tv(nc), el(nc), ev(nc),
& tsat(nc),
& rol(nc), rov(nc), pa(nc), rova(nc), tssn(nc), eva(nc),
& sattmp, cptime(ntest), cpmean, stdev
```

THERMO is driven by the loops:

```
do it = 1, ntest
   call system_clock (icstart, irate, icmax)
   do i = 1, nrepeat
      call thermo(p, el, ev, tl, tv, tsat, rol, rov, pa, rova, tssn, eva,
1     dr, iop, 1, nc)
   enddo
   call system_clock (icstop, irate, icmax)
   cptime(it) = float(icstop-icstart)/float(irate)
endo
do it = 1, ntest
```

The outer DO loop is used to check the consistency of timing results. Output from the program provides timing results for each pass through and the mean and standard deviation for these times. Values of nrepeat and nc must be chosen for any given study so that their product is a constant.

V.3.2. Test Program th2.f
The allocated array data structure is established with the statements:

```
integer, parameter :: ntest=5
integer :: nc=100, nrepeat=200, ndriv=18, jstart=1, jend=100
logical fexist
real(dbl), allocatable :: dr(:,:), p(:), tl(:), tv(:), el:,
& ev(:,), tsat(:,),
& rol(:, rov(:, pa(:, rova(:, tssn(:, eva(:,
allocate( dr(ndriv, nc), p(nc), tl(nc), tv(nc), el(nc))
allocate( tsat(nc), rol(nc), rov(nc), pa(nc), rova(nc))
allocate( tssn(nc), eva(nc), ev(nc) )
```

Values of nc, nrepeat, jstart, and jend can be reset in an input file (array size).
THERMO is driven by the loops:

```fortran
    do it = 1,n\text{test}
        call \text{system\_clock}(ic\text{start},irate,ic\text{max})
        do i = 1,n\text{repeat}
            call thermo(p,el,ev,tl,tsat,rol,rov,pa,rova,tssn,eva,
                        1 dr,iop,j\text{start}, j\text{end})
        enddo
        call \text{system\_clock}(ic\text{stop},irate,ic\text{max})
        c\text{ptime}(it) = float(ic\text{stop}-ic\text{start})/float(irate)
    enddo
```

The presence of j\text{start} and j\text{end} in the argument list permits study of the use of array sections.

V.3.3. Test Program th3.f
The array data structure and driving loops are identical to those in th2.f. Subroutines are wrapped as follows:

```fortran
module thermo\_mod
    use varkind
    implicit none

    private
    public :: thermo

    contains

    subroutine thermo(p,el,ev,tl,tsat,rol,rov,pa,rova,
                        tssn,eva,dr,iop,j\text{start}, n\text{cells})
        use tsatcn
        use hev\_mod
        use satder\_mod
        use sat\text{tmp}\_mod
        use sat\text{prs}\_mod
        use rholiq\_mod
        implicit none

        subroutine thermo evaluates the thermodynamic properties of h2o

        do 300 j=j\text{start}, n\text{cells}
```

```fortran
        ... do 300 j=j\text{start}, n\text{cells}
        ...```
300 continue
    return
end subroutine thermo

end module thermo_mod

The above USE statements refer to modules containing subprograms used by THERMO.

V.3.4. Test Program th4.f
This test converts the TRAC thermodynamic derivative array (dr) to a derived type with the following modules:

module varkind
    integer, parameter :: dbl=selected_real_kind(12,50)
end module varkind

module drivMod
    use varkind
    type drivType
        real (dbl) dtsdp,deldp, devdp, deldt,
        & devdt, drolp, drovp, drolt,
        & drovt, hvst, hlst, dhvsp, dlhsp,
        & dtssp, devat, devap, drvap, drvat
    end type drivType
end module drivMod

The derived type is applied to “dr” with

    type (drivType), allocatable :: dr(:)

and allocated with

    allocate( dr(nc), p(nc), tl(nc), tv(nc), el(nc))

THERMO is driven with the same coding used in th2.f and th3.f.

V.3.5. Test Program th5.f
The array data structure is expanded to include separate arrays for the thermodynamic derivatives and saturation enthalpies:

    real(dbl), allocatable :: p(:,tl(:, tv(:, el(:,
    & ev(:, tsat(:,
    & rol(:, rov(:, pa(:, rova(:, tssn(:, eva(:,
        real (dbl), allocatable :: dtsdp(:,deldp(:, devdp(:,
        & deldt(:, devdt(:, drolp(:, drovp(:, drolt(:,

41
& drovt(:,), hvst(:,), hlst(:,), dhvsp(:,), dhlsp(:,),
& dtssp(:,), devat(:,), devap(:,), drvap(:,), drvat();

which are allocated with the statement:

allocate (dtsdp(nc),deldp(nc), devdp(nc), deldt(nc),
& devdt(nc), drolp(nc), drovp(nc), drolt(nc),
& drovt(nc), hvst(nc), hlst(nc), dhvsp(nc), dhlsp(nc),
& dtssp(nc), devat(nc), devap(nc), drvap(nc), drvat(nc))

THERMO is driven by the modified inner loop:

    do i = 1,nrepea
        call thermo(p,el,ev,t,tl,tv,tsat,rol,rov,pa,rova,
        & tssn,eva, dtsdp,deldp, devdp, deldt,
        & devdt, drolp, drovp, drolt,
        & drovt, hvst, hlst, dhvsp, dhlsp,
        & dtssp, devat, devap, drvap, drvat,
        & iop,jstart,jend)
    enddo

V.3.6. Test Program th4a.f
This program represents a modification of th4.f to include a component-based
data structure with allocated array pointers for thermodynamic variables
(Data Structure 2). This required additional derived-type definitions.

module dualTimeModule
    use varkind
    type dual
        ! Cell-center arrays
        real(dbl), pointer, dimension(:) :: p,t, tlv
        & ev, rol, rov, pa, rova, eva
        ! Cell-face arrays
        end type dual
    end module dualTimeModule

module ptab1dModule
    use varkind
    use dualTimeModule
    use drivMod
    type hydro1D
        type (dual) :: old, new
        ! Single-time arrays
        real(dbl), pointer, dimension(:) :: tsat, tssn
        type (drivType), pointer :: dr(:)
end
end type hydro1D
end module ptabl1dModule

module hydro1dModule
use ptabl1dModule
type (hydro1D), allocatable, dimension(:) :: comp1d
end module hydro1dModule

The derived-type drivType has not changed from what was defined in test th4.f. After using the above modules, the driver program allocates space with the statements:

allocate (comp1d(n test))
do i=1,n test
allocate( comp1d(i)%dr(nc), comp1d(i)%new%p(nc), &
comp1d(i)%new%tl(nc), &
comp1d(i)%new%tv(nc), comp1d(i)%new%el(nc))
allocate( comp1d(i)%tsat(nc), comp1d(i)%new%rol(nc), &
comp1d(i)%new%rov(nc), comp1d(i)%new%pa(nc), &
comp1d(i)%new%rova(nc))
allocate( comp1d(i)%tssn(nc), comp1d(i)%new%eva(nc), &
comp1d(i)%new%ev(nc))
enddo

THERM0 is driven by the loops:

do i = 1,n test
  call system_clock (icstart,irate,icmax)
do j = 1,n repeat
  call thermo(comp1d(i)%new%p,comp1d(i)%new%el, &
comp1d(i)%new%ev,comp1d(i)%new%tl, &
comp1d(i)%new%tv,comp1d(i)%tsat,comp1d(i)%new%rol, &
comp1d(i)%new%rov,comp1d(i)%new%pa,comp1d(i)%new%rova, &
comp1d(i)%tssn,comp1d(i)%new%eva, &
comp1d(i)%dr,iop,jstart, jend)
endo
cftime(i) = float(icstop-icstart)/float(irate)
endo

V.3.7. Test Program th4b.f

This is based on th4a.f, with the necessary changes added to obtain a data structure in which scalar thermodynamic variables are grouped in a cell-derived type (Data Structure 3). It is important to note that most of the work to create this program was accomplished with two global replacement commands in the text editor “tedi”:
and THERMO was converted to a scalar-based subroutine with the replacement:

```plaintext
!pal; subroutine thermo; sr1,
lsa.; end; sr2.,
rx r1,r2 ; ..(&);..1
```

Here, "..&" means any alphanumeric string (variable name), and "..1" represents the placement of the first such string in the successful search. Similar results can be obtained with "sed" or a capable scripting language such as Perl. Modules for derived types, type statements, and argument lists were modified by hand. The first global pattern modification is very general, based on the similarities in the two data structures. The second results from a specific knowledge of the local form of variables within THERMO. A similar modification pattern could be expected in TRAC should a choice be made to extract DO loops from low-level subprograms, modifying them to operate only on scalar variables.

The modules containing revised derived types are:

```plaintext
module dualTimeModule
    use varkind
    type dual
        ! Cell-center Variables
        real(dbl) p,tl,tv,el,
        & ev, rol, rov, pa, rova, eva
    end type dual
end module dualTimeModule

module singleTimeMod
    use varkind
    use drivMod
    type singleTime
        real(dbl) :: tsat, tssn
        type (drivType) :: dr
    end type singleTime
end module singleTimeMod

module ptabldModule
    use dualTimeModule
    use singleTimeMod
    type hydro1D
```
type (dual), pointer :: old(:), new(:)
!
   Single-time arrays
   type (singleTime), pointer :: once(:)
end type hydro1D
end module ptab1dModule

module hydro1dModule
use ptab1dModule
   type (hydro1D), allocatable, dimension(:) :: compld
end module hydro1dModule

The module defining drivType has not changed in this implementation from the original in th4.f.

Allocation of space is a much simpler matter under this data structure.

do i=1,ntest
   allocate( compld(i)%once(nc),compld(i)%new(nc) )
endo

THERM0 now requires an additional external driving loop because the internal loop has been removed.

do i = 1,ntest
   call system_clock (icstart,irate,icmax)
do j = 1,nrepeat
   do k = jstart,jend
      call thermo(comp1d(i)%once(k)%p,comp1d(i)%new(k)%el,
      & comp1d(i)%new(k)%ev,comp1d(i)%new(k)%tl,
      & comp1d(i)%new(k)%tv,comp1d(i)%once(k)%tsat,
      & comp1d(i)%new(k)%rol,comp1d(i)%new(k)%rov,
      & comp1d(i)%new(k)%pa,comp1d(i)%new(k)%rova,
      & comp1d(i)%once(k)%tssn,comp1d(i)%new(k)%eva,
      & comp1d(i)%once(k)%dr,iop)
endo
endo
   call system_clock (icstop,irate,icmax)
cptime(i) = float(icstop-icstart)/float(irate)
endo

V.3.8. Test Program th5a.f
This represents a transformation of th5.f to the component data structure implemented in th4a.f (Data Structure 2). Only one module must be changed.
module ptab1dModule
  use varkind
  use dualTimeModule
  type hydro1D
    type (dual) :: old, new
  ! Single-time arrays
    real(dbl), pointer, dimension(:) :: tsat, tssn
    real(dbl), pointer, dimension(:) :: dtstdp, deldp,
    & devdp, deldt, devdt, drolp, drovp, drolt,
    & drovt, hvst, hlst, dhvsp, dhlsp,
    & dtssp, devat, devap, drvap, drvat
  end type hydro1D
end module ptab1dModule

Allocation follows that of tf4a.f, with the addition of:

allocate (comp1d(i)%dtsdp(nc), comp1d(i)% deldp(nc),
  & comp1d(i)%devdp(nc), comp1d(i)% deldt(nc),
  & comp1d(i)%devdt(nc), comp1d(i)% drolp(nc),
  & comp1d(i)%drovp(nc), comp1d(i)%drolt(nc),
  & comp1d(i)%drovt(nc), comp1d(i)%hvst(nc), comp1d(i)%hlst(nc),
  & comp1d(i)%dhvsp(nc), comp1d(i)%dhlsp(nc),
  & comp1d(i)%dtssp(nc), comp1d(i)%devat(nc),
  & comp1d(i)%devap(nc),
  & comp1d(i)%drvap(nc), comp1d(i)%d Drvat(nc))

THERMO is driven by the loop structure seen in th4a.f, with a call modified
after the pattern established in th5.f:

call thermo(comp1d(i)%new%p, comp1d(i)%new%el,
  & comp1d(i)%new%ev, comp1d(i)%new%t,
  & comp1d(i)%new%tv, comp1d(i)%tsat, comp1d(i)%new%rol,
  & comp1d(i)%new%rov, comp1d(i)%new%pa, comp1d(i)%new%rova,
  & comp1d(i)%tssn, comp1d(i)%new%eva,
  & comp1d(i)%dtsdp, comp1d(i)%deldp,
  & comp1d(i)%devdp, comp1d(i)%deldt,
  & comp1d(i)%devdt, comp1d(i)%drolp,
  & comp1d(i)%drovp, comp1d(i)%drolt,
  & comp1d(i)%drovt, comp1d(i)%hvst, comp1d(i)%hlst,
  & comp1d(i)%dhvsp, comp1d(i)%dhlsp,
  & comp1d(i)%dtssp, comp1d(i)%devat,
  & comp1d(i)%devap, comp1d(i)%drvap,
  & comp1d(i)%drvap, iop, jstart, jend)
V.3.9. Results of the THERMO Benchmark

The program th1.f simply was hardwired to run 200 repeats of thermodynamic evaluations on a 100-element array. This confirmed earlier relationships observed between the use of static and allocated arrays. On the IBM RS6000, no timing difference was observed, and on the DEC Alpha, the allocated arrays actually ran ~2% faster than the static arrays. Some feature of allocated arrays may be permitting slightly better optimization on the DEC compiler.

The other cases were all run for four array configurations. Full calculations were performed for arrays with lengths of 100, 10, and 1. An array section of 10 elements out of a 100-element array also was evaluated. Timings were obtained by running 200 repetitions of the 100-element case, 2000 repetitions of the two 10-element cases, and 20,000 repetitions of the 1-element case. The single-element arrays were used to obtain information on the overhead of the argument list. Times for the 10- and 100-element array runs were within 1% for the DEC Alpha. On the IBM RS6000, the 100-element array runs took ~5% more time than the basic 10-element array run, perhaps because of cache.

As expected, dropping to the case with one element per array (one cell) does not degrade the performance of the cell grouped test th4b.f (Data Structure 3). Other data structures suffered varying slowdowns, from the array size with the fastest time to the time for the one-cell array. For the IBM RS6000, approximate slowing because of argument list overhead was

- ~10% for a simple allocated arrays (th2.f, th3.f),
- ~15% for simple allocated arrays with a derived-type array of derivatives (th4.f),
- ~30% for simple allocated arrays with derivatives in separate arrays (th5.f),
- ~15% for component based arrays and derived-type derivatives (th4a.f), and
- ~37% for component based arrays with derivatives in separate arrays (th5a.f).

For the DEC Alpha, approximate slowing caused by argument list overhead was
• ~6% for a simple allocated arrays (th2.f. th3.f),

• ~6% for simple allocated arrays with a derived-type array of derivatives (th4.f),

• ~11% for simple allocated arrays with derivatives in separate arrays (th5.f),

• ~20% for component-based arrays and derived-type derivatives (th4a.f), and

• ~20% for component-based arrays with derivatives in separate arrays (th5a.f).

These numbers provide some cautions for extreme cases involving large numbers of TRAC components with very few cells per component. However, the more important measure is the comparison between the fastest times of each test program. For the IBM RS6000, in all cases, the fastest data structure was the statically dimensioned arrays without subroutine wrappers (th2.f). Other test cases all had subroutine wrapping in addition to the listed features and were slower by the following approximate percentages:

• ~1% simple allocated arrays (th3.f),

• ~2% simple allocated arrays with thermodynamic derivatives in a derived type (th4.f),

• ~8% simple allocated arrays with thermodynamic derivatives in separate arrays (th5.f),

• ~4% component-based array pointers with derivatives in a derived type (th4a.f),

• ~9% component-based array pointers with derivatives in separate arrays (th5a.f), and

• ~1% properties grouped by cell in a derived-type data structure (th4b.f).

For the DEC Alpha, timings were much closer. When thermodynamic derivatives were not in separate arrays, static array, simple allocated array, and the cell grouped derived-type cases (th1.f, th2.f, th3.f, th4.f, th4b.f), all matched very closely in timing. Creation of separate arrays for thermodynamic derivatives resulted in run times that were slower by 11%
with simple allocated arrays (th5.f) and by 20% with the component-based array pointers (th5a.f). The use of component-based array pointers and a derived type for the thermodynamic derivatives slowed execution only by 3%.

V.4. Data Structure Selection
The relatively close performance of several options permits choices based on other important criteria: readability, maintainability, flexibility, and schedule. The first three favor the component-based options. Considerations of schedule and continuity in the reengineering process dictate that the current selection be component-based, with array pointers assigned to each property. This saves time in changing and debugging large numbers of subprograms with internal DO loops over an array of cell properties. The ease and flexibility demonstrated in the conversion from tests th4a.f to th4b.f, indicate that as time permits, other data structures can be explored with relatively small time requirements for changing the data structure itself. In particular, later stages of modernization could include tasks to convert low-level subroutines to operate on scalars, thus moving DO loops to the next higher level in the calling tree. The effort to then convert to the cell-oriented data structure would be very small compared with the time spent converting the low-level subroutines.
APPENDIX A
SPECIAL SYSTEM-WIDE DATA STRUCTURES

Although most of the system-wide data structures are simple arrays, there are more complex data structures that will be made much more readable in F90. For example, before the advent of derived types in F90, one method of implementing such structures was to use 2D arrays, in which the row indices identified an element of the data structure. The JUN array in TRAC is an example where JUN(2,JPTR) is the component number for the JPTR-th input junction and JUN(1,JPTR) is the corresponding junction number. The BD or boundary array is another example of this technique. The conversion to derived types for these two simple cases is straightforward. Example A.1 gives a code fragment from the current code showing the current boundary array, whereas Example A.2 uses the same fragment to illustrate how the boundary array will be implemented as a derived type. Details of the proposed conversion to derived types for the more complex control system data structure are given in App. B.

A second example of a complex data structure used in TRAC is the "ragged" 2D array, which is an array where the number of rows is different for different columns. For example, TRAC uses such an array where the IL-th "column" contains the component numbers of the components in the IL-th loop, as shown in Example A.3.

```fortran
SUBROUTINE pipe1(bd,vsi,lenbd)
DIMENSION vsi(1), bd(lenbd,1)
....
CALL setbd(bd(1,js1),bd(1,js2),1,ncells,vsi(js1),vsi(js2))
....
CALL jld ...
....
SUBROUTINE jld(b,icell,iface,ifc2,...,xnum)
! fills boundary array at component junction.
....
DIMENSION b(nbd)
INTEGER xnum

b(1)=a(ldx+icell)
b(2)=a(lvol+icell)
b(32)=a(lfa+iface)
b(37)=real(xnum)
```

TRAC often implements a ragged 2D array using indirect indexing by two different methods. With Method 1, there are two 1D arrays, where the primary array, say IORDER, contains the data of interest, and the supporting array, say LOOPN, gives the indices into the primary array of the elements where the various “ragged columns” begin. (With this type of arrangement, it is not necessary to have a second supporting array that gives the indices where the various “columns” end if they each end one array element before the next column begins. However, the supporting array then must have one extra “dummy” element to indicate where the last column ends.)

As illustrated in Example A.3, the first element of the IL-th “column,” IORDER(LOOPN(IL)), is the component number for the first component in the IL-th loop and the last element of that column, IORDER(LOOPN(IL+1)-1), is the component number for the last component in the IL-th loop.

Example A.2. Boundary array as a derived type.
It can be seen from this example that this implementation is a fairly natural one for accessing the individual elements of the column within the same procedure.

F90 supports an analogous implementation that retains the contiguous storage but allows the alternative, which is that the columns also be treated as separate entities where more convenient. Here we use an alternate supporting array that gives the number of elements in each column to support the DO loop. This is shown in Example A.4.

TRAC also uses an alternate method for implementing ragged 2D arrays, which we call Method 2, that is very similar to the F90 implementation of Method 1. Here the supporting array gives the number of elements in each column rather than its starting index. The use of this method is illustrated in Example A.5. In this example, the primary array, DVB, is referenced by an offset into the container array, whereas the supporting array, NJN, is referenced directly.
Method 2 is a simpler approach to use when the supporting array has additional purposes within the code structure because it avoids the need for an extra array simply to support the data structure. It is also appropriate when, as in this example, the "column" is being treated as a single entity such as a 1D array.

The most straightforward translation of this last example to F90 would be as shown in Example A.6.

However, is there any reason why the various "columns" in this case need to be stored contiguously at all, and why does the temporary pointer need to be recomputed on each call to this procedure? The different pointer values simply could be allocated once in a list, as shown in Example A.7.

```fortran
REAL, POINTER, DIMENSION(:) :: pt
DO il = 1, nloops
   pt=>dvb(ldvbil:ldvbil+njn(il)-1)
   CALL sub (pt)
ENDDO
```

Example A.6. Direct F90 translation of Method 2 for ragged 2D arrays.

```fortran
TYPE PointerToReal1D
   REAL, DIMENSION(:) :: pt
END TYPE PointerToReal1D
TYPE (PointerToReal1D), DIMENSION(1:NLOOPS) :: dvb
DO il = 1, nloops
   IF (Firstcall) CALL TRACAllo (dvb(il)%pt, njn(il))
   CALL sub (dvb(il)%pt)
ENDDO
```

Example A.7. Proposed F90 implementation of Method 2 for ragged 2D arrays.
APPENDIX B
CONTROL SYSTEM DATA STORAGE

We refer to the "Control System data" as those sets of data that implement TRAC-P's signal variables, trips, and control blocks, as well as the data sets that support these capabilities. There are 13 basic types of Control System data in all: (1) a set of 10 integers used to allocate storage based on the input specifications for the Control System [e.g., NTSV (the total number of signal variables)], (2) data for multipass control-parameter evaluation, (3) signal variable data, (4) control block data, (5) control block tabular data, (6) user-specified control system units-label data, (7) trip data, (8) trip-signal-expression signal data, (9) trip-controlled-trip signal data, (10) trip set-point-factor table data, (11) trip-initiated restart dump and problem termination data, (12) trip-initiated time-domain data, and (13) a single variable for the current problem time.

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. In the archival code (Version 5.4.25), the Control System variables, although conceptually individually named sets of data (as they appear to the user preparing an input deck), are actually stored in a single block within the A array, starting at pointer LCNTL; they are accessed during the course of a calculation via offsets from LCNTL into the A array. Real, integer, and character data types are mixed. There is excellent documentation in the archival code on the allocation of these data sets in the header of subroutine RCNTL. Although this documentation is very useful, the overall integer-pointer offset-into-A-array logic is difficult to read, maintain, and extend. Version 2.0 of TRAC-M retains the pointer-offset logic of the archival code for the Control System, but the data have been given individual arrays called act and ict, which have parameterized dimensions, in a new global COMMON block called ctCom. This was done as part of the initial portability effort for TRAC-M, for which the equivalencing of reals and integers was removed. The detailed documentation of the archival code in subroutine RCNTL has been retained.

The Control System data sets lend themselves ideally to F90's derived types and its standardization of storage allocation at run time. We illustrate a proposed data structure in Examples B.1, B.2, and B.3, which show the structure's use, definition, and allocation, respectively. Example B.1 shows a proposed modification to one of the Control System support routines (subroutine TRIP from TRAC-M Version 2.0) that reflects use of an F90-derived type for the trip data; all of the original coding is retained, with the parts superseded by the new F90 logic commented out. The trip data are now part of array tripData, the elements of which are of a (derived) data type for the trips. Each individual data member for each element of array tripData is identified by a meaningful name. The complex, error-prone, and difficult-to-
extend offset calculation is eliminated. Example B.2 shows a fragment of
proposed module controlSysStorage, which would define derived data types
for 11 of the Control System data sets (items 2–12 listed in the first paragraph
of this appendix) and also declare the variables for the other two data sets
(items 1 and 13). In this code fragment, the individual members for derived-
type tripDataT are declared, and array tripData then is declared as an
allocatable array of (derived) data type tripDataT. Example B.3 shows a
fragment of proposed subroutine controlSysInit, which would allocate storage
for the Control System data sets at run time (after the input is read).
SUBROUTINE trip(idtp, iset, tlaps, delsv)
IMPLICIT REAL*8 (a-h,o-z)
!
! subroutine trip inquires about the set status of a trip
! 
! input -- idtp = trip identification number
! output -- iset = trip set position (-1=on-reverse,
! 0=off,1=on-forward)
! tlaps = time since trip was last reset (s)
! delsv = signal value minus set point value
!
!!! USE new F90 module controlSysStorage
USE controlSysStorage
!!!
!!! pointers.h no longer needed
!!! INCLUDE 'pointers.h'
!!!
!!! INCLUDE 'contrllr.h'
INCLUDE 'iounits.h'
!!! blankcom.h no longer needed
!!! INCLUDE 'blankcom.h'
!!!
!!! IF (idtp.NE.0) THEN
!!!
!!! no longer needed: ntrp is in module controlSysStorage
!!!
ntrp=ict(lcnt1+4)
!!!
!!! IF (ntrp.GT.0) THEN
!!!
!!! replace following offset logic into ict array
!!!
!!! kpt=lcnt1+10+6*ict(lcnt1+5)+7*ict(lcnt1+1)
!!! & +23*ict(lcnt1+2)
!!! & +ict(lcnt1+3)+ict(lcnt1+4)+5*ict(lcnt1+6)
!!! DO n=1,ntrp
!!! IF (ict(kpt+1).EQ.idtp) GOTO 140
!!! kpt=kpt+80
!!! ENDDO
!!!
!!! with loop over trip id numbers in tripData
!!! (tripData is declared in module controlSysStorage)
!!!
Example B.1. Subroutine TRIP modified to use F90-derived type for trip data.
DO n=1,ntrp
    IF (tripData(n)%idtp.EQ.idtp) THEN
        iset=tripData(n)%iset
        tlaps=etime-tripData(n)%param1(2)
        delsv=tripData(n)%delsv
        RETURN ! trip found,
        ! return with iset, tlaps, and delsv
    ENDIF
ENDDO

!!! error condition
WRITE (imout,130) idtp
WRITE (iout,130) idtp
WRITE (itty,130) idtp
130 FORMAT (' ***** trip id =',i5,
    & ' was not found in the list',
    & ' of defined trips *****')
CALL error(1,'*trip* invalid trip id number',3)

!!! following no longer needed
!!! GOTO 150
!!!
!!!140 CONTINUE
!!! iset=ict(kpt+3)
!!! tlaps=etime-act(kpt+22)
!!! delsv=act(kpt+18)
ELSE
WRITE (imout,100) idtp
WRITE (iout,100) idtp
WRITE (itty,100) idtp
100 FORMAT (' ***** the set status of trip id =',i6,
    & ' is undefined because no trips exist *****')
CALL error(1,'*trip* there are no trips',3)
ENDIF
ENDIF
!
!!150 CONTINUE
RETURN
END

Example B.1 (cont). Subroutine TRIP modified to use F90-derived type for trip data.
Example B.2 (cont). Module controlSysStorage: define F90-derived data types for Control System data sets.
Example B.2 (cont). Module controlSysStorage: define F90-derived data types for Control System data sets.
Example B.2 (cont). Module controlSysStorage: define F90-derived data types for Control System data sets.

SUBROUTINE controlSysInit
IMPLICIT NONE
USE controlSysStorage

CALL TRACAllo (evalPassData, ntcp)
CALL TRACAllo (sigVarData, ntsv)
CALL TRACAllo (controlBlockData, ntcb)
CALL TRACAllo (cbTableData, ntcf)
CALL TRACAllo (ctrlSysUnitsData, (ntcb+ntrp+ntse))
CALL TRACAllo (tripData, ntrp)
CALL TRACAllo (sigExprData, ntse)
CALL TRACAllo (tctSignalData, ntct)
CALL TRACAllo (setPointData, ntsf)
CALL TRACAllo (tripDumpData, (min (ntdp,1) + ntdp))
CALL TRACAllo (tripTimeStepData, ntsd)

END SUBROUTINE controlSysInit

Example B.3. Subroutine controlSysInit: allocate storage for Control System data sets.

! trip-controlled trip (tct) data
TYPE tctSignalDataT

! ...

! trip set-point-factor table data
TYPE setPointDataT

! ...

! trip-initiated restart dump/problem termination data
TYPE tripDumpDataT

! ...

! trip-initiated time-domain data
TYPE tripTimeStepDataT

! ...

! etime: problem time corresponding to these control system data
! data
REAL :: etime
!
END MODULE controlSysStorage
APPENDIX C
TRACALLO GENERIC SUBROUTINE

F90 provides a very simple protocol for dynamic memory allocation and deallocation that supports all anticipated TRAC needs. This protocol uses the two built-in generic functions: allocate and deallocate. This appendix presents the actual source code for a prototype for the generic memory allocation function, TRACAllo, which will be built on top of the F90 allocation function, allocate. TRACAllo is an example of a user-defined generic function that is made possible by the new F90 INTERFACE statement. The generic function implementation allows a set of different functions distinguished by having differing types and/or numbers of arguments to be referenced with the same function or subroutine name. (This is sometimes called operator overloading.)

In Example C.1 we show a simple program using TRACAllo to allocate first a REAL array, then an INTEGER array, and, finally, another REAL array. We have intentionally given the last array an invalid size so as to illustrate the error-handling facility of TRACAllo. This program also illustrates the F90 extension, which allows the use of optional subroutine arguments. This feature is used here to provide for setting the optional initial value for allocated arrays. The output from this program is given in Example C.2.

```
PROGRAM TRACdyn
  USE kindModule
  USE TRACAlloModule
  IMPLICIT NONE
  REAL(realKind), POINTER, DIMENSION(:) :: x, xfunny
  INTEGER(intKind), POINTER, DIMENSION(:) :: iarray
  INTEGER(intKind), PARAMETER :: ncells = 100
  INTEGER(intKind), PARAMETER :: funny = 1000000000

  CALL TRACAllo (x, ncells, 'XI, initialvalue = 0._realKind)
  CALL TRACAllo (iarray, ncells, 'iarray')
  CALL TRACAllo (xfunny, funny, 'xfunny')

  STOP
END PROGRAM
```

Example C.1. TRACAllo test program.
Allocated REAL 1-D POINTER x to 100 words.
Allocated integer 1-D POINTER iarray to 100 words.
Got allocate error code 494
Allocation error for pointer to Real 1-D array.
Trouble allocating xfunny.
FORTRAN STOP TRACAllo.

Example C.2. Output from TRACAllo test program.

The TRACAllo test program uses two modules. The first module, shown in Examples C.3–C.5, implements the TRACAllo generic function. Example C.3 shows the overall structure of the module with place holders for the two specific subroutines, shown respectively, in the next two examples. In the prototype shown here, only the coding for REAL 1D and INTEGER 1D arrays is given. The completed TRACAllo functionality will have an interface to specialized subroutines to allocate all of the additional types of arrays required by TRAC-M.

The second module that is part of this prototype uses F90 intrinsic functions to set up a portable means of specifying the types of the variables and is shown in Example C.6. Clearly, TRAC-M will require a generalization of this module.

MODULE TRACAlloModule
  USE kindModule
  IMPLICIT NONE
  INTERFACE TRACAllo
    MODULE PROCEDURE real1DAallo
    MODULE PROCEDURE integer1DAallo
  END INTERFACE
CONTAINS

  (SUBROUTINE real1DAallo goes here).

  (SUBROUTINE integer1DAallo goes here).
END MODULE TRACAlloModule

Example C.3. Prototype TRACAllo F90 module.
SUBROUTINE real1DAlo (pt, n, name, initialValue)

IMPLICIT NONE

REAL(realKind), POINTER, DIMENSION(:) :: pt
INTEGER(intKind), INTENT(in) :: n
CHARACTER(len=*) , INTENT(in) :: name
REAL(realKind), OPTIONAL:: initialValue
INTEGER(intKind) errorStatus

IF (associated (pt)) THEN
    WRITE (*, *) name, ' is already allocated.'
    WRITE (*, *) 'Allocation error for pointer to real 1-D array.'
    stop 'TRACAllo'
ENDIF

allocate (pt(n), stat=errorStatus)

IF (errorStatus .ne. 0) then
    WRITE (*,*) 'Got allocate error code ', errorStatus
    WRITE (*, *) 'Allocation error for pointer to Real 1-D array.'
    WRITE (*,*) 'Trouble allocating ', name, '.'
    STOP 'TRACAllo'
ENDIF

IF (present (initialValue)) pt = initialValue

WRITE (*,*) 'Allocated REAL 1-D POINTER ', name, ' to ', n, ' words.'
END SUBROUTINE real1DAlo

Example C.4. Subroutine for REAL 1D arrays from the TRACAllo F90 module.
SUBROUTINE integer1DAllo (pt, n, name, initialValue)
IMPLICIT NONE

INTEGER(intKind), POINTER, DIMENSION(:) :: pt
INTEGER(intKind), INTENT(in) :: n
CHARACTER(len=*) , INTENT(in) :: name
INTEGER(intKind), OPTIONAL:: : initialValue
INTEGER(intKind) errorStatus

IF (associated (pt)) THEN
  WRITE (*, *) name, ' is already allocated.'
  WRITE (*, '*') 'Allocation error for pointer to Integer 1-D array.'
  stop 'TRACAllo'
ENDIF

allocate (pt(n), stat=errorStatus)

IF (errorStatus .ne. 0) then
  WRITE (*, '*') 'Allocation error for pointer to Integer 1-D array.', &
  'Trouble allocating ', name, '. '
  stop 'TRACAllo'
ENDIF

IF (present (initialValue)) pt = initialValue

WRITE (*,*), 'Allocated integer 1-D POINTER ', name, ' to ', n, ' words.'
END SUBROUTINE integer1DAllo

Example C.5. Subroutine for INTEGER 1D arrays from the TRACAllo F90 module.

MODULE kindModule
IMPLICIT NONE

INTEGER, PARAMETER :: realKind = selected_real_kind(15,307)
INTEGER, PARAMETER :: intKind = selected_int_kind (8)
END MODULE kindModule

Example C.6. Prototype module for specifying types of variables.
APPENDIX D
SUMMARY OF FOLLOW-ON TASKS AND FUTURE WORK

There are many references to possible follow-on tasks in the SDID. These are currently unfunded. They are included in the SDID as a way of clarifying the scope of work. These tasks are called out in this appendix for convenience. This list is not intended to suggest any prioritization of these tasks relative to each other or to additional TRAC modernization.

Follow-on Task 1. Better organization of common data.
This is a plan to develop better organization of the system-wide data currently in common blocks to make the data flow more understandable. In the original TRAC design, each common block was intended for a certain category of data as indicated by the common block name. However, this has degraded over time.

Follow-on Task 2. Better organization of the module replacing the PTRS common block. In TRAC-M, the new allocatable arrays corresponding to offset pointers in the PTRS common block will be placed in a module of the same name. Later, subgroups of these arrays will be placed in separate modules. This will make the data flow more obvious and improve code readability.

Follow-on Task 3. Study of data layout issues. This document addresses data structures only, not the actual data layout in the computer memory. For reasons discussed below, data layout can have a major impact on computational efficiency. However, the pointer feature of F90 allows us the flexibility of treating data layout separately from data structures. Consider a simple example of allocating two arrays, as shown in Examples D.1 and D.2.

In Example D.1, there is no necessary relationship between where the two arrays x and y are stored in memory. Now consider Example D.2.

```fortran
INTEGER, PARAMETER :: n = 100
REAL, POINTER, DIMENSION(:) :: x, y
allocate (x, n)
allocate (y, n)
CALL crunch (x, y, n)
```
Example D.1. Allocation of two not necessarily contiguous arrays.
**Example D.2. Allocation of two forced contiguous arrays.**

Here, with a minor change, we have used pointers to force the arrays \( x \) and \( y \) to be stored contiguously.

All computer hardware has a hierarchical memory structure such that some memory can be accessed much faster than other memory. A typical hierarchy for a single processor includes registers, various levels of cache, main memory, and secondary memory (such as disks), with memory access from fastest to slowest, respectively. The operating system may include a virtual memory or paging feature so that all or some subset of the memory is globally addressable and appears the same to the programmer; nonetheless, the computational efficiency is affected when the slower memory accesses actually occur. Parallel machines that use multiple processors add complexity in that on-processor memory access is faster than access of nearby processors and that, in turn, is faster than access of more distant processors. We will study the relationship of the control of data layout to computational efficiency, especially in the context of parallelization. We will modify the data layout as appropriate.

**Follow-on Task 4. Support of a static allocation option.** An option will be implemented into TRAC-M to generate a report giving the names and sizes of all of the dynamic arrays for the particular input model. This report then would be the basis for automated source code editing to create a custom, static version by replacing the declarations for dynamically allocated pointers with fixed array sizes. However, note that it is not generally possible to use exactly the same array sizes with static memory allocation as with dynamic. For example, consider an array of arrays. This could be implemented as an array of a derived type, with the components of the derived type being pointers to other arrays, as shown in Fig. D-1.
TYPE arrays 
  REAL, POINTER, DIMENSION(:) :: x 
  REAL, POINTER, DIMENSION(:) :: y 
END TYPE arrays 

TYPE (arrays), DIMENSION(:) :: arrayOfArrays 
allocate (arrayOfArrays(2))  
allocate (arrayOfArrays(1)%x(100))  
allocate (arrayOfArrays(1)%y(101))  
allocate (arrayOfArrays(2)%x(200))  
allocate (arrayOfArrays(2)%y(201)) 

Fig. D-1. Dynamic allocation of an array of arrays.

With this dynamic allocation, each component of each instance of the derived type can be allocated to a different size, as shown in Fig. D-1. However, when utilizing static allocation with this same data structure, each component of all instances of the derived type would have to have the same size, as shown in Fig. D-2.

Follow-on Task 5. Test the Turbine component and restore functionality as necessary. The Turbine component has a data structure slightly different from the other 1D components and has not been tested or maintained for a number of years. During the data structure reimplementation task, we will reimplement the data structures for the Turbine in a manner consistent with other 1D components; however, this will not be documented independently. Development of test problems, testing, debugging, and documentation of this component is needed before it can be utilized.

Follow-on Task 6. Reimplement the separator component properly as an option in the Tee component. The TRAC input manual references a separator component of type “sepd”. This is not actually a different component but simply the (mis)use of the component-type name to activate a special process model option within the Tee component. This will be reimplemented as an option.

TYPE arrays 
  REAL, POINTER, DIMENSION(200) :: x 
  REAL, POINTER, DIMENSION(201) :: y 
END TYPE arrays 

TYPE (arrays), DIMENSION(2) :: arrayOfArrays 

Fig. D-2. Static allocation of an array of arrays.
Follow-on Task 7. Reconsideration of built-in heat structures. The built-in heat structures (walls) in the fluid components represent a somewhat redundant capability in TRAC. Consideration of their removal has been given previously. However, users have requested that we retain this option because it is much more convenient than using a Heat Structure component, where its simple functionality is adequate. The built-in heat structures have their nodes aligned with the fluid cell centers, in contrast to the Heat Structure component. We should reconsider the need for this option if we ever change the alignment for the Heat Structure component.

Follow-on Task 8. Redesign of computational data structures to enhance extensibility and/or computational efficiency. Whether there will be a need for redesign to support major extensions or computational efficiency is not yet clear.

Follow-on Task 9. Selection of formatting standards and automated support. Uniform appearance is generally thought to make source code more readable. There are various automated tools to support formatting, including freeware available on the World Wide Web. These tools usually provide a wide choice of options. Automated tools allow code developers to use their personal style during development and then automatically conform to the standard when submitting code to the official code repository.
APPENDIX E
NEW AND DELETED SUBROUTINES

The following subroutines appearing in TRAC-PF1/MOD2, Version 5.4.25 have been eliminated.

<table>
<thead>
<tr>
<th>Old Routine</th>
<th>New Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>aiplen.f</td>
<td>igbrak.f</td>
</tr>
<tr>
<td>aivssl.f</td>
<td>igcomp.f</td>
</tr>
<tr>
<td>bfgrf.f</td>
<td>igfill.f</td>
</tr>
<tr>
<td>bits.f</td>
<td>ighstr.f</td>
</tr>
<tr>
<td>clrint.f</td>
<td>igpipe.f</td>
</tr>
<tr>
<td>ctain1.f</td>
<td>igplen.f</td>
</tr>
<tr>
<td>ctain2.f</td>
<td>igprzr.f</td>
</tr>
<tr>
<td>ctain3.f</td>
<td>igpump.f</td>
</tr>
<tr>
<td>endgrf.f</td>
<td>igraf.f</td>
</tr>
<tr>
<td>getbit.f</td>
<td>igsvcb.f</td>
</tr>
<tr>
<td>glevel.f</td>
<td>igtee.f</td>
</tr>
<tr>
<td>graf.f</td>
<td>igturb.f</td>
</tr>
<tr>
<td>grfget.f</td>
<td>igvlve.f</td>
</tr>
<tr>
<td>grfput.f</td>
<td>igvssl.f</td>
</tr>
<tr>
<td>icmpr.f</td>
<td>ilevel.f</td>
</tr>
<tr>
<td>idiff.f</td>
<td>ipack.f</td>
</tr>
<tr>
<td>ieeecvt.f</td>
<td>isamaxt.f</td>
</tr>
<tr>
<td></td>
<td>iunpack.f</td>
</tr>
<tr>
<td></td>
<td>ldchar.f</td>
</tr>
<tr>
<td></td>
<td>lentab.f</td>
</tr>
<tr>
<td></td>
<td>load.f</td>
</tr>
<tr>
<td></td>
<td>locpmp.f</td>
</tr>
<tr>
<td></td>
<td>loctee.f</td>
</tr>
<tr>
<td></td>
<td>locvlv.f</td>
</tr>
<tr>
<td></td>
<td>modify.f</td>
</tr>
<tr>
<td></td>
<td>movlev.f</td>
</tr>
<tr>
<td></td>
<td>packit.f</td>
</tr>
<tr>
<td></td>
<td>ptrsa.f</td>
</tr>
<tr>
<td></td>
<td>rlmacht.f</td>
</tr>
<tr>
<td></td>
<td>rdcm.f</td>
</tr>
<tr>
<td></td>
<td>ring0.f</td>
</tr>
<tr>
<td></td>
<td>rsperr.f</td>
</tr>
<tr>
<td></td>
<td>scputm.f</td>
</tr>
<tr>
<td></td>
<td>scputm.f</td>
</tr>
<tr>
<td></td>
<td>setic.f</td>
</tr>
<tr>
<td></td>
<td>sgbfat.f</td>
</tr>
<tr>
<td></td>
<td>sgbslt.f</td>
</tr>
<tr>
<td></td>
<td>shift.f</td>
</tr>
<tr>
<td></td>
<td>shiftr.f</td>
</tr>
<tr>
<td></td>
<td>switch.f</td>
</tr>
<tr>
<td></td>
<td>tcputm.f</td>
</tr>
<tr>
<td></td>
<td>unpkit.f</td>
</tr>
<tr>
<td></td>
<td>vdpcss.f</td>
</tr>
<tr>
<td></td>
<td>vpoint.f</td>
</tr>
<tr>
<td></td>
<td>wand.f</td>
</tr>
<tr>
<td></td>
<td>wiarr.f</td>
</tr>
<tr>
<td></td>
<td>wlabn.f</td>
</tr>
<tr>
<td></td>
<td>wor.f</td>
</tr>
<tr>
<td></td>
<td>wrlcm.f</td>
</tr>
<tr>
<td></td>
<td>wxor.f</td>
</tr>
</tbody>
</table>

Routine aiplen was deleted because it was no longer necessary with the deletion of the plenum inverted database.

Routine aivssl was deleted because it used pointer arithmetic and was not called by any of our test problems. Its functionality will be replaced in the F90 code.

Routine bits was deleted because its functionality was temporarily replaced by a C-language call.

Routines ctain1, ctain2, and ctain3 were removed because they were nonfunctional place holders.

Routine getbit was deleted because its functionality was temporarily replaced by a C-language call.

Routine icmpr was deleted because it made use of F77 untyped procedure interfaces.

Routine idiff was deleted because its use required nonstandard Fortran.

Routine ilevel was deleted when it became unnecessary with the conversion of vessel integer arrays to REAL*8 storage.
Routine ldchar was deleted because it made use of F77 untyped procedure interfaces.

Routine lentab was deleted because it made use of pointer arithmetic.

Routine load was replaced by loadn, which has logic for optional conversion to REAL*8 storage.

Routines matching pattern loc* were deleted because they used pointer arithmetic.

Routine ptrsa was deleted because it was unnecessary with the removal of the inverted vessel database.

Routine rdlcm was deleted because it used untyped vector copies.

Routine rlmacht returned a platform-dependent number specifying the relative precision of floating point arithmetic. It was temporarily replaced by a portable C routine.

Routine scputm was deleted because it was nonfunctional in the base code.

Routine seterr was deleted because it was nonportable.

Routine tcputm was replaced by a temporary C routine.

Routine vpoint was deleted with the removal of the nonportable SUN Pointer coding.

Routine wiarr was replaced by wiarn, which correctly treats REAL*8 storage of integer arrays.

Routine wlabn was replaced by wlabin, which correctly treats REAL*8 storage of integer arrays.

Routine rdlcm wrlcm was deleted because it used untyped vector copies.

The following routines were deleted when their bit-packing functionality was replaced by portable C calls: wand.f  wxor.f  wor.f  shift.f  .

The following routines were deleted in Version 1.2 because they were either used exclusively in trcgrf functionality or not called at all in the base code:

sgbfat.f  iunpack.f  grfget.f  igpump.f  unpkit.f  igsvcb.f  ieeeecvt.f  grfput.f  
    rspermf  glevel.f  ggraf.f  sbigsl.f  igbrak.f  ipack.f  clrint.f  igplen.f  bggrfr.f  
    igprzr.f  igfill.f  igvlve.f  iggraf.f  shiftr.f  igpipe.f  ighstr.f  packit.f  endgrf.f
The following routines have been added to TRAC during the modernization effort.

```
c2r.f    dmpxvlt.f    rdfl.t    rstvavlt.f
checksize.f    dmpavlt.f    rdptvl.t    rstvl.t
cleari.f    dmpavlt.f    rdpvlt.f    rstvsl.f
clrbrvlt.f    dmpsvl.t    rdpvlt.f    setrod.f
clrfivlt.f    getgen.f    rdpvlt.f    settype.f
clpivlt.f    getplent.f    rdpvlt.f    wiar.f
clrlvl.t    getpump.f    rdpvlt.f    wrlav.t
clrprvlt.f    getrod.f    rdtbvl.t    wbrvlt.f
clrpuvlt.f    gettee.f    rdtevl.t    wrfvl.t
clrvdvl.t    geturb.f    rdvavlt.f    wrfl.t
clrvtvl.t    gettype.f    rdvlt.f    wrprvl.t
clrvavl.t    getvalve.f    rdsavl.t    wrpvl.f
clrvsvl.t    getvsl.f    rstbrvl.t    wrprvl.t
dmpbrvlt.f    getss.f    rstfvl.t    wrptr.t
dmpfvl.t    i42r.t    rstfl.t    wrpvl.t
dmpfl.t    loadn.f    rstopvl.t    wrrdvl.t
dmpplvl.t    r2c.f    rtopvl.t    wrtbvl.t
dmpplvl.t    r2c32.f    rtopvl.t    wrettvl.t
dmpplvl.t    r2i4.f    rtpvvl.t    wrvavlt.t
dmpplvl.t    r2ii.f    rtpvvl.t    wrvlt.t
dmpplvl.t    rdbrvlt.f    rstrvl.t    wrvsl.t
```

The identification of temporary subroutines will be provided later.

The routine checksize was added to check for exceeding the static dimensions of temporary arrays.

The routine cleari was added to clear INTEGER arrays using a typed argument.

The routine loadn was added to provide logic for optional conversion to REAL*8 storage; it replaces routine load.

The routine settype was added to set a REAL type parameter using the CHARACTER*8 input description.

The routine setrod was added to set variables in the rod variable-length table, replacing the use of pointer arithmetic.

The routine wiarn correctly treats the REAL*8 storage of integer arrays and replaces routine wiarr.
The routine wlabin correctly treats REAL*8 storage of integer arrays and replaces routine wlabin.

The routines rstflt, dmpflt, wrflt, rdflt, and the routines matching the patterns rst*vlt, dmp*vlt, wr*vlt, and rd*vlt were added to replace vector copies using deleted routines wrlcm and rdlcm with element-by-element copies for restarting, dumping, writing to the A array, and reading from the A array, respectively.

The routines matching the pattern clr*vlt were added to initialize the variable-length tables element by element.

The routines matching the pattern get* were added to standardize intercomponent information passing using constant indices into data structures rather than pointer arithmetic, as was done in the deleted routines matching pattern loc*.

The routines r2c, c2r, and r2c32 were added to standardize and consolidate translations to and from real and character data types.

The routines i42r, r2i4, and r2i1i were added to copy real to integer arrays and vice versa. The was necessary because of the removal of the assumption of uniform word length.
APPENDIX F
CODING PRACTICES

The following list is only a start. A much more extensive list of coding practices must be developed. Los Alamos is just beginning the F90 effort and will add to this list as other coding practices firm up as we engage in the database restructuring effort.

- All new coding will be standard F90.

- All new variables will be explicitly typed, and all new routines will include IMPLICIT NONE statements.

- All Fortran keywords in new coding will be in all caps.

- All new variable names will have the first letter of each element capitalized except the first, as in pipeData.

- All derived type names will end in "T", as in pipeDataT. Underscores will not be used.

- Names will be long enough to be self-documenting, within reason, with a suggested limit of 15 characters.

- All new coding will be structured, with an indentation level of two spaces.

- “GOTO” statements will be used sparingly, if at all.

The use of case-sensitive coding rules in F90 is only for ease of reading and understanding the source. Experience from C and C++ is that a few simple rules for use of case greatly enhance a self-documenting source. In F90, the compiler is case insensitive; thus, any conflicts between names that differ only in case are immediately apparent.

In any case, available code revision control systems, such as CVS ( Concurrent Version System), provide for automation of pretty-printing upon submission of a new or modified source. We have not yet addressed source format issues (see Task 9 of App. D).

We have standards for source format during development, namely that all sources be compilable in both free and fixed format modes. Specifically, the old F77 fixed-format rules apply, with the following additional specifications.
1. Comments lines are indicated with a "!" in column 1.
2. Lines that are continued are indicated with a "&" in column 74.
3. Continuation lines are indicated with an "&" in column 6.