APPENDIX D
EXAMPLE TRAC-M UPDATE (OR CVS) USAGE

To be provided by the US NRC.

APPENDIX E
GRAPHICS (XTV/XMGR5) VARIABLES

E.1. Introduction

Appendix E lists the variables that are written to the graphics-data TRCXTV file. Subroutine `xtvdr` orchestrates the creation of the graphics file; each data edit is written by the subroutine responsible for that particular component or data structure. Those variables containing the parenthetical "Header variable only" do not vary with time and appear only in the graphics header edit.

Variables initially are listed by subroutine rather than by component to prevent multiple listings of the variables output by subroutine `xtv1D`. The format of the appendix makes it easy to determine all possible variables for a given component while still making it clear which variables apply to particular components. Because the exact variables available from a given calculation are dependent on options and input parameters, we have not maintained the sequence of the variables; however, we have alphabetized the variables for ease of reference. We have provided definitions and, as appropriate, the corresponding SI and English units. This listing is based on TRAC-M Version 3.0.

E.2. Global Variable Graphics

The global variables apply to the overall calculation as opposed to specific components or cells within a component. Subroutine `xtvgnpr` is responsible for these graphics variables, with the exception of `timet`, which is output by `xtvdr`.

Variable	Dimension	Description
`cputot`	1	Total CPU time (s) since time 0.0 s in the calculation.
`delt`	1	Timestep size (s).
`dprmax`	1	Maximum fractional pressure change over the current timestep (parameter used in the timestep-control logic).
`dtlmax`	1	Maximum liquid-temperature change (K, °F) over the current timestep (parameter used in the timestep-control logic).
`dtrmax`	1	Maximum HTSTR-component ROD or SLAB element wall temperature change (K, °F) over the current timestep.
`dtsmax`	1	Maximum saturation temperature change (K, °F) over the current timestep.
`dtvmax`	1	Maximum vapor-temperature change (K, °F) over the current timestep (parameter used in the timestep-control logic).
`timet`	1	Transient time (s) in the calculation.
`tnstep`	1	Total number of timesteps since time 0.0 s in the calculation.

E.3. Signal-Variable, Control-Block, and Trip-Signal Graphics

Subroutine `xtvcntl` is responsible for all of the signal variables, control blocks, and trip signals specified through input from the input-data file TRACIN and restart-data file TRCRST. Subroutine `xtvcntl` loops over all of the signal variables in the order of increasing magnitude of their ID numbers and similarly loops over all of the control blocks and all of the trips. The quantities written to the graphics file are

1. the parameter value of each signal variable at the current timestep, along with a figure label having its signal-variable ID number, parameter title, and units of the signal-variable parameter;

2. the output-parameter value from each control block at the current timestep, along with a figure label of its control-block ID number and the units of the control-block output parameter; and

3. the trip signal from each trip at the current timestep, along with a figure label of its trip ID number and the units of the trip signal.

For TRAC-P to output control-block, output-signal, and trip-signal units to the control-block and trip-signal figure labels, the user must specify those units through input by units-name labels. This is done when one or more of the NAMELIST-variable I/O-units flags `iogrf`, `ioinp`, `iolab`, and `ioout` has a value of 1 to specify Engish units. Users desiring all input and output in SI units with control-block, output-signal, and trip-signal graphics labels with SI units should input NAMELIST variables `iolab` = 1 while leaving `inlab` = 0 (default value). Inputing `inlab` = 3 would output a comment-labeled input-data file `inlab` in English units.

Variable	Dimension	Description
`sv`	1	Signal-variable data (although the dimension of each is 1, there are `ntsv` of them and each has its own units-name label).
`cb`	1	Control-block output (although the dimension of each is 1, there are `ntcb` of them and each has its own units-name label based on the user-defined units-name label of `cbxmin` and `cbxmax`).
`ts`	1	Trip-signal data [although the dimension of each is 1, there are `ntrp` of them and each has its own units-name label based on the user-defined units name label of `setpt(i)`, `i` = 1 to 2 or 4].

E.4. General 1D Hydraulic-Component Graphics

Subroutine `xtv1d` outputs the graphics-catalog variables that are common to all of the 1D hydraulic components (PIPE, PRIZER, PUMP, TEE, and VALVE). For TEE components, the dimension of cell-centered variables includes space for a phantom cell

between the main-tube and side-tube cells. This accounts for the fact that there are more interfaces than cells, and side-tube values are stored after main-tube values. In some cases, the outputting of parameter values depends on user-specified options in the TRAC-P input-data TRACIN file that cause those parameters to be evaluated. Note that because of wall heat conduction (nodes), these components may be listed as 2D components in XTV when nodes > 2.

Variable	Dimension	Description
alpn	ncellt	Cell gas volume fractions (–).
alven	ncellt	Cell liquid-side interfacial heat-transfer coefficients (W K^{-1}, Btu $°F^{-1}$ h^{-1}) [HTC * interfacial area].
alvn	ncellt	Cell-flashing interfacial heat-transfer coefficients (W K^{-1}, Btu $°F^{-1}$ h^{-1}) [HTC * interfacial area].
am	ncellt	Cell noncondensable-gas masses (kg, lb_m).
chtan	ncellt	Cell noncondensable-gas interfacial heat-transfer coefficients (W K^{-1}, Btu $°F^{-1}$ h^{-1}) [HTC * interfacial area].
chtin	ncellt	Cell gas-side interfacial heat-transfer coefficients (W K^{-1}, Btu $°F^{-1}$ h^{-1}) [HTC * interfacial area].
cifn	ncellt+1	Interface interfacial-drag coefficients (kg m^{-4}, lb_m ft^{-4}).
concn	ncellt	Cell dissolved-solute concentration ratio [kg(solute) kg^{-1} (liquid), lb_m(solute) lb_m-1(liquid)].
fa	ncellt+1	Interface flow areas (m^2, ft^2) (header variable only).
hgam	ncellt	Cell subcooled boiling heat flux (W m^{-2}, Btu ft^{-2} h^{-1}).
hil	ncellt	Cell wall liquid heat-transfer coefficients (W m^{-2} K^{-1}, Btu ft^{-2} $°F^{-1}$ h^{-1}).
hiv	ncellt	Cell wall gas heat-transfer coefficients (W m^{-2} K^{-1}, Btu ft^{-2} $°F^{-1}$ h^{-1}).
htlsci	1	Inner-surface heat loss (W, Btu h^{-1}) from the wall.
htlsco	1	Outer-surface heat loss (W, Btu h^{-1}) from the wall.
id	1	Component ID number (header variable only).
idr	ncellt	Cell wall heat-transfer regime numbers (–).
ncellt	1	Total number of cells, including phantom cell (header variable only).
njun	1	Number of junctions on this component.
nlegs	1	Number of legs (side tubes) on this component.
pan	ncellt	Cell noncondensable-gas partial pressures (Pa, psia).
pinteg	1	Total heat-transfer rate to the wall (w, Btu h^{-1}).
pn	ncellt	Cell total pressures (Pa, psia).
regnm	ncellt+1	Interface flow-regime numbers.
rmvm	ncellt+1	Interface fluid mass flows (kg s^{-1}, lb_m h^{-1}).
rmvf	ncellt+1	Interface gas mass flows (kg s^{-1}, lb_m h^{-1}).
roan	ncellt	Cell noncondensable-gas densities (kg m^{-3}, lb_m ft^{-3}).
roln	ncellt	Cell liquid densities (kg m^{-3}, lb_m ft^{-3}).
rom	ncellt	Cell mixture densities (kg m^{-3}, lb_m ft^{-3}).
rovn	ncellt	Cell gas densities (kg m^{-3}, lb_m ft^{-3}).

sn	ncellt	Cell plated-solute mass/fluid volume (kg m^{-3}, lb_m ft^{-3}).
tcen	1	Total heat convected to the fluid (W s, Btu).
tln	ncellt	Cell liquid temperatures (K, °F).
tsat	ncellt	Cell saturation temperatures (K, °F) based on the total pressures.
tssn	ncellt	Cell saturation temperatures (K, °F) based on the steam partial pressures.
tvn	ncellt	Cell gas temperatures (K, °F).
twan	1	Absolute error in the total heat convected to the fluid (W s, Btu).
twen	1	Effective error in the total heat convected to the fluid (W s, Btu).
twn	nodes* ncellt	Node-cell wall temperatures (K, °F) in the order: node 1 to NODES for cell 1, node 1 to NODES for cell 2, etc.
type	1	Component type (header variable only).
vln	ncellt+1	Interface liquid velocities (m s^{-1}, ft s^{-1}).
vol	ncellt	Cell volumes (m^3, ft^3) (header variable only).
vvn	ncellt+1	Interface gas velocities (m s^{-1}, ft s^{-1}).
wfl	ncellt+1	Interface friction factors (–).
x	ncellt	Cell upper bounds. (m, ft) (header variable only)(header variable only).

E.5. BREAK-Component Graphics. Subroutine `xtvbrak` outputs all graphics variables for the BREAK component.

Variable	Dimension	Description
alpn	1	BREAK-cell gas volume fraction (–).
bsa	1	Time-integrated, noncondensable-gas mass flow (kg, lb_m).
bsmass	1	Time-integrated mass flow (kg, lb_m) into the BREAK cell.
bxa	1	Noncondensable-gas mass flow (kg s^{-1}, lb_m h^{-1}).
bxmass	1	Mass flow (kg s^{-1}, lb_m h^{-1}) into the BREAK cell.
concn	1	BREAK-cell, dissolved-solute concentration ratio [kg(solute) kg^{-1}(liquid), lb_m(solute) lb_m-1(liquid)].
enth	1	BREAK-cell fluid enthalpy (W s kg^{-1}, Btu lb_m-1).
fa	2	BREAK-interface flow areas (m^2, ft^2) (header variable only).
id	1	Component ID number (header variable only).
ncellt	1	Total number of cells (should be 1) (header variable only).
pan	1	BREAK-cell, noncondensable-gas partial pressure (Pa, psia).
pn	1	BREAK-cell total pressure (Pa, psia).
tln	1	BREAK-cell liquid temperature (K, °F).
tvn	1	BREAK-cell gas temperature (K, °F).
type	1	Component type (header variable only).
vol	1	BREAK-cell volume (m^3, ft^3) (header variable only).
x	1	BREAK-cell upper bound (m, ft) (header variable only).

E.6. FILL-Component Graphics.

Subroutine `xtvfill` outputs all graphics variables for the FILL component.

Variable	Dimension	Description
`alpn`	1	FILL-cell gas volume fraction (–).
`concn`	1	FILL-cell, dissolved-solute concentration ratio [kg(solute) kg^{-1}(liquid), lb_m(solute) lb_m-1(liquid)].
`enth`	1	FILL-cell fluid enthalpy (W s kg^{-1}, Btu lb_m-1).
`fa`	2	FILL-interface flow areas (m^2, ft^2) (header variable only).
`fxmass`	1	Mass flow (kg s^{-1}, lb_m h^{-1}) out of the FILL cell.
`id`	1	Component ID number (header variable only).
`ncellt`	1	Total number of cells (should be 1) (header variable only).
`pan`	1	FILL-cell, noncondensable-gas partial pressure (Pa, psia).
`pn`	1	FILL-cell total pressure (Pa, psia).
`tln`	1	FILL-cell liquid temperature (K, °F).
`tvn`	1	FILL-cell gas temperature (K, °F).
`type`	1	Component type (header variable only).
`vln`	1	FILL-interface liquid velocity (m s^{-1}, ft s^{-1}).
`vol`	1	FILL-cell volume (m^3, ft^3) (header variable only).
`vvn`	1	FILL-interface gas velocity (m s^{-1}, ft s^{-1}).
`x`	1	FILL-cell upper bound (m, ft) (header variable only).

E.7. HTSTR (Heat-Structure)-Component ROD- or SLAB-Element Graphics.

Subroutine `xtvht` outputs all graphics variables for the HTSTR component ROD or SLAB elements.

Variable	Dimension	Description
`alreac`	1	Gas volume-fraction reactivity (–).
`cepwn`	2	Outer-surface and inner-surface heat-transfer difference (W, Btu h^{-1}).
`dbreac`	1	Dissolved- and plated-solute reactivity (–).
`hrfli`	`nzmax`	Liquid heat-transfer coefficient (W m^{-2} K^{-1}, Btu ft^{-2} $°F^{-1}$ h^{-1}) for the inner surface of the ROD or SLAB elements.
`hrflo`	`nzmax`	Liquid heat-transfer coefficient (W m^{-2} K^{-1}, Btu ft^{-2} $°F^{-1}$ h^{-1}) for the outer surface of the ROD or SLAB elements.
`hrfvi`	`nzmax`	Gas heat-transfer coefficient (W m^{-2} K^{-1}, Btu ft^{-2} $°F^{-1}$ h^{-1}) for the inner surface of the ROD or SLAB elements.
`hrfvo`	`nzmax`	Gas heat-transfer coefficient (W m^{-2} K^{-1}, Btu ft^{-2} $°F^{-1}$ h^{-1}) for the outer surface of the ROD or SLAB elements.
`id`	1	Component ID number (header variable only).
`ihtfi`	`nzmax`	Heat-transfer regime numbers for the inner surface of the ROD or SLAB elements.
`ihtfo`	`nzmax`	Heat-transfer regime numbers for the outer surface of the ROD or SLAB elements.

nodes	1	Number of ROD-radial or SLAB-thickness heat-transfer nodes (first level only).
nrods	1	Total number of ROD or SLAB elements evaluated by the HTSTR component (header variable only)
nzmax	1	Maximum number of rows of nodes in the axial direction of the HTSTR component (header variable only).
pgreac	1	Programmed reactivity (–).
powli	ncrz	Inner-surface heat transfer to the liquid (W, Btu h^{-1}).
powlo	ncrz	Outer-surface heat transfer to the liquid (W, Btu h^{-1}).
powvi	ncrz	Inner-surface heat transfer to the gas (W, Btu h^{-1}).
powvo	ncrz	Outer-surface heat transfer to the gas (W, Btu h^{-1}).
rftn	nodes* nzmax	ROD- or SLAB-element temperatures (K, °F), ordered node 1 to node NODES for row 1, node 1 to node NODES for row 2, etc.
rmckn	1	Reactor multiplication constant $K_{eff\ (-)}$.
rpower	1	Reactor power (W, Btu h^{-1}).
rzht	ncrz+1	Axial positions of the rows of nodes (m, ft).
stnui	nzmax	Inner-surface Stanton number (–) of the ROD or SLAB element.
stnuo	nzmax	Outer-surface Stanton number (–) of the ROD or SLAB element.
tcefni	1	Inner-surface total heat transfer to the fluid (W s, Btu).
tcefno	1	Outer-surface total heat transfer to the fluid (W s, Btu).
tcreac	1	Coolant-temperature reactivity (–).
tfreac	1	Fuel-temperature reactivity (–).
tldi	nzmaxz	Inner-surface liquid temperatures (K, °F) at bubble departure.
tldo	nzmaxz	Outer-surface liquid temperatures (K, °F) at bubble departure.
tpowi	1	Total power across the inner surface of the heat-structure component (W, Btu h^{-1}).
tpowo	1	Total power across the inner surface of the heat-structure component (W, Btu h^{-1}).
trhmax	1	Maximum temperature (K, °F) of the supplemental ROD or SLAB elements.
tramax	1	Maximum temperature (K, °F) of the average power ROD or SLAB elements.
twani	1	Inner-surface absolute error in the heat transfer to the fluid (W s, Btu).
twano	1	Outer-surface absolute error in the heat transfer to the fluid (W s, Btu).
tweni	1	Inner-surface effective error in the heat transfer to the fluid (W s, Btu).
tweno	1	Outer-surface effective error in the heat transfer to the fluid (W s, Btu).
type	1	Component type (header variable only).

zht	nzmax	Axial positions (m, ft) of the rows of nodes in the ROD or SLAB elements.

E.8. PIPE-Component Graphics.

In addition to a call to xtv1d, subroutine xtvpipe outputs graphics variables specific to the PIPE component.

Variable	Dimension	Description
cpow	1	Heater power (W, Btu h^{-1}) to the fluid.
qout	1	Liquid volume discharged (m^3, ft^3) at the exit (interface ncells+1) when the accumulator flag iacc > 0.
vflow	1	Volumetric fluid flow (m^3 s^{-1}, gpm) at the exit (interface ncells+1) when the accumulator flag iacc > 0.
z	1	Water level (m, ft) in the PIPE component (assumes the component is vertically oriented with cell 1 at the top) when the accumulator flag iacc > 0.

E.9. PLENUM-Component Graphics.

Subroutine xtvplen outputs all graphics variables specific to the PLENUM component.

Variable	Dimension	Description
alpn	1	Cell gas volume fraction (–).
am	1	Cell noncondensable-gas mass (kg, lb_m).
concn	1	Cell dissolved-solute concentration ratio [kg(solute) kg^{-1}(liquid), lb_m(solute) lb_m-1(liquid)].
dx	npljn	Cell lengths (m, ft) associated with each PLENUM-component junction (header variable only).
id	1	Component ID number (header variable only).
ncellt	1	Total number of cells (should be 1) (header variable only).
npljn	1	Number of junctions (header variable only).
pan	1	Cell noncondensable-gas partial pressure (Pa, psia).
pn	1	Cell total pressure (Pa, psia).
roan	1	Cell noncondensable-gas density (kg m^{-3}, lb_m ft^{-3}).
roln	1	Cell liquid density (kg m^{-3}, lb_m ft^{-3}).
rom	1	Cell mixture density (kg m^{-3}, lb_m ft^{-3}).
rovn	1	Cell gas density (kg m^{-3}, lb_m ft^{-3}).
sn	1	Cell plated-solute mass/fluid volume (kg m^{-3}, lb_m ft^{-3}).
tln	1	Cell liquid temperature (K, °F).
tsat	1	Cell saturation temperature (K, °F) based on the total pressure.
tvn	1	Cell gas temperature (K, °F).
type	1	Component type (header variable only).
vol	1	Cell volume (m^3, ft^3) (header variable only).

E.10. PRIZER (Pressurizer)-Component Graphics.

In addition to a call to `xtvld`, subroutine `xtvprzr` outputs graphics variables specific to the PRIZER component.

Variable	Dimension	Description
`flow`	1	Volumetric flow (m^3 s^{-1}, gpm) at the exit (interface `ncells+1`) of the PRIZER.
`qin`	1	Heater/sprayer power (W, Btu h^{-1}).
`qout`	1	Liquid volume discharged (m^3, ft^3) at the exit (interface `ncells+1`) of the PRIZER.
`z`	1	Water level (m, ft) in the PRIZER component (assumes the component is vertically oriented with cell 1 at the top).

E.11. PUMP-Component Graphics.

In addition to a call to `xtvld`, subroutine `xtvpump` outputs graphics variables specific to the PUMP component.

Variable	Dimension	Description
`alpha`	1	Gas volume fraction donored across the second (pump-impeller) interface (weighted 10% new, 90% old).
`delp`	1	PUMP ?P (Pa, psia) across the second (pump-impeller) interface (pressure of cell 2 minus pressure of cell 1).
`flow`	1	Volumetric fluid flow (m^3 s^{-1}, gpm) donored across the second (pump-impeller) interface.
`head`	1	PUMP head (Pa m^3 kg^{-1} or m^2 s^{-2} or N m kg^{-1}, lb_f ft lb_m^{-1}) from the homologous curves and two-phase degradation multiplier.
`mflow`	1	Fluid mass flow (kg s^{-1}, lb_m h^{-1}) across the second (pump-impeller) interface.
`omegan`	1	Pump-impeller rotational speed (rad s^{-1}, rpm).
`rho`	1	Fluid mixture density (kg m^{-3}, lb_m ft^{-3}) donored across the second (pump-impeller) interface.
`smom`	1	Momentum source (Pa, psia) applied at the second (pump-impeller) interface based on the PUMP head.
`torque`	1	PUMP hydraulic torque (Pa m^3, lb_f ft) from the homologous curves and two-phase degradation multiplier.

E.12. TEE-Component Graphics.

In addition to a call to `xtvld`, subroutine `xtvtee` outputs graphics variables specific to the TEE component.

Variable	Dimension	Description
`powr1`	1	Heater power (W, Btu h^{-1}) to the main-tube fluid.

powr2	1	Heater power (W, Btu h^{-1}) to the side-tube fluid.

E.13. VALVE-Component Graphics.

In addition to a call to xtvld, subroutine xtvvlve outputs graphics variables specific to the VALVE component.

Variable	Dimension	Description
area	1	Adjustable valve-interface flow area (m^2, ft^2).

E.14. 3D VESSEL-Component Graphics

Subroutine xtvvsl outputs graphics variables to the VESSEL component. The cell and interface data are written on a 3D basis in ROW MAJOR format, unlike TRCGRF, which used a level format. As with the 1D variables, interface variables have one more value than cell variables on the face axis. For example vlnz, the z-direction liquid velocity, has nrsx*ntsx*(nasx+1) values. The VESSEL variables output to graphics are very much dependent on the options selected and parameters set in the VESSEL input-data, NAMELIST, and other general options. The following abbreviations are used for dimensions in this section:

ncells = nrsx*ntsx*nasx (values at every cell)
xrfaces = (nrsx+1)*ntsx*nasx (values at each x/r face, including ic0m)
ytfaces = nrsx*(ntsx+1)*nasx (values at each y/θ face, including jc0m)
zfaces = nrsx*ntsx* (nasx+1) (values at each z face, including kc0m)

Variable	Dimension	Description
alpn	ncells	Cell gas volume fractions(-).
alven	ncells	Cell liquid-side interfacial heat-transfer coefficients (W K^{-1}, Btu $°F^{-1}$ h^{-1}) (area folded in).
alvn	ncells	Cell flashing interfacial heat-transfer coefficients (W K^{-1}, Btu $°F^{-1}$ h^{-1}) (area folded in).
am	ncells	Cell noncondensable-gas masses (kg, lb_m).
chtan	ncells	Cell noncondensable-gas interfacial heat-transfer coefficients (W K^{-1}, Btu $°F^{-1}$ h^{-1}) (area folded in).
chtin	ncells	Cell vapor-side interfacial heat-transfer coefficients (W K^{-1}, Btu $°F^{-1}$ h^{-1}) (area folded in).
cimfr	1	Reactor-core inlet mass flow (kg s^{-1}, lb_m h^{-1}).
cimfrl	1	Reactor-core inlet, liquid mass flow (kg s^{-1}, lb_m h^{-1}).
cimfrv	1	Reactor-core inlet, gas mass flow (kg s^{-1}, lb_m h^{-1}).
cixr	ncells	Radial or x-direction interfacial-drag coefficients (kg m^{-4}, lb_m ft^{-4}).
ciyt	ncells	Azimuthal or y-direction interfacial-drag coefficients (kg m^{-4}, lb_m ft^{-4}).
ciz	ncells	Axial interfacial-drag coefficients (kg m^{-4}, lb_m ft^{-4}).
comfr	1	Reactor-core region, outlet mass flow (kg s^{-1}, lb_m h^{-1}).

comfrl	1	Reactor-core outlet, liquid mass flow (kg s^{-1}, lb_m h^{-1}).
comfrv	1	Reactor-core outlet, gas mass flow (kg s^{-1}, lb_m h^{-1}).
concn	ncells	Cell dissolved-solute concentration ratio [kg(solute) kg^{-1}(liquid), lb_m(solute) lb_m-1(liquid)].
corelq	1	Reactor-core liquid volume fraction.
dcflow	1	Downcomer mass flow (kg s^{-1}, lb_m h^{-1}) (sums the axial flow out of the downcomer at level IDCL).
dclqvl	1	Downcomer liquid volume fraction.
faxr	xrfaces	Interface fluid flow areas (m^2, ft^2) (header variable only).
fayt	ytfaces	Interface fluid flow areas (m^2, ft^2) (header variable only).
faz	zfaces	Interface fluid flow areas (m^2, ft^2) (header variable only).
gamn	ncells	Vapor (steam) generation rate (kg m^{-3}, lb_m ft^{-3}).
hgam	ncells	Cell subcooled boiling heat flux (W m^{-2}, Btu ft^{-2} h^{-1}).
icj	ncsr	1D hydraulic component numbers connected to source-connection junctions (header variable only).
id	1	Component ID number(header variable only).
isrc	ncsr	Cell numbers to which source-connection junctions are connected (header variable only).
isrf	ncsr	Face code to which source-connection junctions are connected (header variable only).
isrl	ncsr	Level numbers to which source-connection junctions are connected (header variable only).
nasx	1	Number of axial levels (header variable only).
ncsr	1	Number of VESSEL source-connection junctions to 1D hydraulic components (header variable only).
nrsx	1	Number of radial rings or x-direction cells (header variable only).
nsrl	nasx	Number of source-connection junctions on each level (header variable only).
ntsx	1	Number of azimuthal segments or y-direction cells (header variable only).
pan	ncells	Cell noncondensable-gas partial pressures (Pa, psia).
pcore	1	Reactor-core, volume-averaged pressure (Pa, psia).
pdc	1	Downcomer volume-averaged total pressure (Pa, psia).
plp	1	Lower-plenum, volume-averaged total pressure (Pa, psia).
pn	ncells	Cell total pressures (Pa, psia).
pup	1	Upper-plenum, volume-averaged total pressure (Pa, psia).
qhstot	1	Total HTSTR-component heat transfer (W, Btu h^{-1}) to the fluid of the VESSEL component.
qsl	ncells	HTSTR-component heat transfer (W, Btu h^{-1}) to the fluid in each VESSEL cell.
r	nrsx	r upper bound (m, ft) of each radial ring or cell (header variable only).
roan	ncells	Cell noncondensable-gas densities (kg m^{-3}, lb_m ft^{-3}).
roln	ncells	Cell liquid densities (kg m^{-3}, lb_m ft^{-3}).
rom	ncells	Cell mixture densities (kg m^{-3}, lb_m ft^{-3}).

rovn	ncells	Cell gas densities (kg m^{-3}, lb_m ft^{-3}).
sn	ncells	Cell plated-solute mass/fluid volume (kg m^{-3}, lb_m ft^{-3}).
t	ntsx	θ upper bound (rad, deg) of each azimuthal segment or sector (header variable only).
tcilmf	1	Time-integrated reactor-core inlet, liquid mass flow (kg, lb_m).
tcivmf	1	Time integrated reactor-core inlet, gas mass flow (kg, lb_m).
tcolmf	1	Time integrated reactor-core outlet, liquid mass flow (kg, lb_m).
tcore	1	Reactor-core, mass-averaged liquid temperature (K, °F).
tcovmf	1	Time integrated reactor-core outlet gas mass flow (kg, lb_m).
tdc	1	Downcomer mass-averaged liquid temperature (K, °F).
tln	ncells	Cell liquid temperatures (K, °F).
tlp	1	Lower-plenum, mass-averaged liquid temperature (K, °F).
tsat	ncells	Cell saturation temperatures (K, °F) based on the total pressures.
tscore	1	Reactor-core average saturation temperature (K, °F) based on the reactor-core, volume-averaged total pressure.
tsdc	1	Downcomer average saturation temperature (K, °F) based on the downcomer volume-averaged total pressure.
tslp	1	Lower-plenum average saturation temperature (K, °F) based on the lower-plenum, volume-averaged total pressure.
tsup	1	Upper-plenum average saturation temperature (K, °F) based on the upper-plenum, volume-averaged total pressure.
tup	1	Upper-plenum mass-averaged liquid temperature (K, °F).
tvn	ncells	Cell gas temperatures (K, °F).
type	1	Component type (header variable only).
vcore	1	Reactor-core liquid mass (kg, lb_m).
vdclq	1	Downcomer liquid mass (kg, lb_m).
vlnxr	xrfaces	Liquid radial or x-direction velocities (m s^{-1}, ft s^{-1}).
vlnyt	ytfaces	Liquid azimuthal or y-direction velocities (m s^{-1}, ft s^{-1}).
vlnz	zfaces	Liquid axial velocities (m s^{-1}, ft s^{-1}).
vlpliq	1	Lower-plenum liquid volume fraction.
vlplm	1	Lower-plenum liquid mass (kg, lb_m).
vlplq	1	Liquid mass below downcomer (kg, lb_m).
vlqmss	1	VESSEL-component liquid mass (kg, lb_m).
vmfrl	ncells	Liquid mass flows (kg s^{-1}, lb_m h^{-1}) (NAMELIST variable imfr = 1].
vmfrlr	xrfaces	Liquid radial mass flows (kg s^{-1}, lb_m h^{-1}) (NAMELIST variable imfr = 3).
vmfrlt	ytfaces	Liquid azimuthal mass flows (kg s^{-1}, lb_m h^{-1}) (NAMELIST variable imfr = 3).
vmfrlz	zfaces	Liquid axial mass flows (kg s^{-1}, lb_m h^{-1}) (NAMELIST variable imfr = 3).

vmfrv	ncells	Gas mass flows (kg s^{-1}, lb_m h^{-1}) (NAMELIST variable imfr = 1).
vmfrvr	xrfaces	Gas radial mass flows (kg s^{-1}, lb_m h^{-1}) (NAMELIST variable imfr = 3).
vmfrvt	ytfaces	Gas azimuthal mass flows (kg s^{-1}, lb_m h^{-1}) (NAMELIST variable imfr = 3).
vmfrvz	zfaces	Gas axial mass flows (kg s^{-1}, lb_m h^{-1}) (NAMELIST variable imfr = 3).
vol	ncells	Cell fluid volumes (m^3, ft^3) (header variable only).
vsflow	1	Fluid mass flow (kg s^{-1}, lb_m h^{-1}) summed over all VESSEL-component source-connection junctions.
vupliq	1	Upper-plenum liquid volume fraction.
vuplm	1	Upper-plenum liquid mass (kg, lb_m).
vvnxr	xrfaces	Gas radial or x-direction velocities (m s^{-1}, ft s^{-1}).
vvnyt	ytfaces	Gas azimuthal or y-direction velocities (m s^{-1}, ft s^{-1}).
vvnz	zfaces	Gas axial velocities (m s^{-1}, ft s^{-1}).
x	nrsx	x upper bound (m, ft) of each x-direction cell (header variable only).
y	ntsx	y upper bound (m, ft) of each y-direction cell (header variable only).
z	nasx	z upper bound (m, ft) of each axial level or cell (header variable only).

APPENDIX F
DESCRIPTION OF TRAC-M BIT FLAGS

TRAC stores a variety of "yes/no" information for all the individual mesh cells of the 1D and 3D hydrodynamic component-types in the form of "bit flags." These bit flags are the individually addressed on/off (set/not set, 1 or 0) bit positions of the computer words in the 1D-component DUALPT arrays `bitn` and `bit`. Arrays `bitn` and `bit` are dimensioned `nfaces` (which is NCELLS + 1) by `TRACAllo`. The 3D VESSEL component also uses its own arrays `bitn` and `bit`.

The bit flags are accessed with TRAC C-language functions `btestc`, `ibsetc`, and `ibclrc`:

`btestc` -- return status of requested bit position

`ibsetc` -- set requested bit to "on" (1)

`ibclrc` -- set requested bit to "off" (0)

These TRAC C functions are named after the corresponding Fortran 90 intrinsic functions `btest`, `ibset`, and `ibclr`.

In addition, TRAC has two Fortran functions, `on1123` and `of1123`, to manage certain bit operations as a group as part of the logic for backups to the start of the `outer` stage (`on1123` clears all bits except a "protected" group; `of1123` clears a group of water-packer-logic bits). These TRAC Fortran functions drive the TRAC C-language bit functions.

TRAC (Version 3.0) currently uses 31 different bit flags (total for 1D and 3D hydrodynamic components), leaving one position available for future use in a 32-bit word. The bit positions for the TRAC C-language bit functions are accessed from TRAC with parameter variables that have meaningful names. The parameter values of the bit flags are assigned in module `Bits`, which also has documentation on the use of each bit flag.

Coding Standard : Any change to the bit flag parameterization should be done in module `Bits`. Any change to the bit flag logic should be documented in module `Bits`.

A listing of module `Bits` follows. After some general information, a detailed description of the use of each bit flag is given. TRAC-M inherited its bit flag logic from TRAC-P. In the development of TRAC-M, the original TRAC-P bit-flag positions were "remapped" for two reasons: to fit all of the bit positions into a single 32-bit word and to group the bits into those that carry information defined at mesh-cell centers and those that carry information defined at cell faces. This grouping will facilitate any future splitting of the `bitn` and `bit` arrays into cell-

center and cell-face arrays. Module `Bits` includes a description of the bit remapping. Use of bit 18 was added after the remapping.

```
                  MODULE Bits
!
!     BEGIN MODULE USE
      USE IntrType
!
!
!
! The array fbit, which holds unchanging geometric information for
! the 3D hydro, is not treated in this file.
!
! Note that TRAC now uses a bit-numbering convention from "right to
! left," starting with bit 1.  The F90 intrinsic bit manipulation
! routines (ibset, btest, etc.) also go from "right to left," but
! start with bit number 0.
!
! Note that the bitn arrays are cleared with 0. (floating point
! zero).  The Cray and the supported IEEE platforms all represent
! 0. as all-zeros.
!
! This version of header file bitflags.h re-maps the original flag id
! numbers (i.e., as used in the pre-branch code -- Version 5.4.25) into
! the range 1 - 32; it also groups all the flag numbers according to
! whether the bit is defined for a mesh-cell center or cell face.
!
! The current bit id number definitions and their original values are as
! follows:
!
! **************  ***************  ****************
! current number  original number  cell center/face
! **************  ***************  ****************
!
!        1                42              c
!        2                43              c
!        3                3               c
!        4                4               c
!        5                5               c
!        6                6               c
!        7                20              c
!        8                21              c
!        9                24              c
!        10               34              c
!        11               11              c
!        12               12              c
!        13               13              c
!        14               26              c
!        15               27              c
!        16               29              c
!        17               30              c
!        18                               c
!        19 ***not used***
!        20               18              f
!        21               19              f
```

```
!        22                22                f
!        23                23                f
!        24                10                f
!        25                25                f
!        26                44                f
!        27                45                f
!        28                28                f
!        29                46                f
!        30                33                f
!        31                31                f
!        32                32                f
!
! bits 33 and higher ***not used***
!
! *************
! *** bit 1 ***
!
! purpose: Used in interfacial heat-transfer logic to determine if the
!          vapor temperature crossed the saturation line since the
!          previous timestep.  Bit 1 is set on in the new-time (bitn)
!          array if vapor temperature tvn is greater than tssn (sat.
!          temp. at steam partial pressure).  If comparison with the
!          old-time (bit) array shows that the sat. line was crossed, the
!          relaxation-limiter logic (transient mode) on changes in chti
!          (vapor-side interfacial heat-transfer coefficient x
!          interfacial area) and chtia (non-condensable interfacial htc
!          x interfacial area) is bypassed.
!
!          Use is identical for 1D, 3D, and plenum.
!
! set in:  htif (outer stage - 1D, 3D, and plenum) -- bitn also cleared
!
! used in: htif (outer stage - 1D, 3D, and plenum)
!
      INTEGER(sik) satLineCrossVap
      PARAMETER (satLineCrossVap=1)
!
! *************
! *** bit 2 ***
!
! purpose: Bit 2 is the liquid analog of bit 1; its logic for
!          clearing, setting, and testing in routine htif is the same as
!          for bit 1, comparing liquid temperature tln to tssn.  If the
!          liquid temperature crossed the sat. line since the last timestep,
!          transient relaxation-limiter logic on alve (liquid-side
!          interfacial htc x interfacial area) and alv (flashing
!          interfacial htc x interfacial area) is bypassed.
!
!          Use is identical for 1D, 3D, and plenum (but see following
!          note).
!
!          Note that subroutine inner calls entry point on1123 in
!          subroutine bits.  on1123 clears all bits except 11, 12, 13,
!          32, and 2; intention is to clear all new-time bits (bitn
!          array) except water packer flags and bits set in prep stage,
```

```
!          for 1D components (plenum is excluded).  Protecting bit 2
!          is no longer needed, and in any case this logic is not
!          parallel with that for bit 1.  This does not appear to
!          cause an actual error in the calculation, but it should be
!          further investigated and at least cleaned up.
!
! set in:  htif (outer stage - 1D, 3D, and plenum) -- bitn also cleared
!
! used in: htif (outer stage - 1D, 3D, and plenum)
!
      INTEGER(sik) satLineCrossLiq
      PARAMETER (satLineCrossLiq=2)
!
! *************
! *** bit 3 ***
!
! purpose: Used in reiteration logic when void fraction is out of
!          bounds in basic (outer) step.  If void fraction exceeds
!          tolerance of 10(-12) (i.e., if .le. -1.0e-12 or .ge. (1.0+
!          1.0e-12)), bit 3 is set on and the logical reiteration flag
!          is set to .true..  If bit 3 has been set on on a previous
!          iteration, this test on void fraction is bypassed.
!
!          Usage identical in 1D, 3D, and plenum hydro.
!
! set in:  tf1ds3 (outer stage - 1D)
!          tf3ds3 (outer stage - 3D)
!          tfplbk (outer stage - plenum)
!
! used in: tf1ds3 (outer stage - 1D)
!          tf3ds3 (outer stage - 3D)
!          tfplbk (outer stage - plenum)
!
      INTEGER(sik) oneVoidFrReit
      PARAMETER (oneVoidFrReit=3)
!
! *************
! *** bit 4 ***
!
! purpose: Two distinct uses. In the initialization stage, bit 4 is set to
!          indicate that internally used FRICs have been calculated from
!          user-input K factors (this logic is part of the input-error
!          checking for consistency at component junctions).  During the
!          calculation, bit 4 is set to indicate mean mass equation
!          will be solved rather than vapor and liquid mass equations
!          (flow is single phase or nearly single phase).
!
!          The input-checking-use of bit 4 is for 1D components.
!          The hydro-use of bit 4 is similar in 1D, 3D, and plenum.
!
!          The parameter meanEqnSet is meant to be used only for the
!          hydro calculation, for 1D, 3D, and plenum.
!
! set in:  chbset (init stage)
!          tf1ds (outer stage - 1D)
```

```
!          tf3ds (outer stage - 3D)
!          tfpln (outer stage - plenum)
!
! used in: chkbd (init stage)
!          tf1ds (outer stage - 1D), tf1ds3 (outer stage - 1D)
!          tf3ds3 (outer stage - 3D)
!          tfpln (outer stage - plenum), tfplbk (outer stage - plenum)
!
      INTEGER(sik) meanEqnSet
      PARAMETER (meanEqnSet=4)
!
! *************
! *** bit 5 ***
!
! purpose: Used in calculation of interfacial heat and mass transfer in
!          basic (outer) step.  Bit 5 is set on in a hydro cell for
!          condensation conditions (negative gamma and void fraction
!          greater than zero; see following note on plenum).
!
!          Use is very similar in 1D, 3D, and plenum.  Plenum logic for
!          setting does _not_ have test on void fraction.
!
! set in:  tf1ds (outer stage - 1D)
!          tf3ds (outer stage - 3D)
!          tfpln (outer stage - plenum)
!
! used in: tf1ds (outer stage - 1D)
!          tf3ds (outer stage - 3D)
!          tfpln (outer stage - plenum)
!
      INTEGER(sik) condensing
      PARAMETER (condensing=5)
!
! *************
! *** bit 6 ***
!
! purpose: Evaporation/flashing analog of bit 5.  Bit 6 is set on if
!          gamma is positive and the void fraction is less than one.
!
!          Use is very similar in 1D, 3D, and plenum.  Plenum logic for
!          setting does have test on void fraction.
!
! set in:  tf1ds (outer stage - 1D)
!          tf3ds (outer stage - 3D)
!          tfpln (outer stage - plenum)
!
! used in: tf1ds (outer stage - 1D)
!          tf3ds (outer stage - 3D)
!          tfpln (outer stage - plenum)
!
      INTEGER(sik) evapOrFlashing
      PARAMETER (evapOrFlashing=6)
!
! *************
! *** bit 7 ***
```

```
!
! purpose: When bit 7 is on, the old-time/new-time weighting factor
!          for donor-cell quantities used in the 1D and plenum
!          mass and energy equations is set to 1.0.  This forces
!          the fluxes to 100% new-time weighting.  The explicit/
!          implicit weighting factor is local variable xvset, which is
!          also local array dalp, which is array rhs in the 1D and
!          plenum data.
!
!          Bit 7 is used in similar fashion by 1D and plenum; it is not
!          used by 3D for any purpose, including the 3D xvset logic.
!          Bit 7 is cleared in subroutine htif for all components,
!          but this has no effect on 3D.  htif is called only on the
!          first Newton iteration (oitno=1); once bit 7 is set for a
!          given series of iterations, it remains set.
!
! set in:  htif (outer stage - 1D, 3D, and plenum) -- bitn cleared
!          tf1ds (outer stage - 1D)
!          tfpln (outer stage - plenum)
!
! used in: tf1ds (outer stage - 1D)
!          tfpln (outer stage - plenum)
!
!
      INTEGER(sik) freezeXvset
      PARAMETER (freezeXvset=7)
!
! *************
! *** bit 8 ***
!
! purpose: Used in equation set logic in basic (outer) stage.
!          Set in the back-substitution routines for use in
!          subsequent iterations for a given timestep for basic
!          energy equation.  Bit 8 is set on for situation of
!          almost, but not quite, solid water in a cell (very small
!          bubbles). The old-time void fraction must be .le.
!          1.0e-8, and the new-time void fraction must be .lt. -1.0e-12.
!          When on, bit 8 forces the vapor temperature to equal the
!          saturation temperature corresponding to the partial pressure
!          of steam.
!
!          The void fraction test for setting bit 8 has been
!          modified by update fixb21 (bit 8 was bit 21 in Version
!          5.4.25).
!
!          Use is the same in 1D, 3D, and plenum.
!
! set in:  tf1ds3 (outer stage - 1D)
!          tf3ds3 (outer stage - 3D)
!          tfplbk (outer stage - plenum)
!
!
! used in: tf1ds (outer stage - 1D)
!          tf3ds (outer stage - 3D)
!          tfpln (outer stage - plenum)
```

```
!
      INTEGER(sik) tinyBubbles
      PARAMETER (tinyBubbles=8)
!
! *************
! *** bit 9 ***
!
! purpose: Set in basic (outer) step when special logic is used to
!          change the current guess for the new-time value of the void
!          fraction before linearization.  If bit 9 is set (from a
!          previous iteration), the special void fraction logic
!          is bypassed (i.e., the bit is used to allow only one use of
!          this logic in a given series of Newton iterations).
!
!          1D, 3D, and plenum logic the same (1D and plenum use old- and
!          new-time bits 20 and 21 for velocity-reversal information;
!          3D uses old/new time donor-cell factor arrays (owlz, wlz,
!          etc.) for same purpose.
!
! set in:  tf1ds (outer stage - 1D)
!          tf3ds (outer stage - 3D)
!          tfpln (outer stage - plenum)
!
! used in: tf1ds (outer stage - 1D)
!          tf3ds (outer stage - 3D)
!          tfpln (outer stage - plenum)
!
      INTEGER(sik) triedVoidFrReset
      PARAMETER (triedVoidFrReset=9)
!
! **************
! *** bit 10 ***
!
! purpose: Used in 3D hydro only (there is identical logic in the 1D
!          that does not use a bit flag).  Bit 10 is set on for a mesh
!          cell when the net noncondensable ("air") flow into the cell
!          is .gt. 1.0e-20 kg for the current timestep.
!          Bit 10 subsequently is used in the same step in the logic to
!          set an initial guess for the air partial pressure.  If bit
!          10 is not on the initial guess is bypassed (there are other
!          tests that also can bypass the air logic).
!          The initial air partial pressure guess is the total
!          pressure minus the saturation pressure corresponding to
!          the current liquid temperature.
!
!          The 1D logic that corresponds to that for bit 10 is in
!          subroutine tf1ds3, at statement label 1337 in the pre-
!          branch code (Version 5.4.25); the air flow is in array dr.
!
! set in:  tf3ds (outer stage - 3D)
!
! used in: tf3ds3 (outer stage - 3D)
!
      INTEGER(sik) netAirFlow
      PARAMETER (netAirFlow=10)
```

```
!
! **************
! *** bit 11 ***
!
! purpose: Used with bits 12 and 13 in 1D water packing/stretch logic.
!          Used with bit 13 in 3D water pack/stretch logic.  Not used
!          by plenum.
!
!         Water packing and stretching are checked for determined?in each 1D and
!           3D cell at the start of the back-substitution
!           routines of the outer stage (tf1ds3 and tf3ds3
!           for 1D and 3D, respectively).  If water packing
!           is detected, the back substitution
!           is skipped and backup to the start of outer is forced.
!           In the 1D bit, 11 is set on for packing or stretching at a
!           cell's left face (bit 12 is used for the right face); in the
!           case of a stretch, bit 13 is also set on.  In the 3D, bit 11
!           indicates packing and bit 13 indicates stretch for the cell
!           (the stretch information is passed to the bd array by
!           routine j3d).
!
!           Note that subroutine inner calls entry point on1123 in
!           subroutine bits to clear all 1D bits _except_ 11, 12, 13, 32,
!           and 2 (see additional notes on bit 2).  Subroutine poster
!           call bits entry point of1123 to _clear_ 1D 11, 12, and 13 if
!           water packing flag ipakon .ne. 0 (bit and bitn arrays).
!           j3d (vessel source junction boundary array routine) also
!           calls of1123 for bd(53).
!
!           Parameter packAtLeftFace is intended for 1D use.
!           Parameter pack3D is intended for 3D use.
!
! set in:   j3d -- bd(53) only
!           tf1ds3 (outer stage - 1D)
!           tf3ds3 (outer stage - 3D)
!           poster (post stage - 1D) -- bitn and bit cleared if ipakon
!                                       .ne. 0
!
! used in: tf1ds (outer stage - 1D)
!           tf1ds1 (outer stage - 1D)
!           tf1ds3 (outer stage - 1D)
!           tf3ds1 (outer stage - 3D) -- bit 13 not used
!           tf3ds3 (outer stage - 3D) -- bit 13 not used
!
      INTEGER(sik) packAtLeftFace
      INTEGER(sik) pack3D
      PARAMETER (packAtLeftFace=11)
      PARAMETER (pack3D=11)
!
! **************
! *** bit 12 ***
!
! purpose: Used with bits 11 and 13 in 1D water packing/stretch logic.
!          Not used by 3D or plenum.  Indicates pack or stretch
!          detected at 1D cell's right face.  See additional notes
```

```
!          under bit 11.
!
!          See bit 11 on use of entry points on1123 and of1123 in bits.
!
! set in:  j3d -- bd(53) only
!          tf1ds3 (outer stage - 1D)
!          poster (post stage - 1D) -- bitn and bit cleared if ipakon
!                                      .ne. 0
!
! used in: tf1ds (outer stage - 1D)
!          tf1ds1 (outer stage - 1D)
!          tf1ds3 (outer stage - 1D)
!
      INTEGER(sik) packAtRightFace
      PARAMETER (packAtRightFace=12)
!
! **************
! *** bit 13 ***
!
! purpose: Used with bits 11 and 12 in 1D water packing/stretch logic.
!          Used with bit 11 in 3D water pack/stretch logic.  Not used
!          by plenum.
!
!          See bit 11 on use of entry points on1123 and of1123 in bits.
!
!          Parameter stretch is intended for 1D use.
!          Parameter stretch3D is intended for 3D use (this is passed
!          to the bd array by routine j3d).
!
! set in:  j3d -- bd(53) only
!          tf1ds3 (outer stage - 1D)
!          tf3ds3 (outer stage - 3D)
!          poster (post stage - 1D) -- bitn and bit cleared if ipakon
!                                      .ne. 0
!
! used in: tf1ds1 (outer stage - 1D)
!          tf1ds3 (outer stage - 1D)
!
      INTEGER(sik) stretch
      INTEGER(sik) stretch3D
      PARAMETER (stretch=13)
      PARAMETER (stretch3D=13)
!
! **************
! *** bit 14 ***
!
! purpose: Used in timestep-size control logic, in conjunction with
!          bit 15.  Bits 14 and 15, used with the void fraction arrays
!          alpn, alp, and alpo, save void fraction change behavior
!          looking back over three steps.  Bits 14 and 15
!          control calculation of variables oau and oal (in common
!          block chgalp), which are used in subroutine newdlt to
!          determine timestep size at start of next step.  oau is the
!          largest increase in void fraction in the system immediately
!          after a decrease, which in turn had followed an increase
```

```
!          (all for a given hydro cell). oal measures the analogous
!          situation for a decrease in void fraction.  Bit 14 is set on
!          for a hydro cell when void fraction has increased in that
!          cell w.r.t. previous timestep, in the bitn array.
!
!          Use of bit 14 is identical in 1D, 3D, and plenum.
!
!          Note that blkdat now sets variables xoau and xoal (common
!          block chgalp) to 1.0, which effectively turns off the
!          oscillating-void-fraction (oau or oal) timestep-size control.
!          Void-fraction-change timestep-size control now only uses
!          variables dau and dal, which only look back to the
!          previous step.  The dau/dal logic needs only arrays alpn
!          and alp, and not bits 14 and 15.
!
! set in:  poster (post stage - 1D)
!          bkstb3 (post stage - 3D)
!          plen3 (post stage - plenum)
!
! used in: poster (post stage - 1D)
!          bkstb3 (post stage - 3D)
!          plen3 (post stage - plenum)
!
      INTEGER(sik) newVoidFrUp
      PARAMETER (newVoidFrUp=14)
!
! **************
! *** bit 15 ***
!
! purpose: Used in conjunction with bit 14 for oscillating-void
!          fraction timestep-size control.  Bit 15 is set on (bitn
!          array) for a hydro cell when old-time bit 14 (bit array)
!          is on (i.e., when void fraction had increased during
!          previous timestep).  Bit 15 is saved in the old-time
!          bit array for use in the oau/oal logic.
!
!          Use of bit 15 is identical in 1D, 3D, and plenum.
!
!          Same note applies concerning variables xoau and xoal
!          as for bit 14.
!
! set in:  poster (post stage - 1D)
!          bkstb3 (post stage - 3D)
!          plen3 (post stage - plenum)
!
! used in: poster (post stage - 1D)
!          bkstb3 (post stage - 3D)
!          plen3 (post stage - plenum)
!
      INTEGER(sik) oldVoidFrUp
      PARAMETER (oldVoidFrUp=15)
!
! **************
! *** bit 16 ***
!
```

```
! purpose: Set on for a cell when net mass flow into cell is negative.
!          When bit 16 is on, the water pack/stretch logic in the back-
!          substitution routines is bypassed.
!
!          Use is same in 1D, 3D, and plenum.
!
! set in:  tf1ds (outer stage - 1D) -- always cleared before logic for
!                                      setting
!          tf3ds (outer stage - 3D)
!          tfpln (outer stage - plenum)
!
! used in: tf1ds3 (outer stage - 1D)
!          tf3ds3 (outer stage - 3D)
!          tfplbk (outer stage - plenum)
!
      INTEGER(sik) netMassOut
      PARAMETER (netMassOut=16)
!
! **************
! *** bit 17 ***
!
! purpose: Used in equation set logic.  The back-substitution routines
!          have logic to force the void fraction to 1.0 or 0.0 if bit 4
!          (for one of the single-phase mass equation sets) is on.  If
!          bit 17 also is on, the forcing to 0.0 is bypassed.  Instead,
!          equation to set steam pressure to saturation pressure
!          corresponding to the liquid temperature has been used.
!
!          Use is same in 1D, 3D, and plenum.
!
! set in:  tf1ds (outer stage - 1D)
!          tf3ds (outer stage - 3D)
!          tfpln (outer stage - plenum)
!
! used in: tf1ds3 (outer stage - 1D)
!          tf3ds3 (outer stage - 3D)
!          tfplbk (outer stage - plenum)
!
      INTEGER(sik) specEqnSteamP
      PARAMETER (specEqnSteamP=17)
!
! **************
! *** bit 18 ***
!
! purpose: Used to force the variable xvset to zero in the semi-implicit
!          mass and energy equations.  This is necessary for proper
!          functioning of various separation models that flux void
!          and/or liquid fractions out of a cell that are not equal to
!          the mean cell quantities
!
!          Currently Used only in the 1D
!
! set in:  tee2  (outer stage - 1D)
!
! used in: tf1ds  (outer stage - 1D)
```

```
!
      INTEGER(sik) noXvset
      PARAMETER (noXvset=18)
!
! **************
! *** bit 19 *** not used
!
! **************
! *** bit 20 ***
!
! purpose: 1D and plenum hydro only (including break and fill
!          components).  Cell-face flag to indicate vapor velocity
!          direction; used in logic for vapor donor cell weighting
!          factors and vapor velocity reversal.
!          The vapor velocity reversal information is used with
!          corresponding liquid information in the reiteration
!          logic (see bits 22 and 23), and alone in the
!          interfacial shear logic (see bit 26).  Vapor velocity
!          reversal information also is used in logic for special
!          void fraction guess (see bit 9).
!
!          Bit 20 is set on when vapor velocity is negative at
!          corresponding face.
!
! set in:  tf1ds1 (outer stage - 1D)
!          tf1ds3 (outer stage - 1D)
!
! used in: break1 (prep stage) -- bd(38) only
!          fill1 (prep stage) -- bd(38) only
!          flux (prep stage - 1D)
!          auxpln (outer stage - plenum) - bd(53) only
!          tf1ds (outer stage - 1D)
!          tf1ds1 (outer stage - 1D) -- bd(53) only
!          tf1ds3 (outer stage - 1D)
!          tfpln (outer stage - plenum) -- bd(38) and bd(53) only
!          poster (post stage - 1D) -- bit 21 not used
!          stbme (post stage - 1D)
!          tee3 (post stage - 1D)
!
      INTEGER(sik) negVapVel
      PARAMETER (negVapVel=20)
!
! **************
! *** bit 21 ***
!
! purpose: Liquid analog of bit 20.
!          1D and plenum hydro only (including break and fill
!          components).  Cell-face flag to indicate liquid velocity
!          direction; used in logic for liquid donor cell weighting
!          factors and liquid velocity reversal.
!          The liquid velocity reversal information is used with
!          corresponding vapor information in the reiteration
!          logic (see bits 22 and 23); there is no corresponding use in
!          the vapor-direction interfacial shear logic (see bit 26).
!          Liquid velocity reversal information also is used in logic
```

```
!          for special void fraction guess (see bit 9).
!
!          Bit 21 is set on when liquid velocity is negative at
!          corresponding face.
!
! set in:  tf1ds1 (outer stage - 1D)
!          tf1ds3 (outer stage - 1D)
!
! used in: break1 (prep stage) -- bd(38) only
!          fill1 (prep stage) -- bd(38) only
!          flux (prep stage - 1D)
!          auxpln (outer stage - plenum) - bd(53) only
!          tf1ds (outer stage - 1D)
!          tf1ds1 (outer stage - 1D) -- bd(53) only
!          tf1ds3 (outer stage - 1D)
!          tfpln (outer stage - plenum) -- bd(38) and bd(53) only
!          stbme (post stage - 1D)
!          tee3 (post stage - 1D)
!
      INTEGER(sik) negLiqVel
      PARAMETER (negLiqVel=21)
!
! **************
! *** bit 22 ***
!
! purpose: Used in logic that determines if a reiteration is forced
!          by a flow reversal.  Bit 22 is set on if the vapor mass-flow
!          threshold for a flow reversal reiteration is exceeded.
!          This threshold is set by variable frev (common block xvol).
!
!          Used in similar fashion for 1D and 3D; not used by plenum.
!          In 3D bit 22 is for radial (or x) face [bits 24 and 25 are
!          used for same purpose for axial and theta (or y) faces].
!          For 1D, new-time bit 20 is first used to check for a vapor
!          flow reversal; then bit 22 is used to see if the
!          vapor mass flow sensitivity level has been exceeded.
!
!          Parameter significantVapFlow is intended for 1D use.
!          Parameter significantVapFlowxr is intended for 3D use.
!
! set in:  tf1ds (outer stage - 1D)
!          tf3ds (outer stage - 3D)
!
! used in: tf1ds3 (outer stage - 1D)
!          tf3ds3 (outer stage - 3D)
!
      INTEGER(sik) significantVapFlow
      INTEGER(sik) significantVapFlowxr
      PARAMETER (significantVapFlow=22)
      PARAMETER (significantVapFlowxr=22)
!
! **************
! *** bit 23 ***
!
! purpose: Liquid analog of bit 22 (similar 3D use for bits 31 and 28).
```

```
!          Used in logic that determines if a reiteration is forced
!          by a flow reversal.  Bit 23 is set on if the liquid mass-flow
!          threshold for a flow reversal reiteration is exceeded.
!          This threshold is set by variable frev (common block xvol).
!
!          Used in similar fashion for 1D and 3D; not used by plenum.
!          In 3D, bit 23 is for radial (or x) face [bits 31 and 28 are
!          used for same purpose for axial and theta (or y) faces].
!          For 1D, new-time bit 21 is first used to check for a liquid
!          flow reversal; then bit 23 is used to see if the
!          liquid mass flow sensitivity level has been exceeded.
!
!          Parameter significantLiqFlow is intended for 1D use.
!          Parameter significantLiqFlowxr is intended for 3D use.
!
! set in:  tf1ds (outer stage - 1D)
!          tf3ds (outer stage - 3D)
!
! used in: tf1ds3 (outer stage - 1D)
!          tf3ds3 (outer stage - 3D)
!
      INTEGER(sik) significantLiqFlow
      INTEGER(sik) significantLiqFlowxr
      PARAMETER (significantLiqFlow=23)
      PARAMETER (significantLiqFlowxr=23)
!
! **************
! *** bit 24 ***
!
! purpose: 3D hydro only; same use as bit 22, for axial face.
!          Uses variable frev for vapor flow threshold.
!
! set in:  tf3ds (outer stage - 3D)
!
! used in: tf3ds3 (outer stage - 3D)
!
      INTEGER(sik) significantVapFlowz
      PARAMETER (significantVapFlowz=24)
!
! **************
! *** bit 25 ***
!
! purpose: 3D hydro only; same use as bit 22, for theta (or y) face.
!          Uses variable frev for vapor flow threshold.
!
! set in:  tf3ds (outer stage - 3D)
!
! used in: tf3ds3 (outer stage - 3D)
!
      INTEGER(sik) significantVapFlowyt
      PARAMETER (significantVapFlowyt=25)
!
! **************
! *** bit 26 ***
!
```

```
! purpose: Set in post stage to indicate vapor velocity has changed
!          direction during timestep being completed.  Used in prep
!          stage of subsequent timestep in calculation of interfacial
!          shear coefficients.  If bit is on, relaxation-limiter logic
!          for interfacial shear coefficient (used for transient mode)
!          is turned off.
!
!          Used in similar fashion for 1D and 3D; not used by plenum.
!          In 3D, bit 26 is for theta (or y) face [bits 27 and 29 are
!          used for same purpose for axial and radial (or x) faces].
!          1D sets new-time bit 26 according to status of old-time
!          bit 20 and new-time vapor velocity; 3D sets bit 26 according
!          to status of old-time and new-time donor-cell factors
!          for vapor at theta (or y) face (arrays owvyt and wvyt).
!
!          Parameter changeVapVel is intended for 1D use.
!          Parameter changeVapVelyt is intended for 3D use.
!
! set in:  poster (post stage - 1D)
!          ff3d (post stage - 3D)
!
! used in: femom (prep stage - 1D) (Note: StbVel1D does this in Version 3.0.)
!          cif3 (prep stage - 3D)
!
      INTEGER(sik) changeVapVel
      INTEGER(sik) changeVapVelyt
      PARAMETER (changeVapVel=26)
      PARAMETER (changeVapVelyt=26)
!
! **************
! *** bit 27 ***
!
! purpose: 3D hydro only; same use as bit 26, for axial face.  Set
!          according to status of arrays owvz and wvz.
!
! set in:  ff3d (post stage - 3D)
!
! used in: cif3 (prep stage - 3D)
!
      INTEGER(sik) changeVapVelz
      PARAMETER (changeVapVelz=27)
!
! **************
! *** bit 28 ***
!
! purpose: 3D hydro only; same use as bit 23, for theta (or y) face.
!          Uses variable frev for liquid flow threshold.
!
! set in:  tf3ds (outer stage - 3D)
!
! used in: tf3ds3 (outer stage - 3D)
!
      INTEGER(sik) significantLiqFlowyt
      PARAMETER (significantLiqFlowyt=28)
!
```

```
! **************
! *** bit 29 ***
!
! purpose: 3D hydro only; same use as bit 26, for radial (or x) face.
!          Set according to status of arrays owvxr and wvxr.
!
! set in:  ff3d (post stage - 3D)
!
! used in: cif3 (prep stage - 3D)
!
      INTEGER(sik) changeVapVelxr
      PARAMETER (changeVapVelxr=29)
!
! **************
! *** bit 30 ***
!
! purpose: Flag for choked-flow model.  Bit 30 is set on for a cell edge
!          if subroutine choke determines choked flow exists (the model
!          itself is invoked by user-input).  1D only.
!
!          The logic used to set bits 11 and 12 for a water
!          pack condition in the 1D is bypassed if bit 30 indicates
!          choking at the left or right cell face in question.
!
!          If bit 30 is on, subroutine ecomp prints -1.111e-11 for the
!          liquid wall friction for 1D components.
!
! set in:  tf1ds1 (outer stage - 1D) -- calls choke
!
! used in: ecomp (large edits for 1D)
!          femom (prep stage - 1D) -- logic not used
                (Note: StbVel1D does this in Version 3.0; also not  actually
                       using the logic.)
!          tf1ds1 (outer stage - 1D) -- bd(53) only, used to set same
!                                       bit
!          tf1ds3 (outer stage - 1D)
!
      INTEGER(sik) chokedFlowOn
      PARAMETER (chokedFlowOn=30)
!
! **************
! *** bit 31 ***
!
! purpose: 3D hydro only; same use as bit 23, for axial face.
!          Uses variable frev for liquid flow threshold.
!
! set in:  tf3ds (outer stage - 3D)
!
! used in: tf3ds3 (outer stage - 3D)
!
      INTEGER(sik) significantLiqFlowz
      PARAMETER (significantLiqFlowz=31)
!
! **************
! *** bit 32 ***
```

```
!
! purpose: Used to control choked-flow model when namelist variable
!          icflow is 2 (which invokes user-control of model at all 1D
!          cell faces).  Bit 32 is set in input stage for a face when
!          icflow .eq. 2 and component input array variable icflg is
!          nonzero (model is on for face).  1D only.
!
!          If icflow .eq. 2 and bit 32 is not set, the call to choke
!          in tf1ds1 is bypassed (see bit 30).
!
!          Bit 32 also is used by subroutine chkbd as an input check
!          on the consistency of choked-flow-option array icflg at
!          component junctions.
!
!          Bit 32 is one of the bits "protected" by entry point on1123
!          in subroutine bits (see notes on bit 11).
!
! set in:  rcomp (input for 1D)
!          preper (prep stage - 1D) -- after bitn cleared, bit 32 reset
!                                      if old-time bit 32 was on
!
! used in: chkbd (boundary array consistency check)
!          preper (prep stage - 1D) -- only to reset new-time bit 32
!          tf1ds1 (outer stage - 1D)
!
      INTEGER(sik) userChokeControl
      PARAMETER (userChokeControl=32)
!
! *************************
! *** bits 33 and higher *** not used
!
!
      END MODULE Bits
```

APPENDIX G
ADDING NEW VARIABLES TO TRAC

This appendix includes "coding standards" and "coding requirements." A standard promotes maintainability and extensibility. A requirement indicates that the code will not function properly if it is not followed.

Numerous coding examples in this appendix show the addition of a new variable to TRAC; sample additions that would be entered (or existing coding that would be modified) by a TRAC developer are shown in bold.

Note: XTV/XMGR5 Graphics System. TRAC-M/F90 Version 3.0 uses a now-obsolete version of the XTV/XMGR5 graphics system, which is implemented by Fortran module `Xtv`. Module `Xtv` is to be replaced in a future version of TRAC-M/F90 by modules `CXtvXFaces`, `XtvComps`, `XtvData`, `XtvDump`, and `XtvSetup`. The new implementation of the XTV/XMGR5 logic will include many arrays and derived types, all of which will be defined in module `XtvData`.

G.1. New Component Variables

G.1.1. Summary

Component Data-Type `genTabT` (FLT):

1. Modify the definition of data-type `genTabT` and the parameterization of the length of data-type `genTabT`.

2. Add the new variable to the dump/restart file.

3. Read the new variable from the dump/restart file.

4. Modify subroutine `GetGenTable` (as needed).

5. Echo new input variable (as needed).

6. Add to edits (as needed).

Component Data-Type "comp_type"`TabT` (VLTs):

1. Modify the definition of data type "comp_type"`TabT` and the parameterization of the length of data-type "comp_type"`TabT`.

2. Add the new variable to the dump/restart file.

3. Read the new variable from the dump/restart file.

4. Add or modify subroutine Get"Comp_type"Tab (as needed).

5. Add or modify subroutine Set"Comp_type"Tab (as needed).

6. Echo new input variable (as needed).

7. Add to edits (as needed).

Component Arrays:

For "comp_type"-specific arrays (1D components):

1. Add declaration of array to type "comp_type"ArrayT in module "Comp_type"zrray.

2. Add allocation of storage for array with call to subroutine TRACAllo in "comp_type" input routines, which are in module "Comp_type".

3. Add array to dump file with call to subroutine bfoutn in "comp_type" dump routine in module "Comp_type".

4. Read array from input file tracin with call to subroutine loadn, and echo to output file trcout with call to subroutine warray using "comp_type" input routine in module "Comp_type" after storage allocation.

5. Read array from restart file trcrst with call to subroutine bfinn and echo (restart) array to trcout with call to subroutine warray, using "comp_type" restart routine in module "Comp_type".

6. Write array to large (major) edits in trcout (as needed).

For General Data Arrays (1D components):

1. Add declaration of array to type g1DArrayT in module Gen1DArray.

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

3. Add array to dump file with call to subroutine bfoutn in subroutine dcomp.

4. Read array from input file tracin with call to subroutine loadn, and echo to output file trcout with call to subroutine warray, using subroutine rcomp after storage allocation.

5. Read array from restart file trcrst with call to subroutine bfinn, and echo (restart) array to trcout with call to subroutine warray, using subroutine recomp.

6. Write array to large (major) edits in `trcout` (as needed) with call to subroutine `wcomp`.

7. If the array stores old- or new-time values of a variable, add assignment statements for it to subroutine `TimeUpGen1D` (module `Gen1DArray`) (in two places).

8. If appropriate, add an assignment statement for the array to subroutine `BackUpGen1D` (module `Gen1DArray`).

9. On an as-needed basis, add a new index variable for the array to the module `Gen1DArray` data interface, and add a corresponding array reference to the case construct in subroutine `Get1DArrayPointer` (module `Gen1DArray`).

For the 3D VESSEL component, the procedure to add an array is similar. The storage arrays and service routines are described in Section H.1.4.2.

System Services

If the new variable is needed for intercomponent communication that is supported by the System Services, the System Services should be modified.

G.1.2. Adding a New Variable To Data-Type `genTabT` (the component FLT)

Each TRAC component type has a set of data that is "global" for a given component and where the variables are the same for all component types (mostly scalar variables) that are stored in elements of derived data-type `genTabT` (this is the TRAC-P FLT). Examples of such data are the component type and the user-assigned component number. Data for a given individual component are stored in array `genTab`, which is of derived-type `genTabT` and dimension `maxComps`.

All `genTabT`/`genTab`-related logic is handled by

module `Flt`.

The logic in module `Flt` comprises

- a definition of derived data-type `genTabT` (declaration of its elements);
- a declaration of array `genTab`;
- parameterization of the total length of data-type `genTabT` (for dump/restart);
- subroutine `GenTableDump` to add an individual component's variables that are stored in array `genTab` to the dump/restart file;

- subroutine GenTableRst to read a component's genTab data from the dump/restart file; and

- subroutine GetGenTable to access certain genTab data of a component other than the current instantiated component (e.g., data needed by a heat structure from a coupled hydro component).

The following code fragment gives an overview of the areas in module Flt that are affected when a new variable is added to genTabT. After this code fragment, specific details are given.

Fragment of current coding in module Flt (file FltM.f):

```
      MODULE Flt
                  ---
                  ---
                  ---
      TYPE genTabT
        REAL(sdk) htlsci
        REAL(sdk) htlsco
        REAL(sdk) pinteg
        REAL(sdk) title(4)
        INTEGER(sik) icflg
        INTEGER(sik) id
        INTEGER(sik) irest
        INTEGER(sik) typeIndex
        INTEGER(sik) lenvlt
        INTEGER(sik) lextra
        INTEGER(sik) ncellt
        INTEGER(sik) numbm1
        INTEGER(sik) numbm2
        INTEGER(sik) numbm3
        INTEGER(sik) numbn1
        INTEGER(sik) numbn2
        INTEGER(sik) numbn3
        INTEGER(sik) nodes
        INTEGER(sik) num
        REAL(sdk) type
      END TYPE genTabT
!
      TYPE(genTabT),DIMENSION(maxComps) :: genTab
                  ---
                  ---
                  ---
      INTEGER(sik) genDumpSize
       PARAMETER (genDumpSize=23)  <<<--- parameter genDumpSize
!
      CONTAINS
!
!
      SUBROUTINE GenTableDump(compInd,reordered)
                  ---
```

```
                  ---
                  ---
SUBROUTINE GenTableRst(compInd)
                  ---
                  ---
                  ---
SUBROUTINE GetGenTable(name,compInd,ival,rval,reordered)
                  ---
                  ---
                  ---
```

End fragment of current coding

To add real variable xxxxx or integer variable iiiii to genTabT, the following steps must be taken:

1. Modify the definition of data type genTabT and the parameterization of the length of data-type genTabT:

When adding real variable xxxxx to genTabT:

1a. add its declaration to the elements of type genTabT and

1b. increase the size of parameter-variable genDumpSize as appropriate:

```
TYPE genTabT
---
---
---
   REAL(sdk) type
   REAL(sdk) xxxxx
END TYPE genTabT
---
---
---
INTEGER(sik) genDumpSize
PARAMETER (genDumpSize=n)
---
---
---
```

where n = the total number of words in type genTabT (reals + integers), counting individual array elements separately (e.g., four words for genTabT variable-array title). In Version 2.119, genDumpSize = 23. If variable xxxxx is to be an array, specify its dimension (e.g., five words) within its declaration in genTabT:

```
REAL(sdk) xxxxx(5)
```

Follow an analogous procedure when adding integer variable iiiii to genTabT:

```
TYPE genTabT
---
---
---
 REAL(sdk) type
  INTEGER(sik) iiiii
END TYPE genTabT
---
---
---
INTEGER(sik) genDumpSize
PARAMETER (genDumpSize=n)
---
---
---
```

where again, n = the total number of words in data-type genTabT.

Coding Standard: The existing ordering of variables in data-type genTabT is not significant. However, as described below, the order in which genTabT elements are added to the dump file must match the order in which they are read from the restart file. This ordering mostly follows the declarations in genTabT (data element num is dumped first). New declarations in genTabT should be appended to the end of the current declarations.

2. Add the new variable to the dump/restart file:

Fragment of current coding in module Flt (file FltM.f):

```
                SUBROUTINE GenTableDump(compInd,reordered)
                ---
                ---
                ---
      CALL bfoutis(genTab(ordInd)%num,1,ictrld)
      CALL bfouts(genTab(ordInd)%htlsci,1,ictrld)
                ---
                ---
                ---
      CALL bfoutn(genTab(ordInd)%title,4,ictrld)
                ---
                ---
                ---
      CALL bfoutis(genTab(ordInd)%nodes,1,ictrld)
      CALL bfouts(genTab(ordInd)%type,1,ictrld)
!
    RETURN
```

```
      END SUBROUTINE GenTableDump
```

End fragment of current coding

Subroutine GenTableDump calls subroutines bfoutis, bfouts, and bfoutn, which are in module Restart. They are the standard service routines for dumping integer scalar, real scalar, and real array data, respectively; module Restart also has service routine subroutine bfoutni for dumping integer array data (see Appendix B), which presently are not in data-type genTabT. The first actual argument in the bfout calls is the (starting) location of the data to be dumped by the call; it includes index-variable ordInd, which is the Component Index into array genTab. ordInd is calculated in subroutine GenTableDump according to input argument-flag reordered (reordered has been set if the call to GenTableDump is done after the network logic and call to subroutine ASIGN from subroutine INPUT). The second argument in the bfout calls is the number of words to add to the dump file for the call. The third argument, ictrld, is the standard control array for the dump logic, which should not have to be changed.

When adding real scalar variable xxxxx to genTabT: add a call in subroutine GenTableDump to service routine subroutine bfouts.

```
                  SUBROUTINE GenTableDump(compInd,reordered)
                  ---
                  ---
                  ---
      CALL bfouts(genTab(ordInd)%type,1,ictrld)
       CALL bfouts(genTab(ordInd)%xxxxx,1,ictrld)
!
      RETURN
      END SUBROUTINE GenTableDump
```

When adding integer scalar variable iiiii to genTabT: add a call in subroutine GenTableDump to service routine subroutine bfoutis.

```
                  SUBROUTINE GenTableDump(compInd,reordered)
                  ---
                  ---
                  ---
       CALL bfouts(genTab(ordInd)%type,1,ictrld)
       CALL bfoutis(genTab(ordInd)%iiiii,1,ictrld)
!
      RETURN
       END SUBROUTINE GenTableDump
```

When adding real array variable xxxxx (e.g., of length five words) to genTabT: add a call in subroutine GenTableDump to service routine subroutine bfoutn.

```
                SUBROUTINE GenTableDump(compInd,reordered)
                ---
                ---
                ---
      CALL bfouts(genTab(ordInd)%type,1,ictrld)
       CALL bfoutn(genTab(ordInd)%xxxxx,5,ictrld)
!
      RETURN
      END SUBROUTINE GenTableDump
```

When adding integer array variable iiiii (e.g., of length five words) to genTabT: add a call in subroutine GenTableDump to service routine subroutine bfoutni.

```
                SUBROUTINE GenTableDump(compInd,reordered)
                ---
                ---
                ---
      CALL bfouts(genTab(ordInd)%type,1,ictrld)
       CALL bfoutni(genTab(ordInd)%iiiii,5,ictrld)
!
      RETURN
      END SUBROUTINE GenTableDump
```

Coding Requirement: The order of the calls to the bfout routines for individual data elements in subroutine GenTableDump must match the order of the calls to the bfin routines in subroutine GenTableRst (see following section).

Coding Standard: New calls to the bfout routines in subroutine GenTableDump should be appended to the end of the list of existing calls.

3. Read the new variable from the dump/restart file:

Fragment of current coding in module Flt (file FltM.f):

```
      SUBROUTINE GenTableRst(compInd)
                 ---
                 ---
```

```
                  ---
!     The first element, num, is read in SUB rdrest
!     to check missing component list
!
      CALL bfins(genTab(compInd)%htlsci,1,ictrlr)
                  ---
                  ---
                  ---
      CALL bfinn(genTab(compInd)%title,4,ictrlr)
      CALL bfinis(genTab(compInd)%icflg,1,ictrlr)
                  ---
                  ---
                  ---
      CALL bfins(genTab(compInd)%type,1,ictrlr)
!
      RETURN
      END SUBROUTINE GenTableRst
```

End fragment of current coding

Subroutine GenTableRst calls subroutines bfinis, bfins, and bfinn, which are in module Restart. They are the standard service routines for reading from the dump/restart file integer scalar, real scalar, and real array data, respectively; module Restart also has service routine subroutine bfinni for reading integer array data from the dump/restart file (see Appendix B). The first actual argument in the bfin calls is the (starting) location of the data to be read by the call; this argument includes index variable compInd, which is the component index into array genTab (by the point where the dump/restart file is read, the ordInd logic used by subroutine GenTableDump is not needed). The second argument in the bfin calls is the number of words to read from the dump/restart file for the call. The third argument, ictrlr, is the standard control array for the restart logic, which should not have to be changed.

When adding real scalar variable xxxxx to genTabT: add a call in subroutine GenTableRst to service routine subroutine bfins.

```
                  SUBROUTINE GenTableRst(compInd)
                  ---
                  ---
                  ---
      CALL bfins(genTab(compInd)%type,1,ictrlr)
       CALL bfins(genTab(compInd)%xxxxx,1,ictrlr)
!
      RETURN
      END SUBROUTINE GenTableRst
```

When adding integer scalar variable iiiii to genTabT: add a call in subroutine GenTableRst to service routine subroutine bfinis.

```
                SUBROUTINE GenTableRst(compInd)
                ---
                ---
                ---
    CALL bfins(genTab(compInd)%type,1,ictrlr)
     CALL bfinis(genTab(compInd)%iiiii,1,ictrlr)
!
    RETURN
    END SUBROUTINE GenTableRst
```

When adding real array variable xxxxx (e.g., of length five words) to genTabT: add a call in subroutine GenTableRst to service routine subroutine bfinn.

```
                SUBROUTINE GenTableRst(compInd)
                ---
                ---
                ---
    CALL bfins(genTab(compInd)%type,1,ictrlr)
     CALL bfinn(genTab(compInd)%xxxxx,5,ictrlr)
!
    RETURN
    END SUBROUTINE GenTableRst
```

When adding integer array variable iiiii (e.g., of length five words) to genTabT: add a call in subroutine GenTableRst to service routine subroutine bfinni.

```
                SUBROUTINE GenTableRst(compInd)
                ---
                ---
                ---
    CALL bfins(genTab(compInd)%type,1,ictrlr)
     CALL bfinni(genTab(compInd)%iiiii,5,ictrlr)
!
    RETURN
    END SUBROUTINE GenTableRst
```

Coding Requirement: The order of the calls to the bfin routines for individual data elements in subroutine GenTableRst must match the order of the calls to the bfout routines in subroutine GenTableDump.

Coding Standard: New calls to the bfin routines in subroutine GenTableRst should be appended to the end of the list of existing calls.

4. Adding a variable to genTabT that will be used when the component is not instantiated:

Subroutine GetGenTable is a service routine in module Flt that is called from routines that need GenTabT data of a component that is not instantiated. For example, the HTSTR subroutine irodl needs the component type of hydro components that are coupled to a specific HTSTR's inner and/or outer surface.

The argument list of subroutine GetGenTable can return either a real or integer scalar data element for any component index, either reordered or not reordered:

```
      SUBROUTINE GetGenTable(name,compInd,ival,rval,reordered)
      ---
      ---
      ---
      IF (name.EQ.'lenvlt') THEN
        ival=genTab(ordInd)%lenvlt
      ELSEIF (name.EQ.'type') THEN
        rval=genTab(ordInd)%type
      ELSEIF (name.EQ.'typeIndex') THEN
        ival=genTab(ordInd)%typeIndex
      ELSEIF (name.EQ.'ncellt') THEN
        ival=genTab(ordInd)%ncellt
      ELSE
        CALL error(1,'*GetGenTable* variable name not recognized  ',4)
      ENDIF
!
      RETURN
      END SUBROUTINE GetGenTable
```

Subroutine GetGenTable currently (Version 2.119) is set up to treat four GenTabT data elements:

lenvlt, type, typeIndex, and ncellt

New or altered models that require other data elements of GenTabT from uninstantiated components should make appropriate additions to subroutine GetGenTable.

Coding Standard: Subroutine GetGenTable should be used to return all data elements of data-type GenTabT from uninstantiated components.

Using a Variable in genTabT

Instantiated Components

For cases where a genTabT data element is used in a routine that is processing data of a specific (instantiated) component (for Component Index cco or cci), refer directly to the genTabT data element, using the appropriate component index into array genTab (cco

after the reordering of components by subroutine ASIGN or `cci` for components as they are listed in the input deck, before the network-logic reordering).

For example:

```
------------------------------------------------------------------------------
SUBROUTINE tf1ds1(alpo,alp,rov,rol,vl,vv,p,vlt,vvt,vln,vvn,dr,    &
---
---
---
IF (genTab(cco)%type.NE.pumph) msc0=msc
---
---
---
iccmx=genTab(cco)%num
---
---
---
SUBROUTINE tf1ds3(alp,p,vlt,vvt,fa,ara,arv,chti,alv,alve,chtia,   &
---
---
---
WRITE (imout,599) nstep,oitno,genTab(cco)%num,j,etime             &
---
---
---
------------------------------------------------------------------------------
```

Noninstantiated Components

For cases where a `genTabT` data element of an uninstantiated component is needed, use a call to subroutine `GetGenTable`, which is in module `Flt`. Currently (Version 2.119), there are seven calls to `GetGenTable` in TRAC, four from module `Hpss` (for the HPSS initialization) and three from module `RodCrunch` (providing hydrodynamic-Component information to HTSTRs).

The following example is taken from subroutine `irod1` in module `RodCrunch`:

```
60        CALL GetGenTable('type',j,idum1,ityp,.TRUE.)
```

where

argument 1 is a character string corresponding to one of the `genTabTdata` elements treated by `GetGenTable`,

argument 2 is the index of the component requested by `irod1`,

argument 3 receives a returned integer value,

argument 4 receives a returned real value, and

argument 5 indicates if the component list has been reordered.

G.1.3. Adding A New Variable To Data-Types "comp_type" TabT (The Component VLTs)

Each TRAC component type has a set of data that is "global" for a given component and where the variables are common to all components of a given type. These data sets, which are the TRAC-P VLTs, comprise mostly scalar variables that are defined by elements of one of a set of derived data-types; there is a separate derived type for each of the 10 component-types. The 10 VLT data types currently defined in TRAC are

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

These 10 derived data types are referred to as a group by the term

"comp_type"TabT.

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

```
TYPE(breakTabT),DIMENSION(maxComps) :: breakTab
```

to store VLT data for all the individual BREAK components in the input.

All VLT-related logic for a component type is handled by a module that is specific for that type, which has a name of the form

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

In Version 2.119 there are 10 "Comp_type"Vlt modules:

```
BreakVlt
FillVlt
```

```
PipeVlt
PlenVlt
PrizeVlt
PumpVlt
RodVlt
TeeVlt
ValveVlt
VessVlt
```

The logic in each module "Comp_type"`Vlt` comprises

- a definition of derived data-type "comp_type"`TabT` (declaration of its elements);
- a declaration of array "comp_type"`Tab` to be of type "comp_type"`TabT` and dimension `maxComps`;
- parameterization of the total length of data-type "comp_type"`TabT` (for dump/restart);
- subroutine "Comp_type"`TableDump`, which adds an individual component's variables that are stored in array "comp_type"`Tab` to the dump/restart file; and
- subroutine "Comp_type"`TableRst`, which reads a component's "comp_type"`Tab` data from the dump/restart file.

Two additional subroutines are contained in only some of the "Comp_type"`Vlt` modules:

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

In Version 2.119 there are `Get`"Comp_type"`Tab` subroutines for the ROD, TEE, VALVE, PUMP, and VESSEL component types; there is a `Set`"Comp_type"`Tab` subroutine only for the ROD type.

The component-type routines for dump and restart, subroutine "Comp_type"`TableDump` and subroutine "Comp_type"`TableRst`, are called by generic driver subroutines `dmpVLT` and `rstVLT`, respectively, which branch according to the component-type. Subroutines `dmpVLT` and `rstVLT` pass the Component Index `ordInd`

to the component-level dump and restart routines; dmpVLT assumes that reordering has been done; rstVLT assumes reordering has not been done.

The following code fragment gives an overview of the areas in a module "Comp_type"Vlt that are affected when a new variable is added to a "comp_type"TabT data type. The coding is taken from module RodVlt. After this code fragment, specific details are given, again using module RodVlt.

Fragment of current coding in module RodVlt (file RodVltM.f):

```
                    MODULE RodVlt
                    ---
                    ---
                    ---
                    TYPE rodTabT
                      REAL(sdk) amh2
                      REAL(sdk) bcr0
                      ---
                      ---
                      ---
                      INTEGER(sik) nzpwz
                      INTEGER(sik) nzznhc
                    END TYPE rodTabT
!
                    TYPE(rodTabT),DIMENSION(maxComps) :: rodTab
!
!                   This is used to calculate the total dump size, and must
!                   be adjusted when changes are made to rodTab
!
                    INTEGER(sik) rodDumpSize
                    PARAMETER (rodDumpSize=163)  <<<--- rodDumpSize
!
                    CONTAINS
!
                    SUBROUTINE RodTableDump(ordInd,caller)
                    ---
                    ---
                    ---
                    SUBROUTINE RodTableRst(ordInd,caller)
                    ---
                    ---
                    ---
                    SUBROUTINE GetRodTab(name,compInd,ival,rval,reordered)
                    ---
                    ---
                    --
                    SUBROUTINE SetRodTab(name,compInd,ival,rval,reordered)
                    ---
                    ---
                    ---
```

End fragment of current coding

The following examples show the addition of new real variable xxxxx and integer variable iiiii to data-type rodTabT. The logic is the same for the other "comp_type"TabT component data-types (addition of new coding to an existing subroutine Get"Comp_type"Tab or a subroutine Set"Comp_type"Tab or creation of a new Get or Set routine for a component type not already covered is needed only if the new variable is to be made available for reading or overwriting when one of its specific components is not instantiated).

Module RodVlt (file RodVltM.f)

To add real variable xxxxx or integer variable iiiii to rodTabT, the following steps must be taken:

1. Modify the definition of data-type rodTabT and the parameterization of the length of data-type rodTabT:

When adding real variable xxxxx to rodTabT:

1a. add its declaration to the elements of type rodTabT and

1b. increase the size of parameter-variable rodDumpSize as appropriate:

```
 TYPE rodTabT
             ---
             ---
             ---
   REAL(sdk) zlptop
   REAL(sdk) zlpbot
              REAL(sdk) xxxxx  <<<--- declare xxxxx
   INTEGER(sik) iaf
             ---
             ---
             ---
 END TYPE rodTabT
!
      TYPE(rodTabT),DIMENSION(maxComps) :: rodTab
!
!     This is used to calculate the total dump size, and must
!     be adjusted when changes are made to rodTab
!
      INTEGER(sik) rodDumpSize
      PARAMETER (rodDumpSize=n) <<<--- modify rodDumpSize
                 ---
```

```
---
---
```

where n = the total number of words in type rodTabT (reals + integers), counting any individual array elements separately. In Version 2.119, rodDumpSize = 163. Each module "Comp_type"Vlt has a "comp_type"DumpSize parameter; these are calculated separately for each component type. If variable xxxxx is to be an array, specify its dimension (e.g., five words) within its declaration in rodTabT:

```
REAL(sdk) xxxxx(5)
```

Follow an analogous procedure when adding integer variable iiiii to rodTabT:

```
 TYPE rodTabT
   REAL(sdk) amh2
             ---
             ---
             ---
               INTEGER(sik) nzznhc
                INTEGER(sik) iiiii
END TYPE rodTabT
!
TYPE(rodTabT),DIMENSION(maxComps) :: rodTab
!
!    This is used to calculate the total dump size, and must
!    be adjusted when changes are made to rodTab
!
INTEGER(sik) rodDumpSize
 PARAMETER (rodDumpSize=n)
             ---
             ---
             ---
```

where again, n = the total number of words in data-type rodTabT.

Coding Standard: The current ordering of variables in data-type "comp_type"TabT is not significant. However, as described below, the order in which "comp_type"TabT elements are added to the dump file must match the order in which they are read from the restart file. For ease of maintenance, reals and integers are now grouped separately in the "comp_type"TabT definitions, and they are dumped in the same order as they are defined. New declarations of reals in a "comp_type"TabT to the end of the current real declarations, and integers should be appended to the end of the current integers. (Alphabetical order is not a requirement.)

(Note that the special "place-holding" variables used in TRAC-P (aa1111, z11111, ia1111, and zi1111) for determining the length of the component-type VLTs are not used by TRAC-M.)

2. Add the new variable to the dump/restart file:

The "Comp_type"TableDump routines call the standard service routines in module Restart for adding data to the dump/restart file:

subroutine bfouts -- for real scalar variables
subroutine bfoutn -- for real array variables
subroutine bfoutis -- for integer scalar variables
subroutine bfoutni -- for integer array variables

Fragment of current coding in module RodVlt (file RodVltM.f):

```
SUBROUTINE RodTableDump(ordInd,caller)
---
---
---
  CALL bfouts(rodTab(ordInd)%amh2,1,ictrld)
---
---
---
  CALL bfoutn(rodTab(ordInd)%dtnht,2,ictrld)
---
---
---
  CALL bfoutis(rodTab(ordInd)%iaf,1,ictrld)
---
---
---
  CALL bfoutni(rodTab(ordInd)%ibu,4,ictrld)
---
---
---
  CALL bfoutni(rodTab(ordInd)%ircjtb,16,ictrld)
---
---
---
END SUBROUTINE RodTableDump
```

End fragment of current coding

---

where in each of the bfout calls,

> argument 1 is the (starting) location of the data to be dumped by the call; it includes index-variable ordInd, which is the Component Index into array rodTab. ordInd is passed to RodTableDump by subroutine dmpVLT; dmpVLT assumes that component reordering has been done.

argument 2 is the number of words to add to the dump file for the call.

argument 3, ictrld, is the standard control array for the dump logic; it should not have to be changed.

Note that in the fragment above, rodTabT data element ircjtb is a 2D array (4,4); in the call to bfoutni, it is necessary to pass only the total number of words to be dumped, 16.

When adding real scalar variable xxxxx to a "comp_type"TabT: add a call in subroutine "Comp_type"TableDump to Service Routine subroutine bfouts. This new call should be appended to the end of the current list of reals:

```
                SUBROUTINE RodTableDump(ordInd,caller)
                ---
                ---
                ---
      CALL bfouts(rodTab(ordInd)%zupbot,1,ictrld)
      CALL bfouts(rodTab(ordInd)%zlptop,1,ictrld)
      CALL bfouts(rodTab(ordInd)%zlpbot,1,ictrld)
        CALL bfouts(rodTab(ordInd)%xxxxx,1,ictrld)
                ---
                ---
                ---
!
                RETURN
                END SUBROUTINE RodTableDump
```

When adding real array variable xxxxx (e.g., of length five words) to a "comp_type"TabT: add a call in subroutine "Comp_type"TableDump to service routine subroutine bfoutn. This new call should be appended to the end of the current list of reals:

```
                SUBROUTINE RodTableDump(ordInd,caller)
                ---
                ---
                ---
      CALL bfouts(rodTab(ordInd)%zupbot,1,ictrld)
      CALL bfouts(rodTab(ordInd)%zlptop,1,ictrld)
      CALL bfouts(rodTab(ordInd)%zlpbot,1,ictrld)
        CALL bfoutn(rodTab(ordInd)%xxxxx,5,ictrld)
                ---
                ---
                ---
!
       RETURN
```

```
      END SUBROUTINE RodTableDump
```

When adding integer scalar variable iiiii to a "comp_type"TabT: add a call in subroutine "Comp_type"TableDump to service routine subroutine bfoutis. This new call should be appended to the end of the current list of integers:

```
                  SUBROUTINE RodTableDump(ordInd,caller)
                  ---
                  ---
                  ---
      CALL bfoutis(rodTab(ordInd)%nzpwi,1,ictrld)
      CALL bfoutis(rodTab(ordInd)%nzpwz,1,ictrld)
      CALL bfoutis(rodTab(ordInd)%nzznhc,1,ictrld)
        CALL bfoutis(rodTab(ordInd)%iiiii,1,ictrld)
!
      RETURN
      END SUBROUTINE RodTableDump
```

When adding integer array variable iiiii (e.g., of length 5 words) to a "comp_type"TabT : add a call in subroutine "Comp_type"TableDump to service routine subroutine bfoutni. This new call should be appended to the end of the current list of integers:

```
       SUBROUTINE RodTableDump(ordInd,caller)
                  ---
                  ---
                  ---
      CALL bfoutis(rodTab(ordInd)%nzpwi,1,ictrld)
      CALL bfoutis(rodTab(ordInd)%nzpwz,1,ictrld)
      CALL bfoutis(rodTab(ordInd)%nzznhc,1,ictrld)
        CALL bfoutni(rodTab(ordInd)%iiiii,5,ictrld)
!
       RETURN
       END SUBROUTINE RodTableDump
```

Coding Requirement: The order of the calls to the bfout routines for the individual variables in a subroutine "Comp type"TableDump must match the order of the calls to the bfin routines in the corresponding subroutine "Comp type"TableRst (see the following section).

Coding Standard: New calls for reals (scalar and array) to the bfout routines in a subroutine "Comp type"TableDump should be appended to the end of the list of existing real calls. New calls for integers (scalar and array) to the bfout routines in a subroutine

"Comp_type"TableDump should be appended to the end of the list of existing integer calls.

3. Read the new variable from the dump/restart file:

The "Comp_type"TableRst routines call the standard service routines in module Restart for reading data from the dump/restart file:

subroutine bfins -- for real scalar variables
subroutine bfinn -- for real array variables
subroutine bfinis -- for integer scalar variables
subroutine bfinni -- for integer array variables

Fragment of current coding in module RodVlt (file RodVltM.f):

```
SUBROUTINE RodTableRst(ordInd,caller)
---
---
---
  CALL bfins(rodTab(ordInd)%amh2,1,ictrlr)
---
---
---
  CALL bfinn(rodTab(ordInd)%dtnht,2,ictrlr)
---
---
---
  CALL bfinis(rodTab(ordInd)%iaf,1,ictrlr)
---
---
---
  CALL bfinni(rodTab(ordInd)%ibu,4,ictrlr)
---
---
---
  CALL bfinni(rodTab(ordInd)%ircjtb,16,ictrlr)
---
---
---
END SUBROUTINE RodTableRst
```

End fragment of current coding

---

where in each of the bfin calls,

argument 1 is the (starting) location of the data to be read by the call; it includes index-variable ordInd, which is the Component Index into array rodTab. ordInd is passed to RodTableRst by subroutine rstVLT; rstVLT assumes component reordering has not been done.

argument 2 is the number of words to read from the restart file for the call.

argument 3, ictrlr, is the standard control array for the restart logic; it should not have to be changed.

Note that in the fragment above, rodTabT data element ircjtb is a 2D array (4,4); in the call to bfinni, it is necessary to pass only the total number of words to be read, 16.

When adding real scalar variable xxxxx to a "comp_type"TabT: add a call in subroutine "Comp_type"TableRst to service routine subroutine bfins. This new call should be appended to the end of the current list of reals:

```
                SUBROUTINE RodTableRst(ordInd,caller)
                ---
                ---
                ---
      CALL bfins(rodTab(ordInd)%zupbot,1,ictrlr)
      CALL bfins(rodTab(ordInd)%zlptop,1,ictrlr)
      CALL bfins(rodTab(ordInd)%zlpbot,1,ictrlr)
        CALL bfins(rodTab(ordInd)%xxxxx,1,ictrlr)
                ---
                ---
                ---
!
                RETURN
                END SUBROUTINE RodTableRst
```

When adding real array variable xxxxx (e.g., of length five words) to a "comp_type"TabT: add a call in subroutine "Comp_type"TableRst to service routine subroutine bfinn. This new call should be appended to the end of the current list of reals:

```
                SUBROUTINE RodTableRst(ordInd,caller)
                ---
                ---
                ---
      CALL bfins(rodTab(ordInd)%zupbot,1,ictrlr)
      CALL bfins(rodTab(ordInd)%zlptop,1,ictrlr)
      CALL bfins(rodTab(ordInd)%zlpbot,1,ictrlr)
        CALL bfinn(rodTab(ordInd)%xxxxx,5,ictrlr)
                ---
                ---
```

```
                  ---
!
      RETURN
      END SUBROUTINE RodTableRst
```

When adding integer scalar variable iiiii to a "comp_type"TabT: add a call in subroutine "Comp_type"TableRst to service routine subroutine bfinis. This new call should be appended to the end of the current list of integers:

```
                  SUBROUTINE RodTableRst(ordInd,caller)
                  ---
                  ---
                  ---
      CALL bfinis(rodTab(ordInd)%nzpwi,1,ictrlr)
      CALL bfinis(rodTab(ordInd)%nzpwz,1,ictrlr)
      CALL bfinis(rodTab(ordInd)%iiiii,1,ictrlr)
!
      RETURN
      END SUBROUTINE RodTableRst
```

When adding integer array variable iiiii (e.g., of length five words) to a "comp_type"TabT : add a call in subroutine "Comp_type"TableRst to service routine subroutine bfinni. This new call should be appended to the end of the current list of integers:

```
                  SUBROUTINE RodTableRst(ordInd,caller)
                  ---
                  ---
                  ---
      CALL bfinis(rodTab(ordInd)%nzpwi,1,ictrlr)
      CALL bfinis(rodTab(ordInd)%nzpwz,1,ictrlr)
      CALL bfinis(rodTab(ordInd)%nzznhc,1,ictrlr)
        CALL bfinni(rodTab(ordInd)%iiiii,5,ictrlr)
!
      RETURN
      END SUBROUTINE RodTableRst
```

Coding Requirement: The order of the calls to the bfin routines for individual variables in a subroutine "Comp_type"TableRst must match the order of the calls to the bfout routines in the corresponding subroutine "Comp_type"TableDump (see previous section).

Coding Standard: New calls for reals (scalar and array) to the bfin routines in a subroutine "Comp_type"TableRst should be appended to the end of the list of existing real calls. New calls for integers (scalar and array) to the bfin routines in a subroutine "Comp_type"TableRst should be appended to the end of the list of existing integer calls.

4. Adding a variable to "comp_type"TabT that will be used when the component is not instantiated:

Some of the component types have data access service routines in their module "comp_type"Vlt;; these routines are called from routines that need "comp_type"TabT data of components that are not instantiated. These routines have names of the form Get"Comp_type"Tab. Some component types (currently only the HTSTR) have a corresponding service routine in their module "comp_type"Vlt that allows overwriting of their data when one of their components is not instantiated. These routines have names of the form Set"Comp_type"Tab. For example, in subroutine core1, the coupled neutronics-group logic needs the power of uninstantiated HTSTRs and also needs to adjust their power:

```
!
!  adjust the power for the heat structures in a neutronics
!  calculation group
!
         IF (rodTab(cco)%m1dt.NE.0) THEN
           ratio=rodTab(cco)%rpowrn/rodTab(cco)%rpowr
           icmpm1=icmp-1
           DO i=icmp1,icmpm1
!           NOT TESTED IN SHORT SET
!           jdearing 1/97
             CALL GetRodTab('rpowrn',i,idum1,rpowrnx,.TRUE.)<<<---
             rpowrnx=rpowrnx*ratio                          <<<---
             CALL SetRodTab('rpowrn',i,idum1,rpowrnx,.TRUE.)<<<---
!            a(ig(lm1dp+i-1))=ratio*a(ig(lm1dp+i-1))
           ENDDO
           icmp1=0
         ENDIF
```

The argument list of a subroutine Get"Comp_type"Tab can return either a real or integer scalar data element for any component index, either reordered or not reordered. The logic is similar to that of subroutine GetGenTable (see the section on the component FLTs):

```
      SUBROUTINE GetRodTab(name,compInd,ival,rval,reordered)
      ---
      ---
      ordInd = compInd
      if(reordered) ordInd = compIndices(compInd)
!
      IF     (name.EQ.'iis') THEN
        ival=rodTab(ordInd)%iis
      ELSEIF (name.EQ.'idbci') THEN
        ival=rodTab(ordInd)%idbci
      ELSEIF (name.EQ.'idbco') THEN
        ival=rodTab(ordInd)%idbco
      ELSEIF (name.EQ.'rpowrn') THEN
        rval=rodTab(ordInd)%rpowrn
```

```
      ELSEIF (name.EQ.'ncrx') THEN
        ival=rodTab(ordInd)%ncrx
      ELSE
        CALL error(1,'*getRodTab* variable name not recognized   ',4)
      ENDIF
!
                  RETURN
                  END SUBROUTINE GetRodTab
```

Currently (Version 2.119), there are Get VLT routines for the PUMP, HTSTR, TEE, VALVE, and VESSEL components. The various subroutine Get"Comp_type"Tab are designed to treat "comp_type"TabT data elements on an as-needed basis.

New or altered models that require other data elements of a "comp_type"TabT from uninstantiated components should make appropriate additions to the appropriate subroutine Get"Comp_type"Tab.

Coding Standard: A subroutine Get"Comp_type"Tab should be used to return all data elements of a data-type "comp_type"TabT from uninstantiated components.

The argument list of a subroutine Set"Comp_type"Tab can overwrite either a real or integer scalar data element for any component index, either reordered or not reordered:

```
      SUBROUTINE SetRodTab(name,compInd,ival,rval,reordered)
      ---
      ---
      ---
      ordInd = compInd
      if(reordered) ordInd = compIndices(compInd)
!
      IF     (name.EQ.'rpowrn') THEN
        rodTab(cco)%rpowrn=rval   ERROR: (cco --->>> ordInd)
      ELSE
        CALL error(1,'*setRod* variable name not recognized   ',4)
      ENDIF
!
      RETURN
      END SUBROUTINE SetRodTab
```

Currently (Version 2.119), there is a Set VLT routine only for the HTSTR component. Subroutine SetRodTab is designed to treat rodTabT data elements on an as-needed basis.

New or altered models that need to overwrite other data elements of a "comp_type"TabT of uninstantiated components should make appropriate additions to the appropriate subroutine Set"Comp_type"Tab.

Coding Standard: A subroutine Set"Comp_type"Tab should be used for any overwrites of data elements of a data-type "comp_type"TabT of uninstantiated components.

Using a Variable in a "comp_type" TabT:

Instantiated Components

For cases where a "comp_type"TabT data element is used in a routine that is processing data of a specific (instantiated) component (for Component Index cco or cci), refer directly to the "comp_type"TabT data element, using the appropriate component index into array "comp_type"Tab (cco after the reordering of components by subroutine ASIGN, or cci for components as they are listed in the input deck, before the network-logic reordering).

For example:

```
      MODULE Pipe
!
!     BEGIN MODULE USE
      USE PipeArray
!
      CONTAINS

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

Noninstantiated Components

For cases where the value of a "comp_type"TabT data element of an uninstantiated component is needed, use a call to the appropriate subroutine Get"Comp_type"Tab, which is in module "Comp_type"Vlt. Where a "comp_type"TabT data element of an

uninstantiated component must be overwritten, use a call to the appropriate subroutine Set"Comp_type"Tab, which is also in module "Comp_type"Vlt.

The following example is taken from subroutine core1 in module RodTask:

```
CALL GetRodTab('rpowrn',i,idum1,rpowrnx,.TRUE.)
rpowrnx=rpowrnx*ratio
CALL SetRodTab('rpowrn',i,idum1,rpowrnx,.TRUE.)    ,
```

where for both routines:

argument 1 is a character string corresponding to one of the rodTabT data elements treated by GetRodTab or SetRodTab,

argument 2 is the index of the component requested by core1,

argument 3 receives/overwrites an integer value,

argument 4 receives/overwrites a real value, and

argument 5 indicates if the component list has been reordered.

G.1.4. Adding A New Component Array Variable

G.1.4.1. 1D Hydrodynamic Components

The addition of new arrays that are specific for a given 1D hydrodynamic component type is treated first. A new array then may be added to the general 1D-component data arrays if the new general array is to be made available to the data interface in module Gen1DArray (for noninstantiated components).

All TRAC 1D hydrodynamic component types have a module with a name of the form

module "Comp_type".

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

In the following we use the PIPE-component type for specific examples. Module Pipe contains the following routines:

```
Module Pipe (file PipeM.f)

    USE PipeArray (file PipeArrayM.f)

    CONTAINS
```

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

Adding an Array -- Specific for PIPE Component

The following example shows the addition of a new real, rank-1 array called

```
YYYYY   ,
```

of dimension ncells.

Module PipeArray (file PipeArrayM.f)

In Version 2.120, module PipeArray defines derived data-type pipeArrayT for the PIPE-component-specific arrays powrf, powtb, qp3rf, and qp3tb and declares array pipeAr to be of this type:

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

Add the following declaration for new array yyyyy to derived data-type pipeArrayT:

```
TYPE pipeArrayT
  REAL(sdk),     POINTER, DIMENSION(:)    :: powrf
  REAL(sdk),     POINTER, DIMENSION(:)    :: powtb
  REAL(sdk),     POINTER, DIMENSION(:)    :: qp3rf
  REAL(sdk),     POINTER, DIMENSION(:)    :: qp3tb
  REAL(sdk),     POINTER, DIMENSION(:)    :: yyyyy
END TYPE pipeArrayT
```

Module Pipe (file PipeM.f)

Allocate storage for new pipe-specific array yyyyy by calling subroutine TRACAllo. TRACAllo is a generic name (interface) for subroutines AllocRealOneD, AllocRealTwoD, AllocRealThreeD, and AllocIntOneD. The call to TRACAllo is inserted in subroutine rpipe and in subroutine repipe; the form of the call is the same in both routines:

```
SUBROUTINE rpipe(......)
```

```
 ---
 ---
 ---
CALL TRACAllo(pipeAr(cci)%powrf,.....)
CALL TRACAllo(pipeAr(cci)%powtb,.....)
CALL TRACAllo(pipeAr(cci)%qp3rf,.....)
CALL TRACAllo(pipeAr(cci)%qp3tb,.....)
CALL TRACAllo(pipeAr(cci)%yyyyy,ncells,'yyyyy',0.d0)
```

where `cci` is the current Component Index into `pipeAr` (not reordered by subroutine `asign`), `ncells` is the dimension of array `yyyyy`, `'yyyyy'` is the name of `yyyyy`, and `0.d0` will be used to initialize `yyyyy`.

```
SUBROUTINE repipe(.....)
 ---
 ---
 ---
CALL TRACAllo(pipeAr(cci)%powrf,.....)
CALL TRACAllo(pipeAr(cci)%powtb,.....)
CALL TRACAllo(pipeAr(cci)%qp3rf,.....)
CALL TRACAllo(pipeAr(cci)%qp3tb,.....)
CALL TRACAllo(pipeAr(cci)%yyyyy,ncells,'yyyyy',0.d0)
```

where the actual arguments to `TRACAllo` are as in the call from `rpipe`.

Add array `yyyyy` to the dump file `trcdmp` by calling subroutine `bfoutn`; the call to `bfoutn` is inserted in subroutine `dpipe`:

```
SUBROUTINE dpipe(.....)
 ---
 ---
 ---
CALL bfoutn(pipeAr(cco)%powtb,.....)
CALL bfoutn(pipeAr(cco)%powrf,.....)
 ---
CALL bfoutn(pipeAr(cco)%qp3tb,.....)
CALL bfoutn(pipeAr(cco)%qp3rf,.....)
CALL bfoutn(pipeAr(cco)%yyyyy,ncells,ictrld)
```

where `cco` is the current Component Index into `pipeAr` (reordered by subroutine `asign`), `ncells` is the dimension of array `yyyyy`, and `ictrld` is the standard control array for the dump logic.

Read array `yyyyy` from the input file `tracin` by calling subroutine `loadn`, and echo the input to output file `trcout` by calling subroutine `warray`; the read and write calls are inserted in subroutine `rpipe`, after storage for `yyyyy` has been allocated:

```
      SUBROUTINE rpipe(.......)
---
---
CALL loadn(pipeAr(cci)%yyyyy,ncells,1)
CALL warray('yyyyy    ',pipeAr(cci)%yyyyy,ncells,_sv)
```

where cci is the current Component Index (not reordered by asign), ncells is the dimension of array yyyyy, the argument 1 tells loadn to read a real variable, 'yyyyy ' is the name of yyyyy (padded to eight characters), and __sv is the ID number of the signal variable or control block corresponding to the odd-numbered elements of array pipeAr(cci) (see the comments in subroutine warray for more information on argument __sv).

Read array yyyyy from the restart file trcrst by calling subroutine bfinn, and echo the restart-file input to output file trcout by calling subroutine warray; the read and write calls are inserted in subroutine repipe, after storage for yyyyy has been allocated:

```
      SUBROUTINE repipe(.......)
---
---
CALL bfinn(pipeAr(cci)%yyyyy,ncells,ictrlr)
CALL warray('yyyyy    ',pipeAr(cci)%yyyyy,ncells,_sv)
```

where cci is the current Component Index (not reordered by asign), ncells is the dimension of array yyyyy, ictrlr is the standard control array for the restart input logic, 'yyyyy' is the name of yyyyy (padded to eight characters), and __sv is the ID number of the signal variable or control block corresponding to the odd-numbered elements of array pipeAr(cci).

If appropriate, write array yyyyy to the text output file (trcout) large edits. Currently, no data in the component-specific arrays must be written to the large edits for the 1D hydrodynamic components. If the need does arise, the edits should be placed in the component-specific trcout driver routines, such as

```
      SUBROUTINE wpipe(cfmass,cener)     ,
```

which is in module Pipe.

Adding an Array—General ("generic") Arrays

The following example shows the addition of a new real, rank-1 array called

```
      yyyyy    ,
```

of dimension ncells, to the general ("generic") 1D hydrodynamic component arrays, which are elements of derived data-type g1DArrayT (in TRAC-P, these arrays are allocated and accessed in the hydropt and dualpt pointer tables). Many of these

arrays hold data for old- or new-time values of the same quantity (TRAC-P's dualpt); for such old- and new time arrays, sample arrays yyyyy and yyyyyn, respectively, are treated.

The hydropt and dualpt arrays for all 1D components are stored in and accessed from array g1dAr by the following steps:

- The driver input routine (rpipe, repipe, rtee, retee, etc.) for each 1D component in the input (or restart) deck uses module Gen1DArray, which defines derived data-type g1DArrayT. The members of data-type g1DArrayT are the former hydropt and dualpt arrays, which are declared as pointers of dimension (:).

- Module Gen1DArray declares array g1DAr to be of type (g1DArrayT) and of dimension (maxComps).

- The driver input routine then allocates storage for the generic arrays for the specific component it is reading by calling subroutine AllocGen1D, which is also in module Gen1DArray. AllocGen1D has a call to subroutine TRACAllo for each array that is a member of array g1DAr.

The remaining 1D-component generic arrays (TRAC-P's intpt and heatpt) are treated in a similar manner but are stored in their own arrays called intAr and heatAr, respectively. Module IntArray defines data-type intArrayT, and module HeatArray defines data-type heatArrayT. Module Gen1DArray calls TRACAllo for each member of intAr and heatAr.

Module Gen1DArray (file Gen1DArray M.f)

To add real rank-1 array yyyyy, of dimension (ncells), to hydropt:

Within type g1DArrayT add the following declaration:

```
TYPE g1DArrayT
---
---
---
REAL(sdk), POINTER, DIMENSION(:) :: vlsm
REAL(sdk), POINTER, DIMENSION(:) :: qrl
REAL(sdk), POINTER, DIMENSION(:) :: qrv
REAL(sdk), POINTER, DIMENSION(:) :: yyyyy
---
---
```

Allocate ncells of storage for array yyyyy for the component being read from the input or restart file by adding a call to TRACAllo in subroutine AllocGen1D, which is contained in module Gen1DArray:

```
SUBROUTINE AllocGen1D(ncells,nfaces,nods,inflg,ihtflg)
---
---
---
USE Alloc
---
---
---
CALL TRACAllo(g1DAr(cci)%cl,ncells,'cl',0.0d0)
CALL TRACAllo(g1DAr(cci)%dfvdp,nfaces,'dfvdp',0.0d0)
CALL TRACAllo(g1DAr(cci)%dfldp,nfaces,'dfldp',0.0d0)
g1DAr(cci)%cfz=>g1DAr(cci)%dfldp
CALL TRACAllo(g1DAr(cci)%yyyyy,ncells,'yyyyy',0.0d0)
---
---
```

where `cci` is the Component Index (not reordered), `ncells` is the dimension of array `yyyyy`, `'yyyyy'` is the name of `yyyyy`, and `0.0d0` will be used to initialize `yyyyy`. (Note that the association of pointer `cfz` with `dfldp` in this code fragment (taken from Version 2.120) treats one of several special cases in the generic arrays.)

Subroutine `rcomp` (file `rcomp.f`)

Read array `yyyyy` from text input file `tracin`, and echo the input to output file `trcout` by adding the following to subroutine `rcomp`:

```
      SUBROUTINE rcomp(ncells,nods,bump,idef,iconc)
      ---
      ---
      ---
      USE Gen1DArray
      ---
      ---
      ---
      REAL(sdk), POINTER, DIMENSION(:) ::.....................&
&dxb,volb,fab,.....................................&
  & vvnb,tlnb,pnb,.....................................&
  & qpppb,yyyyyb
      ---
      ---
      ---
    yyyyyb=>g1dAr(cci)%yyyyy(bump+1:)
      CALL loadn(yyyyyb,ncells,1)
      CALL warray('yyyyy   ',yyyyyb,ncells,0)
      ---
```

```
---
---
```

where the associations of the local "b-suffix" pointers were required by at least one of the Fortran 90 compilers used when TRAC-M was originally developed.

This example does not include the special logic in `rcomp` for setting certain arrays to user-input default values, under the control of namelist variable `istopt`.

Subroutine `dcomp` (file `dcomp.f`)

Add array yyyyy to the dump file `trcdmp`:

```
      SUBROUTINE dcomp(icomp)
      ---
      ---
      ---
      USE Gen1DArray
      ---
      ---
      ---
      CALL bfoutn(g1dAr(cco)%alven,genTab(cco)%ncellt,ictrld)
      CALL bfoutn(g1dAr(cco)%twan,1,ictrld)
      CALL bfoutn(g1dAr(cco)%twen,1,ictrld)
      CALL bfoutn(g1dAr(cco)%tcen,1,ictrld)
      IF (isolut.NE.0) THEN
        CALL bfoutn(g1dAr(cco)%sn,genTab(cco)%ncellt,ictrld)
        CALL bfoutn(g1dAr(cco)%concn,genTab(cco)%ncellt,ictrld)
      ENDIF
      IF (nods.GT.0) CALL bfoutn(g1dAr(cco)%qppc,genTab(cco)%ncellt      &
     &,ictrld)
      CALL bfoutn(g1dAr(cco)%yyyyy,genTab(cco)%ncellt,ictrld)
```

Subroutine `recomp` (file `recomp.f`)

Read array yyyyy from the restart file `trcrst`:

```
      SUBROUTINE recomp(bump,ncells,nods)
      ---
      ---
      ---
      USE Gen1DArray
      ---
      ---
      ---
      CALL bfinn(g1dAr(cci)%alven,ncells,ictrlr)
      CALL bfinn(g1dAr(cci)%twan,1,ictrlr)
      CALL bfinn(g1dAr(cci)%twen,1,ictrlr)
      CALL bfinn(g1dAr(cci)%tcen,1,ictrlr)
      IF (isolut.NE.0) THEN
```

```
        CALL bfinn(g1dAr(cci)%sn,ncells,ictrlr)
        CALL bfinn(g1dAr(cci)%concn,ncells,ictrlr)
      ENDIF
      IF (nods.GT.0) CALL bfinn(g1dAr(cci)%qppc,ncells,ictrlr)
      CALL bfinn(g1dAr(cci)%yyyyy,ncells,ictrlr)
```

Coding Requirement: The order of the calls to `bfoutn` in subroutine `dcomp` is not significant, but it must match the order of calls to `bfinn` in subroutine `recomp`.

Subroutine `ecomp` (file `ecomp.f`)

If needed, add array `yyyyy` to the large (major) edits to text output file `trcout`. Certainly, the specific changes to `ecomp` will depend on the desired output format; the following code fragment is the existing coding for the basic edit of hydrodynamic data. Note that new-time data typically are printed in the large edits; additional details on adding a dual-time array are given in the following section.

```
      SUBROUTINE ecomp(jstrt,jstop,iflgw,nodes,cfmass,xpintg,cener)
      ---
      ---
      ---
      DIMENSION tmp(10,24) <<<--- array tmp is used for units conversion
!
!     print out hydraulic-solution parameters
!
      WRITE (iout,100) lup,lup,lud,lutp,lutp,lutp,lur,lur,luv,luv,lud
  100 FORMAT (/19x,'ncd-gas'/7x,'pressure    pressure    void fr. temp',  &
     & '.sat. temp.liq. temp.gas   den.liq.  den.vap.   vel.liq',         &
     & '.   vel.gas   wf.liq.'/' cell',3x,a,7x,a,7x,a,8x,a,8x,a,8x,a,    &
     & 5x,a,2x,a,5x,a,6x,a,6x,a)
      nn=1+(jstop-jstrt)/10
      jn=0
      DO n=1,nn
        j1=jstrt+(n-1)*10
        j2= min(j1+9,jstop)
        j0=j1-1
        j3=j2-j0
        DO j=j1,j2
          jj=j-j0
          jm1=j-1
      tmp(jj,1)=g1dAr(cco)%pn(jm1+1)  <<<--- store in array tmp
          tmp(jj,2)=g1dAr(cco)%pan(jm1+1)
          tmp(jj,3)=g1dAr(cco)%alpn(jm1+1)
          tmp(jj,4)=g1dAr(cco)%tsat(jm1+1)
          tmp(jj,5)=g1dAr(cco)%tln(jm1+1)
          tmp(jj,6)=g1dAr(cco)%tvn(jm1+1)
          tmp(jj,7)=g1dAr(cco)%roln(jm1+1)
          tmp(jj,8)=g1dAr(cco)%rovn(jm1+1)
          tmp(jj,9)=g1dAr(cco)%vln(jm1+1)
          tmp(jj,10)=g1dAr(cco)%vvn(jm1+1)
          IF (btestc(g1dAr(cco)%bitn(jm1+1),chokedFlowOn).NE                &
     &     .0) g1dAr(cco)%wfl(jm1+1)=-1.111d-11
          tmp(jj,11)=g1dAr(cco)%wfl(jm1+1)
```

```
          cfmass=cfmass+g1dAr(cco)%vol(jm1+1)*(g1dAr(cco)%arv(jm1+1)      &
     &     +g1dAr(cco)%arl(jm1+1))
          cener=cener+g1dAr(cco)%vol(jm1+1)*(g1dAr(cco)%arev(jm1+1)      &
     &     +g1dAr(cco)%arel(jm1+1))
        ENDDO
        IF (ioout.EQ.1) THEN
          CALL uncnvt('pn',tmp(1,1),j3,1,-1) <<<--- units conversion
          CALL uncnvt('pan',tmp(1,2),j3,1,-1)
          CALL uncnvt('tsat',tmp(1,4),j3,1,-1)
          CALL uncnvt('tln',tmp(1,5),j3,1,-1)
          CALL uncnvt('tvn',tmp(1,6),j3,1,-1)
          CALL uncnvt('roln',tmp(1,7),j3,1,-1)
          CALL uncnvt('rovn',tmp(1,8),j3,1,-1)
          CALL uncnvt('vln',tmp(1,9),j3,1,-1)
          CALL uncnvt('vvn',tmp(1,10),j3,1,-1)
        ENDIF
        j1=j1-j0
        j2=j2-j0
        WRITE (iout,120) (jn+j,(tmp(j,k),k=1,11),j=j1,j2)
 120    FORMAT (1x,i3,1p,2e12.5,6e10.3,2e11.3,e10.3)
        jn=jn+10
      ENDDO
!
```

edit last cell-face:

```
      tmp(1,1)=g1dAr(cco)%vln(jstop+1)
      tmp(2,1)=g1dAr(cco)%vvn(jstop+1)
      IF (ioout.EQ.1) CALL uncnvt('vln',tmp,2,1,-1)
      IF (btestc(g1dAr(cco)%bitn(jstop+1),chokedFlowOn).NE                &
     &.0) g1dAr(cco)%wfl(jstop+1)=-1.111d-11
      jn=jstop-jstrt+2
      WRITE (iout,140) jn,tmp(1,1),tmp(2,1),g1dAr(cco)%wfl(jstop+1)
 140 FORMAT (1x,i3,84x,1p,2e11.3,e10.3)
```

To add real rank-1 arrays yyyyy and yyyyyn, of dimension (ncells), to dualpt:

Within type g1DArrayT, add the following declarations:

```
TYPE g1DArrayT
---
---
---
  REAL(sdk), POINTER, DIMENSION(:) :: twa
  REAL(sdk), POINTER, DIMENSION(:) :: twe
  REAL(sdk), POINTER, DIMENSION(:) :: tce
   REAL(sdk), POINTER, DIMENSION(:) :: YYYYY
---
---
---
  REAL(sdk), POINTER, DIMENSION(:) :: twan
  REAL(sdk), POINTER, DIMENSION(:) :: twen
  REAL(sdk), POINTER, DIMENSION(:) :: tcen
```

```
        REAL(sdk), POINTER, DIMENSION(:) :: yyyyyn
    END TYPE g1DArrayT
```

Allocate ncells of storage for arrays yyyyy and yyyyyn, for the component being read from the input or restart file, by adding calls to TRACAllo in subroutine AllocGen1D, which is contained in module Gen1DArray:

```
SUBROUTINE AllocGen1D(ncells,nfaces,nods,inflg,ihtflg)
---
---
---
USE Alloc
---
---
---
CALL TRACAllo(g1DAr(cci)%alven,ncells,'alven',0.0d0)
CALL TRACAllo(g1DAr(cci)%alvn,ncells,'alvn',0.0d0)
g1DAr(cci)%alpdn=>g1DAr(cci)%alvn
CALL TRACAllo(g1DAr(cci)%alpn,ncells,'alpn',0.0d0)
CALL TRACAllo(g1DAr(cci)%yyyyyn,ncells,'yyyyyn',0.0d0)
---
---
---
CALL TRACAllo(g1DAr(cci)%alve,ncells,'alve',0.0d0)
CALL TRACAllo(g1DAr(cci)%alv,ncells,'alv',0.0d0)
g1DAr(cci)%alpd=>g1DAr(cci)%alv
CALL TRACAllo(g1DAr(cci)%alp,ncells,'alp',0.0d0)
CALL TRACAllo(g1DAr(cci)%yyyyy,ncells,'yyyyy',0.0d0)
```

where cci is the Component Index (not reordered), ncells is the dimension of arrays yyyyyn and yyyyy, 'yyyyyn' and 'yyyyy' are the names of yyyyyn and yyyyy, and 0.0d0 will be used to initialize yyyyyn and yyyyy. [Note that the associations of pointers alpdn and alpd with alvn and alv, respectively, in this code fragment (taken from Version 2.120) treat two of several special cases in the generic arrays.]

Coding Standard: Type G1DARRAYT first declares the hydropt arrays, then the dualpt arrays. Add a new hydropt variable to the existing hydropt portion of type g1DArrayT and a new dualpt variable to the dualpt portion. The dualpt portion first lists old-time arrays, then new-time arrays.

Note that in TRAC-P, the ordering of the assignment of the dualpt array pointers (in subroutine s1dptr) is significant for supporting the logic for a water-packing-type backup. This is no longer the case in TRAC-M: the water-packing, backup-specific arrays are explicitly treated in subroutine BackUpGen1D, in module Gen1DArray (details are given below).

Coding Standard: The order of allocation of the hydropt and dualpt arrays in AllocGen1D is not significant; however, the two sets of arrays are grouped together, and this grouping should be maintained. The dualpt arrays are grouped further into new- and old-time arrays.

Add new-time to old-time data transfers and old-time to new-time data transfers to subroutine TimeUpGen1D, which is contained in module Gen1DArray:

```
                SUBROUTINE TimeUpGen1D(newToOld)
                ---
                ---
                ---
                IF (newToOld) THEN
                ---
                ---
                ---
                g1dAr(cco)%twa    = g1dAr(cco)%twan
                g1dAr(cco)%twe    = g1dAr(cco)%twen
                g1dAr(cco)%tce    = g1dAr(cco)%tcen
                g1dAr(cco)%yyyyy = g1dAr(cco)%yyyyyn
!
                ELSE
!
                ---
                ---
                ---
                g1dAr(cco)%twan   = g1dAr(cco)%twa
                g1dAr(cco)%twen   = g1dAr(cco)%twe
                g1dAr(cco)%tcen   = g1dAr(cco)%tce
                g1dAr(cco)%yyyyyn= g1dAr(cco)%yyyyy
!
                ENDIF
                ---
                ---
                ---
```

where cco is the Component Index (reordered).

Logic for Special Timestep Backups

The "standard" TRAC timestep backup (forced, e.g., by convergence difficulties with a given timestep size) repeats a timestep from the start of the PREP stage. Another type of backup to the start of the outer stage is forced by detection of water packing. Care must be taken that the new-time arrays have the proper values when such a special backup occurs. This logic is handled by subroutine BackUpGen1D, which is contained in module Gen1DArray.

If the values in arrays yyyyy and yyyyyn are the same at the start of the outer stage of a timestep, add the following old-time to new-time data transfer to subroutine BackUpGen1D:

```
                SUBROUTINE BackUpGen1D
                ---
                ---
                ---
                g1dAr(cco)%twn    = g1dAr(cco)%tw
                g1dAr(cco)%vln    = g1dAr(cco)%vl
                g1dAr(cco)%vvn    = g1dAr(cco)%vv
                g1dAr(cco)%yyyyyn= g1dAr(cco)%yyyyy
!
                END SUBROUTINE BackUpGen1D
```

where cco is the Component Index (reordered).

Note that for dual-time arrays, the new-time arrays are dumped and read from the restart file:

Subroutine dcomp (file dcomp.f)

Add array yyyyyn to the dump file trcdmp:

```
      SUBROUTINE dcomp(icomp)
      ---
      ---
      ---
      USE Gen1DArray
      ---
      ---
      ---
      CALL bfoutn(g1dAr(cco)%alven,genTab(cco)%ncellt,ictrld)
      CALL bfoutn(g1dAr(cco)%twan,1,ictrld)
      CALL bfoutn(g1dAr(cco)%twen,1,ictrld)
      CALL bfoutn(g1dAr(cco)%tcen,1,ictrld)
      IF (isolut.NE.0) THEN
        CALL bfoutn(g1dAr(cco)%sn,genTab(cco)%ncellt,ictrld)
        CALL bfoutn(g1dAr(cco)%concn,genTab(cco)%ncellt,ictrld)
      ENDIF
      IF (nods.GT.0) CALL bfoutn(g1dAr(cco)%qppc,genTab(cco)%ncellt      &
    &,ictrld)
      CALL bfoutn(g1dAr(cco)%yyyyyn,genTab(cco)%ncellt,ictrld)
```

Subroutine recomp (file recomp.f)

Read array yyyyyn from the restart file trcrst:

```
      SUBROUTINE recomp(bump,ncells,nods)
```

```
      ---
      ---
      ---
      USE Gen1DArray
      ---
      ---
      ---
      CALL bfinn(g1dAr(cci)%alven,ncells,ictrlr)
      CALL bfinn(g1dAr(cci)%twan,1,ictrlr)
      CALL bfinn(g1dAr(cci)%twen,1,ictrlr)
      CALL bfinn(g1dAr(cci)%tcen,1,ictrlr)
      IF (isolut.NE.0) THEN
        CALL bfinn(g1dAr(cci)%sn,ncells,ictrlr)
        CALL bfinn(g1dAr(cci)%concn,ncells,ictrlr)
      ENDIF
      IF (nods.GT.0) CALL bfinn(g1dAr(cci)%qppc,ncells,ictrlr)
      CALL bfinn(g1dAr(cci)%yyyyyn,ncells,ictrlr)
```

Coding Requirement: The order of the calls to `bfoutn` in subroutine `dcomp` must match the order of calls to `bfinn` in subroutine `recomp`.

Accessing g1DAr (hydropt and dualpt) Array Data

In the hydrodynamic calling chain:

Subroutine `tf1d` (module `Gen1DTask`) calls subroutine `tf1ds` (module `Gen1DCrunch`) using array `g1dAr`:

```
                  SUBROUTINE tf1d(bd1,bd2,ncl)
                  ---
                  ---
                  ---
      CALL tf1ds(g1dAr(cco)%alp,.........................&
&      .....................,g1dAr(cco)%fa,...........&
&     ,g1dAr(cco)%qrl,g1dAr(cco)%qrv)
                  ---
                  ---
                  ---
```

Subroutine `tf1ds` has the same dummy-argument list as in TRAC-P but declares the dummy arguments with `(:)` notation:

```
                  SUBROUTINE
tf1ds(alp,.............................&
&     ,.................,fa,.........................&
&     ,qrl,qrv)
                  ---
                  ---
                  ---
```

```
                REAL(sdk)
alp(:),.................................&
                ---
                ---
                ---
                REAL(sdk).........,fa(:),.......qrl(:),qrv(:)
```

and indexing into the dummy-argument arrays remains as in TRAC-P, e.g.,

```
alp(jstart)

fa(jp)

qrl(j)

qrv(j)
```

Module Gen1DArray Data Interface (for noninstantiated components)

Module Gen1DArray contains all the logic for providing generic 1D array data from noninstantiated components (such as for use by the Control System). This logic comprises all of the needed definitions and declarations, including declaration and setting of array indices, at the start of the module:

```
                .
                MODULE Gen1DArray
                ---
                ---
                ---
                INTEGER(sik), PARAMETER ::   hgamInd=  1
                ---
                ---
                ---
                INTEGER(sik), PARAMETER ::     tlInd= 22
                INTEGER(sik), PARAMETER ::    tlnInd= 23
                ---
                ---
                ---
                INTEGER(sik), PARAMETER ::     faInd= 66
!
                INTEGER(sik),PARAMETER :: num1DFaceArrays=66
!
                TYPE array1DPtrT
                  LOGICAL :: isAssociated
                  REAL(sdk),POINTER,DIMENSION(:) :: array1DPtr
                END TYPE array1DPtrT
!
```

```
                TYPE arrayNodeT
!       Array(comp index) of Pointers to this 1-D Array
        TYPE(array1DPtrT),DIMENSION(maxComps) :: array1DPtrs
                END TYPE arrayNodeT
!
                TYPE(arrayNodeT),DIMENSION(num1DFaceArrays)   ::
faceArs
!
                ---
                ---
                ---
```

This is followed by initialization at the end of subroutine `AllocGen1D`:

```
                SUBROUTINE AllocGen1D(........)
                ---
                ---
                ---
!     Initialize 1-D interface pointers:
      DO nv=1,num1DFaceArrays
        nullify(faceArs(nv)%array1DPtrs(cci)%array1DPtr)
        faceArs(nv)%array1DPtrs(cci)%isAssociated=.FALSE.
      ENDDO
!
      RETURN
      END SUBROUTINE AllocGen1D
                ---
                ---
                ---
```

This is followed by the data interface routines that are called from elsewhere in the code, which in turn are followed by worker routines that the interface routines use to provide the actual requested information:

Data Interface Routines:

```
REAL(sdkx) FUNCTION GetEosDriv1d
REAL(sdkx) FUNCTION GetGen1D
REAL(sdkx) FUNCTION GetGen1D2D
SUBROUTINE GetGen1DArray
SUBROUTINE CopyGen1DArray
SUBROUTINE IncrementGen1D
```

Worker Routines:

```
SUBROUTINE Get1DArrayPointer
SUBROUTINE Get2DArrayPointer
```

Code fragments from these routines are listed here, followed by guidelines for adding a new general 1D array to the data interface.

```
-----------------------------------------------------------------------------------------------
      REAL(sdkx) FUNCTION GetEosDriv1d(compInd,arrayName,cell)
!
!     Extract scalar from inverted derivative container driv
       ---
       ---
       ---
      ordInd = compIndices(compInd)
!
      IF      (arrayName.EQ.'drvdt    ') THEN
        index=9
      ELSE IF(arrayName.EQ.'dr1dt    ') THEN
        index=8
      ELSE IF(arrayName.EQ.'hvst     ') THEN
        index=10
      ELSE IF(arrayName.EQ.'hlst     ') THEN
        index=11
      ENDIF
!
!     Inverted DataBase
      index=index+(cell-1)*nthm
      GetEosDriv1d=g1DAr(ordInd)%driv(index)
!
      END FUNCTION GetEosDriv1d
-----------------------------------------------------------------------------------------------


-----------------------------------------------------------------------------------------------
      REAL(sdkx) FUNCTION GetGen1D(compInd,arrayInd,cell)
!
      IMPLICIT NONE
!
      INTEGER(sik),INTENT(IN) :: compInd,cell,arrayInd
      REAL(sdk), POINTER, DIMENSION(:) :: arPtr
!
       CALL Get1DArrayPointer(arrayInd,compInd,arPtr)
      GetGen1D=arPtr(cell)
!
      END FUNCTION GetGen1D
-----------------------------------------------------------------------------------------------


-----------------------------------------------------------------------------------------------
      REAL(sdkx) FUNCTION GetGen1D2D(compInd,arrayName,i,j)
!
      IMPLICIT NONE
!
      INTEGER(sik),INTENT(IN) :: compInd,i,j
      CHARACTER*8 arrayName
      REAL(sdk), POINTER, DIMENSION(:,:) :: arPtr
!
```

```
      CALL Get2DArrayPointer(arrayName,compInd,arPtr)
      GetGen1D2D=arPtr(i,j)
!
      END FUNCTION GetGen1D2D
-------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------
      SUBROUTINE GetGen1DArray(compInd,arrayInd,arPtr,ncells)
!
      IMPLICIT NONE
!
      INTEGER(sik),INTENT(IN) :: ncells,arrayInd
      REAL(sdk), POINTER, DIMENSION(:) :: arPtr
      INTEGER(sik) compInd
!
       CALL Get1DArrayPointer(arrayInd,compInd,arPtr)
!
      END SUBROUTINE GetGen1DArray
-------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------
      SUBROUTINE CopyGen1DArray(compInd,arrayInd,array,ncells)
!
      IMPLICIT NONE
!
      INTEGER(sik),INTENT(IN) :: ncells,arrayInd
      REAL(sdk), DIMENSION(ncells) :: array
      REAL(sdk), POINTER, DIMENSION(:) :: arPtr
      INTEGER(sik) compInd,i
!
       CALL Get1DArrayPointer(arrayInd,compInd,arPtr)
      DO i=1,ncells
        array(i)=arPtr(i)
      ENDDO
!
      END SUBROUTINE CopyGen1DArray
-------------------------------------------------------------------------------------------------

-------------------------------------------------------------------------------------------------
      SUBROUTINE IncrementGen1D(compInd,arrayInd,cell,value)
!
      IMPLICIT NONE
!
      INTEGER(sik),INTENT(IN) :: compInd,cell,arrayInd
      REAL(sdk),INTENT(IN) :: value
      REAL(sdk), POINTER, DIMENSION(:) :: arPtr
!
       CALL Get1DArrayPointer(arrayInd,compInd,arPtr)
      arPtr(cell)=arPtr(cell)+value
!
      END SUBROUTINE IncrementGen1D
-------------------------------------------------------------------------------------------------
```

```
      SUBROUTINE Get1DArrayPointer(arrayInd,compInd,arPtr)
       ---
       ---
       ---
      INTEGER(sik),INTENT(IN) :: compInd,arrayInd
      INTEGER(sik) :: ordInd
      REAL(sdk), POINTER, DIMENSION(:) :: arPtr
!
      ordInd = compIndices(compInd)
!
      IF(faceArs(arrayInd)%array1DPtrs(ordInd)%isAssociated) THEN
        arPtr=>faceArs(arrayInd)%array1DPtrs(ordInd)%array1DPtr
      ELSE
!
      SELECT CASE (arrayInd)
!
      CASE (hgamInd)
        arPtr=>g1dAr(ordInd)%hgam
      CASE (hlaInd)
        arPtr=>g1dAr(ordInd)%hla
      CASE (hvaInd)
        arPtr=>g1dAr(ordInd)%hva
       ---
       ---
       ---
       CASE (alvnInd)
        arPtr=>g1dAr(ordInd)%alvn
      CASE (faInd)
        arPtr=>g1dAr(ordInd)%fa
!
      CASE DEFAULT
!
        PRINT *, "BAD NAME INDEX TO Gen1dArray:Get1DArrayPointer"
        STOP
      END SELECT
!
      faceArs(arrayInd)%array1DPtrs(ordInd)%array1DPtr=>arPtr
      faceArs(arrayInd)%array1DPtrs(ordInd)%isAssociated=.TRUE.
!
      ENDIF
!
      RETURN
      END SUBROUTINE Get1DArrayPointer
```

```
      SUBROUTINE Get2DArrayPointer(arrayName,compInd,arPtr)
!
!     BEGIN MODULE USE
      USE Global
!
      IMPLICIT NONE
      CHARACTER*8 arrayName
      INTEGER(sik) compInd,ordInd
```

```
      REAL(sdk), POINTER, DIMENSION(:,:) :: arPtr
!
      ordInd = compIndices(compInd)
!
      IF      (arrayName.EQ.'twn     ') THEN
        arPtr=>g1dAr(ordInd)%twn
      ELSE
        PRINT *, "BAD ARRAY NAME TO Gen1dArray:Get2DArrayPointer"
        PRINT *, arrayName
        STOP
      ENDIF
!
      RETURN
      END SUBROUTINE Get2DArrayPointer
```

If the new general array xxxxx is to be made available to the Gen1DArray data interface:

1. Add a new index variable for the array at the end of the current list of index declarations at the start of module Gen1DArray, and increase parameter num1DFaceArrays accordingly:

```
      MODULE Gen1DArray
      ---
      ---
      ---
      INTEGER(sik), PARAMETER ::    hgamInd=  1
      ---
      ---
      ---
      INTEGER(sik), PARAMETER ::      tlInd= 22
      INTEGER(sik), PARAMETER ::     tlnInd= 23
      ---
      ---
      ---
      INTEGER(sik), PARAMETER ::      faInd= 66
      INTEGER(sik), PARAMETER ::  xxxxxInd= 67
!
      INTEGER(sik),PARAMETER :: num1DFaceArrays=67
!
```

2. Add a new case for array xxxxx to subroutine Get1DArrayPointer:

```
   SUBROUTINE Get1DArrayPointer(arrayInd,compInd,arPtr)
      ---
      ---
      ---
      SELECT CASE (arrayInd)
!
```

```
      CASE (hgamInd)
        arPtr=>g1dAr(ordInd)%hgam
      ---
      ---
      ---
      CASE (faInd)
        arPtr=>g1dAr(ordInd)%fa
                  CASE (xxxxxInd)
           arPtr=>g1dAr(ordInd)%xxxxx
!
      CASE DEFAULT
!
        PRINT *, "BAD NAME INDEX TO Gen1dArray:Get1DArrayPointer"
        STOP
      END SELECT
      ---
      ---
      ---
      END SUBROUTINE Get1DArrayPointer
```

Coding Standard: For maintainability, any new case added to subroutine `Get1DArrayPointer` should be appended to the end of the current list of cases.

G.1.4.2. 3D Vessel-Component Arrays

Special Arrays

The 3D Vessel Special Arrays are stored in array `vsAr`, which is of derived-type `vessArrayT`. The elements of type `vessArrayT` are defined in module `VessArray`, and array `vsAr` is declared there to be of dimension(`maxComps`):

```
      MODULE VessArray
!
!     BEGIN MODULE USE
      USE IntrType
      USE GlobalDim
!
      IMPLICIT NONE
!
!     Vessel component specific arrays
!
      TYPE vessArrayT                          <<<--- type vessArrayT
        REAL(sdk), POINTER, DIMENSION(:) :: z
        REAL(sdk), POINTER, DIMENSION(:) :: dz
      ---
      ---
      ---
        INTEGER(sik), POINTER, DIMENSION(:) :: jsn
        INTEGER(sik), POINTER, DIMENSION(:) :: jsnget
```

```
      INTEGER(sik), POINTER, DIMENSION(:) :: jsnput

    ---
    ---
    ---
      REAL(sdk), POINTER, DIMENSION(:) :: esm
      REAL(sdk), POINTER, DIMENSION(:) :: evsm
      INTEGER(sik), POINTER, DIMENSION(:) :: nfcvsm
      INTEGER(sik), POINTER, DIMENSION(:) :: nfclsm
    ---
    ---
    ---

      REAL(sdk), POINTER, DIMENSION(:) :: ztbn
!
    END TYPE vessArrayT
!
    TYPE (vessArrayT),DIMENSION(maxComps) :: vsAr <<<--- declare vsAr
```

Storage is allocated for the individual arrays in array `vsAr` by calls to `TRACAllo` by subroutine `AllocVess`, which is also in module `VessArray`:

```
    MODULE VessArray
    ---
    ---
    ---
    CONTAINS

    SUBROUTINE AllocVess
    ---
    ---
    ---
    nclx=vessTab(cci)%nclx <<<--- local variables used in  TRACAllo CALLs
    nrsx=vessTab(cci)%nrsx
    nytv=vessTab(cci)%nytv
    nasx=vessTab(cci)%nasx
    ntsx=vessTab(cci)%ntsx
    ncsr=vessTab(cci)%ncsr
    nvent=vessTab(cci)%nvent
    ---
    ---
    ---
    CALL TRACAllo(vsAr(cci)%ztbn,nclx,'ztbn',0.0d0)
    CALL TRACAllo(vsAr(cci)%zchfn,nclx,'zchfn',0.0d0)
    CALL TRACAllo(vsAr(cci)%alptn,nclx,'alptn',0.0d0)
    ---
    ---
    ---
```

Subroutine `AllocVess` is called once each by subroutines `rvssl` and `revssl`.

Coding Standard: For maintainability, the order of calls to TRACAllo in subroutine AllocVess should match the (reversed) order of array declarations in derived data-type vessArrayT.

Read the Special Array from the text input file, and echo the read to the text output file with calls to service routines loadn and warray:

Subroutine rvssl (module VessTask):

```
      ---
      ---
      ---
      CALL loadn(vsAr(cci)%z,vessTab(cci)%nasx,1)
      CALL warray('z       ',vsAr(cci)%z,vessTab(cci)%nasx,0)
!
      CALL loadn(vsAr(cci)%rad,vessTab(cci)%nrsx,1)
      IF (vessTab(cci)%igeom.EQ.0) THEN
        CALL warray('r       ',vsAr(cci)%rad,vessTab(cci)%nrsx,0)
      ELSE
        CALL warray('x       ',vsAr(cci)%rad,vessTab(cci)%nrsx,0)
      ENDIF
!
      CALL loadn(vsAr(cci)%th,vessTab(cci)%ntsx,1)
      IF ((vessTab(cci)%igeom.EQ.0).AND.(ioinp.EQ.0).AND                    &
     &.(vsAr(cci)%th(vessTab(cci)%ntsx).GT.2.0d0*pi+0.0001d0)) THEN
        DO i=1,vessTab(cci)%ntsx
          vsAr(cci)%th(i)=(pi/180.0d0)*vsAr(cci)%th(i)
        ENDDO
      ENDIF
      IF (vessTab(cci)%igeom.EQ.0) THEN
        CALL warray('t       ',vsAr(cci)%th,vessTab(cci)%ntsx,0)
      ELSE
        CALL warray('y       ',vsAr(cci)%th,vessTab(cci)%ntsx,0)
      ENDIF
      ---
      ---
      ---
```

Add the Special Array to the dump file, read it from the restart file, and echo the read to the text output file. This is done with calls to service routines bfoutn, bfinn, and warray:

Subroutine dvssl (module VessTask):

```
      ---
      ---
      ---
      CALL bfoutn(vsAr(cco)%z,vessTab(cco)%nasx,ictrld)
      CALL bfoutn(vsAr(cco)%rad,vessTab(cco)%nrsx,ictrld)
      CALL bfoutn(vsAr(cco)%th,vessTab(cco)%ntsx,ictrld)
      ---
```

```
---
---
```

Subrountine revssl (module VessTask):

```
---
---
---
CALL bfinn(vsAr(cci)%z,vessTab(cci)%nasx,ictrlr)
CALL bfinn(vsAr(cci)%rad,vessTab(cci)%nrsx,ictrlr)
CALL bfinn(vsAr(cci)%th,vessTab(cci)%ntsx,ictrlr)
CALL warray('z       ',vsAr(cci)%z,vessTab(cci)%nasx,0)
IF (vessTab(cci)%igeom.EQ.0) THEN
  CALL warray('r       ',vsAr(cci)%rad,vessTab(cci)%nrsx,0)
  CALL warray('t       ',vsAr(cci)%th,vessTab(cci)%ntsx,0)
ELSE
  CALL warray('x       ',vsAr(cci)%rad,vessTab(cci)%nrsx,0)
  CALL warray('y       ',vsAr(cci)%th,vessTab(cci)%ntsx,0)
ENDIF
---
---
---
```

Fluid-Mesh Arrays

The 3D Vessel Fluid-Mesh Arrays are stored in array vsAr3, which is of derived-type vessArray3T. The elements of type vessArray3T are defined in module VessArray3, and array vsAr3 is declared in VessArray3 to be of dimension(maxComps). Storage is allocated for the individual arrays in array vsAr3 by calls to TRACAllo by subroutine AllocVess3, which is also in module VessArray3.

```
MODULE VessArray3
---
---
---
TYPE vessArray3T
  REAL(sdk), POINTER, DIMENSION(:,:,:) :: hla
  REAL(sdk), POINTER, DIMENSION(:,:,:) :: hva
  REAL(sdk), POINTER, DIMENSION(:,:,:) :: q3drl
  REAL(sdk), POINTER, DIMENSION(:,:,:) :: q3drv
---
---
---
  REAL(sdk), POINTER, DIMENSION(:,:,:) :: xv4
  REAL(sdk), POINTER, DIMENSION(:,:,:) :: xv5
  REAL(sdk), POINTER, DIMENSION(:,:,:) :: xv6
  REAL(sdk), POINTER, DIMENSION(:,:,:) :: xvs
END TYPE vessArray3T
!
TYPE (vessArray3T),DIMENSION(maxComps) :: vsAr3
!
```

```
      CONTAINS

      SUBROUTINE AllocVess3(ni,nj,nk,ccix)
!
!     BEGIN MODULE USE
      USE Alloc
!
      IMPLICIT NONE
      INTEGER(sik) ni,nj,nk,ccix
!
      CALL TRACAllo(vsAr3(ccix)%hla,ni,nj,nk,'hla',0.0d0)
      CALL TRACAllo(vsAr3(ccix)%hva,ni,nj,nk,'hva',0.0d0)
      CALL TRACAllo(vsAr3(ccix)%q3drl,ni,nj,nk,'q3drl',0.0d0)
      CALL TRACAllo(vsAr3(ccix)%q3drv,ni,nj,nk,'q3drv',0.0d0)
      ---
      ---
      ---
      CALL TRACAllo(vsAr3(ccix)%xv4,ni,nj,nk,'xv4',0.0d0)
      CALL TRACAllo(vsAr3(ccix)%xv5,ni,nj,nk,'xv5',0.0d0)
      CALL TRACAllo(vsAr3(ccix)%xv6,ni,nj,nk,'xv6',0.0d0)
      CALL TRACAllo(vsAr3(ccix)%xvs,ni,nj,nk,'xvs',0.0d0)
!
      END SUBROUTINE AllocVess3
```

Subroutine `AllocVess3` is called once each by subroutines `rvssl` and `revssl`. Note that `AllocVess3` obtains its dimensioning information through its argument list, unlike subroutine `AllocVess`, which uses data in `VessTab`. Another difference between the two allocation routines is in `AllocVess3`'s use of argument variable `ccix` to index a specific VESSEL component in array `vsAr3` (this will support future parallelization of TRAC).

Many of the Vessel fluid-mesh arrays are dual time (they contain either old- or new-time values of the same quantity), e.g.,

```
REAL(sdk), POINTER, DIMENSION(:,:,:) :: tv <<<--- old vapor temp
REAL(sdk), POINTER, DIMENSION(:,:,:) :: tl
REAL(sdk), POINTER, DIMENSION(:,:,:) :: gam
REAL(sdk), POINTER, DIMENSION(:,:,:) :: p  <<<--- old pressure
---
---
---
REAL(sdk), POINTER, DIMENSION(:,:,:) :: tvn <<<--- new vapor temp
REAL(sdk), POINTER, DIMENSION(:,:,:) :: tln
REAL(sdk), POINTER, DIMENSION(:,:,:) :: gamn
REAL(sdk), POINTER, DIMENSION(:,:,:) :: pn  <<<--- new pressure
```

Coding Standard: For maintainability, declarations of new dual-time array elements that are added to the VESSEL-component, derived-type `vessArray3T` should be included in the appropriate section of `vessArray3T`, old-time variables should be included with the

other old-time variables, and new-time variables should be included with other new-time variables.

Coding Standard: For maintainability, the order of calls to TRACAllo in subroutine AllocVess3 should match the order of array declarations in derived data-type vessArray3T.

If necessary, read the Fluid-Mesh Array from the text input file and echo the read to the text output file with calls to service routines loadn, clearn, and rlevel; the arrays are first read into scratch array scr, grouped by axial level (i,j planes), within a loop over the axial (k) coordinate:

Subroutine rvssl (module VessTask):

```
      ---
      ---
      ---
      ALLOCATE(scr(vessTab(cci)%nclx))
      ---
      ---
      ---
      IF (tlq.NE.1.0d20) inproc=2
      CALL loadn(scr,vessTab(cci)%nclx,1)
      IF ((istopt.NE.0).AND.(tlq.NE.1.0d20)) THEN  <<<-- istopt option
        tmp=tlq
        IF (ioinp.EQ.1) CALL uncnvts('tl',tmp,1,1,-1)
        CALL clearn(tmp,scr,vessTab(cci)%nclx)
      ENDIF
      CALL rlevel('tln',scr,vessTab(cci)%nclx,vsAr3(cci)%tln)
```

The level-input logic supports the feature to input default values for certain data arrays, under control of namelist variable istopt.

There is also logic at the end of subroutine rvssl for the "repeat level" Vessel input feature:

```
        READ (card,710) nlev
  710   FORMAT (14x,i14)
        IF ((nlev.GE.1-vessTab(cci)%igbcz).AND.(nlev.LT.nas)) THEN
          IF (inlab.EQ.3) WRITE (inlab,725) nlev
  725     FORMAT (1h*/12hrepeat level,i4/1h*)
!
      ksnk=kc
      ksrc=nlev+nzbcm
      DO i=vessTab(cci)%ic0mm,vessTab(cci)%iall
        DO j=vessTab(cci)%jc0mm,vessTab(cci)%jall
!
          vsAr3(cci)%alp(i,j,ksnk)=vsAr3(cci)%alp(i,j,ksrc)
          vsAr3(cci)%alpn(i,j,ksnk)=vsAr3(cci)%alpn(i,j,ksrc)
          vsAr3(cci)%hdyt(i,j,ksnk)=vsAr3(cci)%hdyt(i,j,ksrc)
          vsAr3(cci)%hdz(i,j,ksnk)=vsAr3(cci)%hdz(i,j,ksrc)
      ---
```

```
---
---
```

Add the Fluid-Mesh Array to the dump file, read it from the restart file, and echo the read to the text output file with calls to service routines `dlevel`, `bfinn`, `warray`, and `levelr`:

Subroutine `dvssl` (module `VessTask`):

```
                  ---
                  ---
                  ---
!     level data
!
      DO k=k0,kx
        iz=k-nzbcm
        CALL dlevel(vsAr3(cco)%cfzlyt,vessTab(cco)%nclx)
        CALL dlevel(vsAr3(cco)%cfzlz,vessTab(cco)%nclx)
        CALL dlevel(vsAr3(cco)%cfzlxr,vessTab(cco)%nclx)
        CALL dlevel(vsAr3(cco)%cfzvyt,vessTab(cco)%nclx)
        CALL dlevel(vsAr3(cco)%cfzvz,vessTab(cco)%nclx)
                  ---
                  ---
                  ---
```

Subroutine `revssl` (module `VessTask`):

```
                  ---
                  ---
                  ---
!     read level array data
!
      IF (vessTab(cci)%igbcz.EQ.0) THEN
        k0=vessTab(cci)%kc0
        kx=vessTab(cci)%kcx
      ELSE
        k0=vessTab(cci)%kc0m
        kx=vessTab(cci)%kcxp
      ENDIF
!
      ALLOCATE(scr(vessTab(cci)%nclx))    <<<--- scratch array scr
!
      DO k=k0,kx
        nas=k-nzbcm
        iz=nas
        WRITE (iout,140) nas
  140   FORMAT (/' level',i3,' data')
        CALL bfinn(scr,vessTab(cci)%nclx,ictrlr)
        CALL warray('cfzlyt  ',scr,vessTab(cci)%nclx,0)
        CALL levelr(vsAr3(cci)%cfzlyt,scr)
!
```

```
        CALL bfinn(scr,vessTab(cci)%nclx,ictrlr)
        CALL warray('cfzlz   ',scr,vessTab(cci)%nclx,0)
        CALL levelr(vsAr3(cci)%cfzlz,scr)
!
        CALL bfinn(scr,vessTab(cci)%nclx,ictrlr)
        CALL warray('cfzlxr  ',scr,vessTab(cci)%nclx,0)
        CALL levelr(vsAr3(cci)%cfzlxr,scr)
!
        CALL bfinn(scr,vessTab(cci)%nclx,ictrlr)
        CALL warray('cfzvyt  ',scr,vessTab(cci)%nclx,0)
        CALL levelr(vsAr3(cci)%cfzvyt,scr)
!
        CALL bfinn(scr,vessTab(cci)%nclx,ictrlr)
        CALL warray('cfzvz   ',scr,vessTab(cci)%nclx,0)
        CALL levelr(vsAr3(cci)%cfzvz,scr)
                    ---
                    ---
                    ---
```

If appropriate, add a time edit of the array to the text output file `trcout`, with a call to service routine `wlevel` (which calls `leveli` to stack the rank-3 arrays into a temporary rank-1 array for printing):

Subroutine `wvssl` (module `VessTask`):

```
                    ---
                    ---
                    ---
                    k0=vessTab(cco)%kc0
      kx=vessTab(cco)%kcx
      IF (vessTab(cco)%igbcz.NE.0) k0=vessTab(cco)%kc0m
      IF (vessTab(cco)%igbcz.NE.0) kx=vessTab(cco)%kcxp
!
      DO kc=k0,kx
        iz=kc-nzbcm
!
        WRITE (iout,110) iz
  110   FORMAT (/' level',i3,' data')
!
                    ---
                    ---
                    ---
        CALL wlevel('alpn    ',vsAr3(cco)%alpn,vessTab(cco)%nclx)
        CALL wlevel('rovn    ',vsAr3(cco)%rovn,vessTab(cco)%nclx)
        IF (ntsprn.NE.0) CALL wlevel('arvn    ',vsAr3(cco)%arvn          &
     &  ,vessTab(cco)%nclx)
        CALL wlevel('roln    ',vsAr3(cco)%roln,vessTab(cco)%nclx)
        IF (ntsprn.NE.0) CALL wlevel('arln    ',vsAr3(cco)%arln          &
     &  ,vessTab(cco)%nclx)
        CALL wlevel('vvnyt   ',vsAr3(cco)%vvnyt,vessTab(cco)%nclx)
        IF (ntsprn.NE.0) CALL wlevel('vvntyt  ',vsAr3(cco)%vvntyt        &
     &  ,vessTab(cco)%nclx)
        CALL wlevel('vvnz    ',vsAr3(cco)%vvnz,vessTab(cco)%nclx)
        IF (ntsprn.NE.0) CALL wlevel('vvntz   ',vsAr3(cco)%vvntz         &
```

```
     &  ,vessTab(cco)%nclx)
                    ---
                    ---
                    ---
```

If the new array is dual time, add a new-time to old-time data copy to subroutine `timupd` (for timestep advancement):

Subroutine `timupd` (module `VessCrunch`):

```
                    ---
                    ---
                    ---
!      over-writes start of time step variables with end of time
!      step values for one vessel level
!
       IF (im100x.NE.-100) THEN
!
         k=iz+nzbcm
         DO i=vessTab(cco)%ic0m,vessTab(cco)%icx
           DO j=vessTab(cco)%jc0m,vessTab(cco)%jcx
             vsAr3(cco)%alpo(i,j,k)=vsAr3(cco)%alp(i,j,k)
!
             vsAr3(cco)%bit(i,j,k)=vsAr3(cco)%bitn(i,j,k)
             vsAr3(cco)%frci1(i,j,k)=vsAr3(cco)%frci1n(i,j,k)
             vsAr3(cco)%frci2(i,j,k)=vsAr3(cco)%frci2n(i,j,k)
             vsAr3(cco)%frci3(i,j,k)=vsAr3(cco)%frci3n(i,j,k)
             vsAr3(cco)%ciyt(i,j,k)=vsAr3(cco)%cinyt(i,j,k)
             vsAr3(cco)%ciz(i,j,k)=vsAr3(cco)%cinz(i,j,k)
             vsAr3(cco)%cixr(i,j,k)=vsAr3(cco)%cinxr(i,j,k)
                    ---
                    ---
                    ---
                    vsAr3(cco)%owlz(i,j,k)=vsAr3(cco)%wlz(i,j,k)
             vsAr3(cco)%owlxr(i,j,k)=vsAr3(cco)%wlxr(i,j,k)
           ENDDO
         ENDDO
       ENDIF
!
!
       RETURN
       END SUBROUTINE timupd
```

Old-time to new-time data copies for timestep backups are handled by subroutine `bakup`. Copying for special (i.e., water-packer) backups is called from the `outer` stage and for normal backups from POST; as with the 1D-component, dual-time arrays, care must be taken with the old-time/new-time value of the array for the water-packing logic (i.e., add the copy only if the value of the array does not change in the PREP stage).

Subroutine `bakup` (module `VessCrunch`):

```
       SUBROUTINE bakup(iopt)   <<<--- iopt is flag for type of backup
       ---
```

```
      ---
      ---
!     over-writes end of time step variables with start of time
!     step values for one vessel level
!
      k=iz+nzbcm
!
      IF (iopt.EQ.0) THEN
!
!     backup from the post stage
!
        DO i=vessTab(cco)%ic0mm,vessTab(cco)%iall
          DO j=vessTab(cco)%jc0mm,vessTab(cco)%jall
            vsAr3(cco)%bitn(i,j,k)=vsAr3(cco)%bit(i,j,k)
            vsAr3(cco)%frci1n(i,j,k)=vsAr3(cco)%frci1(i,j,k)
            vsAr3(cco)%frci2n(i,j,k)=vsAr3(cco)%frci2(i,j,k)
            vsAr3(cco)%frci3n(i,j,k)=vsAr3(cco)%frci3(i,j,k)
      ---
      ---
      ---
                    vsAr3(cco)%wvxr(i,j,k)=vsAr3(cco)%owvxr(i,j,k)
            vsAr3(cco)%wlyt(i,j,k)=vsAr3(cco)%owlyt(i,j,k)
            vsAr3(cco)%wlz(i,j,k)=vsAr3(cco)%owlz(i,j,k)
            vsAr3(cco)%wlxr(i,j,k)=vsAr3(cco)%owlxr(i,j,k)
          ENDDO
        ENDDO
      ELSE
!
!     backup from the outer stage
!
        DO i=vessTab(cco)%ic0mm,vessTab(cco)%iall
          DO j=vessTab(cco)%jc0mm,vessTab(cco)%jall
            vsAr3(cco)%chtin(i,j,k)=vsAr3(cco)%chti(i,j,k)
            vsAr3(cco)%chtan(i,j,k)=vsAr3(cco)%chtia(i,j,k)
            vsAr3(cco)%alvn(i,j,k)=vsAr3(cco)%alv(i,j,k)
            vsAr3(cco)%alven(i,j,k)=vsAr3(cco)%alve(i,j,k)
      ---
      ---
      ---
                    vsAr3(cco)%wlyt(i,j,k)=vsAr3(cco)%owlyt(i,j,k)
            vsAr3(cco)%wlz(i,j,k)=vsAr3(cco)%owlz(i,j,k)
            vsAr3(cco)%wlxr(i,j,k)=vsAr3(cco)%owlxr(i,j,k)
          ENDDO
        ENDDO
      ENDIF
      RETURN
      END SUBROUTINE bakup
```

Vessel Data Interface (Noninstantiated VESSEL Components)

Function GetVSAR, in module VessArray3, is a data-access routine that returns a value from a subset of the Vessel 3D mesh arrays. GetVSAR takes as input arguments a character string specifying the requested array name, the Vessel's cco index, and the (i, j, k) indices into the array; it operates with IF-THEN-ELSEIF tests on the array name:

```
                MODULE VessArray
                ---
                ---
                ---
                CONTAINS
                ---
                ---
                ---
      REAL(sdkx) FUNCTION GetVSAR(varName,cco,i,j,k)
                ---
                ---
                ---
      INTEGER(sik) i,j,k,cco
      CHARACTER*8 varName
      LOGICAL debug
      DATA debug /.FALSE./
!
      IF (debug) THEN
        WRITE(*,*) 'getVSAR called for ',varName
      ENDIF
!
      IF (varName.EQ.'vvnz    ') THEN
        getVSAR=vsAr3(cco)%vvnz(i,j,k)
      ELSEIF (varName.EQ.'cinz    ') THEN
        getVSAR=vsAr3(cco)%cinz(i,j,k)
      ELSEIF (varName.EQ.'hfg     ') THEN
        getVSAR=vsAr3(cco)%hfg(i,j,k)
      ELSEIF (varName.EQ.'vvxr    ') THEN
        getVSAR=vsAr3(cco)%vvxr(i,j,k)
      ELSEIF (varName.EQ.'chtin   ') THEN
        getVSAR=vsAr3(cco)%chtin(i,j,k)
                ---
                ---
                ---
      ELSEIF (varName.EQ.'vlnz    ') THEN
        getVSAR=vsAr3(cco)%vlnz(i,j,k)
      ELSEIF (varName.EQ.'vlxr    ') THEN
        getVSAR=vsAr3(cco)%vlxr(i,j,k)
      ELSEIF (varName.EQ.'rom     ') THEN
        getVSAR=vsAr3(cco)%rom(i,j,k)
      ELSEIF (varName.EQ.'vlyt    ') THEN
        getVSAR=vsAr3(cco)%vlyt(i,j,k)
      ELSEIF (varName.EQ.'roan    ') THEN
        getVSAR=vsAr3(cco)%roan(i,j,k)
      ENDIF
!
      RETURN
      END FUNCTION getvsar
!
      END MODULE VessArray3
```

G.1.4.3. System Services

Detailed guidelines for modification of the System Services that support intercomponent communication are given in Section 3.2.3.a.

G.1.5. HTSTR Arrays

The HTSTR arrays are stored in array hsAr, which is of derived-type hsArrayT. The elements of type hsArrayT are defined in module HSArray, and array hsAr is declared in HSArray to be of dimension(maxComps). Array hsAr is given the target attribute, and scalar variable chs is declared to be of type hsArrayT, with the pointer attribute. Variable chs is associated with specific array elements (i.e., with specific HTSTR components) of hsAr; this is done only to keep the physical lengths of certain argument lists short (in particular, to keep the number of continuation lines within a limit of 19).

```
                MODULE HSArray
                ---
                ---
                ---
      TYPE hsArrayT
        REAL(sdk), POINTER, DIMENSION(:)     :: rdpwr
        REAL(sdk), POINTER, DIMENSION(:)     :: rs
        REAL(sdk), POINTER, DIMENSION(:)     :: cpowr
        REAL(sdk), POINTER, DIMENSION(:)     :: hs
        REAL(sdk), POINTER, DIMENSION(:)     :: zpwzt
                ---
                ---
                ---
        REAL(sdk), POINTER, DIMENSION(:,:,:)   :: tween
        REAL(sdk), POINTER, DIMENSION(:,:,:)   :: tweeo
        REAL(sdk), POINTER, DIMENSION(:,:,:)   :: cepwn
        REAL(sdk), POINTER, DIMENSION(:,:,:)   :: cepwo
!
      END TYPE hsArrayT
!
      TYPE (hsArrayT),TARGET,DIMENSION(maxComps) :: hsAr
      TYPE (hsArrayT),POINTER:: chs
```

Coding Standard: For maintainability, declarations of new data array elements that are added to the HTSTR-component, derived-type hsArrayT should be included in the appropriate section of hsArrayT, according to the nature of the new array. The various array types include the general global data (which also includes data "global" to a specific copy), the time-dependent global data (including data specific to a copy), the rod and slab-dependent data, the time-dependent rod data, and the surface-dependent rod data (which have single-time and dual-time portions).

Storage is allocated for the individual arrays in array hsAr for a specific HTSTR component by calls to TRACAllo by subroutine pntrod (module RodTask), which is called once each by subroutines rhtstr and rehtst (both in module RodTask).

Coding Standard: For maintainability, the order of calls to TRACAllo in subroutine pntrod should match the order of array declarations in derived data-type hsArrayT.

New-Time to Old-Time and Old-Time to New-Time Data Copies

Module HSArray contains service subroutines TimeUpHS and TimeUpHS1; each routine transfers some of the dual-time HTSTR data arrays from new-time to old-time hsAr arrays or from old-time to new-time arrays (together they treat all of the dual-time arrays), according to the value of input-flag newToOld. Subroutine TimeUpHS is responsible for the global dual-time arrays, and subroutine TimeUpHS1 is responsible for the dual-time rod and rod-surface data.

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

        SUBROUTINE TimeUpHS(newToOld)
!
!     BEGIN MODULE USE
      USE Global
!
      IMPLICIT NONE
!
      LOGICAL newToOld
!
      IF (newToOld) THEN
        hsAr(cco)%cdg  = hsAr(cco)%cdgn    <<<--- dual-time global arrays
        hsAr(cco)%cdh  = hsAr(cco)%cdhn
        hsAr(cco)%clen = hsAr(cco)%clenn
      ELSE
        hsAr(cco)%cdgn  = hsAr(cco)%cdg
        hsAr(cco)%cdhn  = hsAr(cco)%cdh
        hsAr(cco)%clenn = hsAr(cco)%clen
      ENDIF
!
      END SUBROUTINE TimeUpHS

      SUBROUTINE TimeUpHS1(newToOld)
                    ---
                    ---
                    ---

      IF (newToOld) THEN
        hsAr(cco)%radr=hsAr(cco)%radrn  <<<--- dual-time rod arrays
        hsAr(cco)%drz=hsAr(cco)%drzn
        hsAr(cco)%rft=hsAr(cco)%rftn
!
!        Surface-dependent arrays
        hsAr(cco)%hrflo=hsAr(cco)%hrfl
        hsAr(cco)%hrfvo=hsAr(cco)%hrfv
        hsAr(cco)%hrfgo=hsAr(cco)%hrfg
```

```
      ---
      ---
      ---

        hsAr(cco)%cepwo=hsAr(cco)%cepwn
!
      ELSE                                    <<<--- old to new
!
        hsAr(cco)%radrn=hsAr(cco)%radr
        hsAr(cco)%drzn=hsAr(cco)%drz
        hsAr(cco)%rftn=hsAr(cco)%rft
!
!        Surface-dependent arrays
        hsAr(cco)%hrfl=hsAr(cco)%hrflo
        hsAr(cco)%hrfv=hsAr(cco)%hrfvo
        hsAr(cco)%hrfg=hsAr(cco)%hrfgo
                    ---
                    ---
                    ---
        hsAr(cco)%cepwn=hsAr(cco)%cepwo
!
      ENDIF
!
      END SUBROUTINE TimeUpHS1
```

In the PREP stage, subroutine `htstr1` has a call `TimeUpHS1(.TRUE.)`, and `core1` has a call `TimeUpHS(.TRUE.)` for new-time to old-time array copies for timestep advancement. In the POST stage, subroutine `htstr3` has a call `TimeUpHS(.FALSE)` and a call `TimeUpHS1(.FALSE.)` for old-to-new copies in the case of a timestep backup (`oitno` = -100).

HTSTR Data Interface

The HTSTR data interface comprises the following routines, all of which are contained in module `HSArray`:

```
SUBROUTINE GetHS1DPtr(arrayName,compInd,arPtr)

SUBROUTINE GetHS2DPtr(arrayName,compInd,arPtr)

SUBROUTINE GetHS3DPtr(arrayName,compInd,arPtr)

REAL(sdkx) FUNCTION GetHS(compInd,arrayName,cell)

REAL(sdkx) FUNCTION GetHSSurf(compInd,arrayName,rod,cell)

INTEGER(sikx) FUNCTION GetNoht(compInd,rod)

REAL(sdkx) FUNCTION GetHS2d(compInd,arrayName,rod,cell)
```

```
REAL(sdkx) FUNCTION GetHS3d(compInd,arrayName,rod,cell,node)
```

Coding Standard: Any new rank-2 or rank-3 arrays that are to be part of the standard HTSTR data interface should have the following ordering of information in their columns:

```
REAL(sdkx) FUNCTION GetHS2d(compInd,arrayName,rod,cell)
              ---
              ---
              ---
CALL GetHS2DPtr(arrayName,compInd,arPtr)
GetHS2D=arPtr(cell,rod)                    <<<--- cell, rod
              ---
              ---
              ---
REAL(sdkx) FUNCTION GetHS3d(compInd,arrayName,rod,cell,node)
              ---
              ---
              ---
CALL GetHS3DPtr(arrayName,compInd,arPtr)
GetHS3D=arPtr(node,cell,rod)               <<<--- node, cell, rod
              ---
              ---
              ---
```

G.2. Adding A New XTV Graphics Variable

Variables output to the XTV file are output by component and contain seven attributes: Dimension (scalar, 1D, 2D, 3D), Frequency, Data Position, Length, Degeneracy, Color Mapping, and Special Options. In addition, the following is needed: the conditions under which the variable is output, assurance that the variable is indexed in the English units subsystem, and, for 1D components, knowledge of whether this a generic variable or a component-specific variable. Note that the graphics output is limited to floating point values. Integers may be output by conversion to floating-point values, but the addition of character values requires the use of Auxiliary Component Structures, which is not covered in this appendix.

G.2.1. Understanding Variable Attributes

Dimension: This is simply the order of the variable array (scalar or single valued, 1D, etc.) and should not be confused with the dimension of the component. Note that variables having a dimension greater than that of the component are difficult to add and are beyond the description of this appendix. Currently, general problem information, control blocks, signal variables, and trips are treated graphically as scalar components; breaks, fills, generic heat structures (not the rods and slabs), pipes, plenums, prizers, pumps, tees, and valves are 1D components; rods and slabs are 2D components; and the vessel is the only 3D component. Wall conduction in 1D fluid components promotes them to 2D components when the multinode resolution is employed.

Frequency. Two kinds of variables are output to the graphics file: constants that do not change during the course of the calculation and computed values. Constants are said to be static or time-independent and are output in the graphics header and not in the regular data section. Values computed during the course of a calculation are said to be dynamic or time-dependant and are output in the data section of the graphics file. At some later date, there may be some option for varying frequencies of output (e.g., some values output every edit, some only every third edit).

Data Position. TRAC utilizes a staggered mesh: that is, some values are computed at the cell center and some at the cell face. For the most part, volume and inventory quantities are cell centered, whereas velocity and mass flow quantities are face oriented. Face-oriented arrays generally have more values than cell-centered arrays.

Length. TRAC currently implements arrays with two different classes of length: those of a fixed length and those using a special variant of dynamic sizing. "Dynamically sized" arrays employ a fixed dimension and denote the end of active values with the token "-1" after the last value. The actual number of values in a fixed-dimension array can be computed from the axis dimensions and the data position. For example, if a vessel has dimensions of (2,2,4), then cell-centered arrays would output 2 x 2 x 4 = 16 values per graphics edit. Similarly, arrays being face valued along the radial axis would contain (2+1) x 2 x 4 = 24 values per edit. Dynamically sized arrays currently are handled by outputting all values (including the inactive ones) and then utilizing only the active ones. The actual number of values output to the graphics file is calculated in the same manner as for fixed dimension arrays utilizing the maximum number of elements as the dimension for the dynamically sized axis.

Degeneracy. The degeneracy attribute specifies along which axis(es) this variable lies when the dimension of the component is greater than that of the variable. For the purposes of the graphics file, the dimension of the component is that of the highest-dimensioned component. Thus, 1D fluid components are considered 2D components when they have two or more wall conduction nodes. This information is used to help the application determine the length of the array and construct an appropriate diagram in the graphical user interface (GUI). For example, `stnui`, the Stanton number along the inner surface of the heat structure, is degenerate along the j or z axis. Variables that are the same dimension as the component, as well as scalar variables, have no degeneracy. To keep things simple between cylindrical and Cartesian systems, the axis identifiers i, j, and k are used for the first, second, and third axes, respectively.

Color Mapping. The XTV GUI uses two color sets or maps to help visualize data. Quantities involving fluid properties are visualized using "Water Colors," a series of colors ranging from deep blue to white. Quantities involving heat transfer or energy are displayed using "Hot Colors," a series of colors ranging from black to deep red.

Special Options. The XTV GUI supports four options for special variable treatment. The first is the designation of a variable as a vector component. The second option is the designation of the variable as a tensor component. Both Vector and Tensor components are discussed in more detail in a subsequent section. The third option is the use of inset

display tiles. The 3D displays can have a second value set up for visualization; by selecting it from an inset display menu in the GUI, the inset display attribute adds this variable to the inset display menu. The last option is the Unlisted option. Variables that are used only for dynamic sizing of arrays can be designated as Unlisted, in which case they will not be displayed in the variable selection list of the XTV GUI.

G.2.2. Steps to be Completed before Adding Variables to Output

1. Verify that the variable to be output is indexed in the English Units subsystem (see section on the LABPRG functionality for details). If it is not indexed, add the variable to the LABPRG index.

2. Select the appropriate attributes. The table below lists the variable attribute options. Note that for simplicity in TRAC, the dimension and data position attributes have been hybridized into a composite value.

3. Determine whether the variable should be output conditionally. Some variables are meaningful only if a particular namelist option is selected, others if a particular component option is selected.

4. Determine what subroutine should be responsible for outputting the information. See Appendix C for information as to which subroutines output information on the various components.

5. Verify that the desired length of the variable is equal to what TRAC/XTV will calculate. The rules for calculations are as follows:

    - All 0D variables except tensors output one value.

    - 1D, 2D, and 3D variables that are not dynamically sized output `ncellt` values for cell-centered quantities.

    - Face-valued quantities output one more value along the face axis than its normal dimension, unless that axis is the azimuthal axis on cylinders, in which case it outputs the same number of elements along that axis. For example, a 3D variable that is face valued along the radial axis outputs `(ni+1) * nj nk = ncellt + nj*nk` values. Were it to be face valued along the azimuthal axis, it would output `ni * nj * nk = ncellt` values.

    - Dynamically sized values output the maximum number defined for that component axis. The end of "used values" is denoted by a "-1". If using dynamic sizing for the component, this -1 can be in another variable.

    - Dynamic minimum variables output exactly the minimum number of values for a dynamic axis at all timesteps.

TABLE OF VARIABLE ATTRIBUTE PARAMETERS IN TRAC

Attribute	Value	Description
Dimension-variable position / length	This gives the dimension of indexing used and the relative position of the data in the cell. Subdimensional variables must use the correct value here and specify the degeneracy attribute for proper array length specification. (e.g., `hrfio` is a 1D variable that exists in a 2D heat structure. It is declared here as 1D and described below as lying along the 'J' (z) axis	
	`vScalar`	scalar value (not an array)
	`v1dCc`	1D (linear) array, with values at cell centers
	`v1dFa`	1D (linear) array, with values at cell faces
	`v1dDc`	1D (linear) array, dynamically sized by component
	`v2dCc`	2D array [indexed as (i,j)], with values at cell centers
	v2dFaI	2D array [indexed as (i,j)], with values at cell i faces (i faces are on the first axis, typically the radial ax)
	v2dFaJ	2D array [indexed as (i,j)], with values at cell j faces (j faces are on the second axis, typically the axial axis)
	`v2dDc`	2D array [indexed as (i,j)], dynamically sized by the component
	`v2dDnI`	2D array [indexed as (i,j)], dynamically sized by -1 on the I axis, J axis is fixed dimension
	`v2dDnJ`	2D array [indexed as (i,j)], dynamically sized by -1 on the J axis, I axis is fixed dimension
	`v3dCc`	3D array [indexed as (i,j,k)] with values at cell centers
	`v3dFaI`	3D array [indexed as (i,j,k)] with values at cell j faces
	`v3dFaJ`	3D array [indexed as (i,j,k)) with values at cell j faces
	`v3dFaK`	3D array [indexed as (i,j,k)] with values at cell k faces
Frequency	This provides the frequency of output in the graphics file. Later, options for reduced resolution may be specified (e.g., every other edit, every third edit,etc.)	
	`vStatic`	Time-independent value (output on first edit only)
	`vDynamic`	Time-dependent value (output every edit)
Degeneracy	This specifies which axis (es) the degenerate array lies along. For example, all degenerate rod variables lie along the k (z) axis. This is used for selecting the dimension	

	vDgnI, vDgnJ, vDgnK	The 1D variable specified lies along the i, j, or k axis (respectively)
	vDgnIJ, vDgnIK, vDgnJK	The 2D variable lies on the k, j, or i plane (respectively)
	vNotApp	There is no degeneracy for this component
Color Mapping	This specifies what color set to use for the visualization	
	vWtr	Use water colors (blue to white)
	vHot	Use hot colors (black to red)
Special options	These codes are special-purpose options. See the individual explanations	
	vectI	i component of vector-valued function (i,j,k components must be sequential). Note that the name of the vector must be supplied with the I component variable. This name appears in the vectors submenu of XTV. The rank of the vector is equal to the rank of the component. It exists at as many locations as are defined by the length attribute of the component (thus, 2D components will not have vectK defined
	vectJ	j component of vector-valued function (i,j,k components must be sequential)
	vectK	k component of vector-valued function (i,j,k components must be sequential)
	tensorI <name>*	i row of tensor function (i,j,k must be sequential). Note that the name of the tensor must be supplied with the I component variable. The rank of the tensor is given by the component type. Thus,a scalar tensor of a 3D component has three values
	tensorJ	j row of tensor function (i,j,k must be sequential)
	tensorK	k row of tensor function (i,j,k must be sequential)
	vInsetDisp	(inset display) Display as an inset value (as wall temperatures are now)
	vUnlisted	(unlisted value) Do not place on variable selection list (typically used for variable use for dimensioning dynamic arrays)
	vNotApp	option or attribute not relevant to this variable.

- Vectors output the same number of values as regular variables. What makes them vectors is their association with other variables that provide the other axis components.

- Tensors output n* the number of values for a regular variable. n = the rank of the component; thus, a 3D component outputting a 0D tensor outputs three values for the variable. A 2D component outputting a 1D tensor along the j axis has nj*2 values.

Note that some variable are stored in arrays that do not correspond to any component/cell dimension. For example, all of the reactivity properties of a heat structure are stored in a five-element array. To handle this, five scalar variables are output in the graphics file with five unique names: alreac, dbreac, pgreac, tcreac, and tfreac. Similar constructs may be needed in other cases.

6. Locate what variables and areas of code need to be changed. If the variable is a static or first-edit-only variable, it is entered only in the subroutine where it is output. A call must be added to PrintVarDesc, WriteStaticVx, and LuMatch, and the local count of static variables must be increased, conditionally if necessary. If the variable is out put at each edit, the variable information is entered in xtvinit, and adjustments to the number of variables output by the subroutine and the number of variables of each conditional type must be made. If a new conditional type is introduced, a call to PrintVarDesc will be needed, and under all circumstances, a call to XtvBufx will be needed.

7. Verify that the component that is a parent to the variable being added has the flexibility to handle the type of variable being added. For example, some components output only fluid variables and have the vWtr attribute hardwired. If this is the case, either special handling of the variable or adding an array to handle the attribute for the component is needed.

8. If the variable is a static variable, add the variable immediately after the component header information, along with the other static variable(s) for that component. See the sections below titled PrintVarDesc Interface and WriteStaticVx Interface for further details on how to make the appropriate calls to output the static variable.

9. If the variable is a dynamic variable, add the appropriate entries in XtvInit. Note that it will be necessary to increment nXXVar, where XX represents the appropriate group. If this is a conditional variable, (1) no information about its conditionality is entered in XtvInit. Only enter its name, label, composite dimensionality and length, and occasionally its color mapping attribute or degeneracy. (2) Ensure that the appropriate conditional group is incremented so that the locations of each conditional group in the series are recalculated properly. These variables will be defined at the beginning of the relevant output subroutine. Only the global number of possible variables for the subroutine is a module variable. If this is a new conditional group, an nXXVar parameter and an eXXVar parameter must be added.

nXXVar parameters are used to set the number of variables in a group of conditional variables. For example, most component have a conditional group nIsolutVar=2 because most components conditionally output concn and sn if namelist variable isolut is 1. eXXVar represents the end of a conditional group; therefore, in the header creation section, each conditional group will be represented by a conditional DO loop from eX1Var+1 to eX2Var, where X1 represents the end of the previous group and X2 the end of the current group. Each eXXVar is defined parametrically by the global number of variables and the number of variables in each group. Each component also has a variable defined, 'eUnCndVar,' which is the end of the unconditional variables. Insert the additional logic so that these boundaries remain accurate with the definitions in XtvInit.

10. In the same relative location as the order in XtvInit, insert the call to XtvBufX according to the information in the section the XtvBufX interface.

11. Verify the output in two ways: (1) xtv will attempt to read in both the header description and data values. Both have error detection algorithms built in. (2) If the variables read in properly, examine the values output to ensure that they correspond to what TRAC is calculating. If they do not, either the wrong array is being outputted, or there is a variable order mismatch between the header and the data section. Ensure again that the header order matches the data string order. As an aid, a compile option in module data.c of XTV, ECHO_BINARY will echo what values are assigned to each variable when set to 1.

G.2.3. PrintVarDesc Interface

Subroutine PrintVarDesc is responsible for outputting the variable definition to the graphics file header and takes a total of 10 arguments. The first five arguments are the variable attributes, as defined in the table and in the same order. Currently, Vector, Tensor, InsetDisplay, and Unlisted are mutually exclusive options. The variable attributes are followed by the index into the English units conversion factor table, which is returned from subroutine LuMatch. Arguments 7 and 8 are the variable name and label. Argument 9 is the name of the variable controlling the dynamic sizing. Argument 10 is the name of vector if this is the i component of that vector. Enter blanks if these arguments are not needed. The token vNotApp is appropriate for either the degeneracy attribute or the miscellaneous/special options attribute.

The Degeneracy attribute is needed for the 1D and 2D variables that are associated with components that are at least one rank above the variable (3D is rank 3, etc.). 0D (scalar) variables are not degenerate. This attribute tells the GUI both about the dimensions of the variable and what information to use in creating the display.

Only variables using the options v2dDnI or v2dDnJ need to supply argument 9, the name of the variable supplying the locations of the cell boundaries. Similarly, only variables using the option vectI need to supply argument 10, the name of the vector for the GUI.

G.2.4. `WriteStaticVx` Interface

The `WriteStaticVx` subroutines are used to output 1D and 3D static variables. Static variables are output immediately following their definition. A call to `PrintVarDesc`, which has the vStatic attribute set, should be followed immediately by a call to either `WriteStaticV1` or `WriteStaticV3`. 0D variables can be output as a write statement, with only one value on the line; 2D variables will require a `WriteStaticV2`. The interfaces are given below:

```
SUBROUTINE WriteStaticV1(a1d,nc,uIdx)
      REAL(sdk), INTENT(IN):: a1d(:)
      INTEGER(sik), INTENT(IN):: nc, uIdx

SUBROUTINE WriteStaticV3(a3d,k0,kx,j0,jx,i0,ix,uIdx)
      REAL(sdk), INTENT(IN):: a3d(:,:,:)
      INTEGER(sik), INTENT(IN):: k0,kx,j0,jx,i0,ix,uIdx
```

The `WriteStaticVx` subroutines take three groups of arguments: a pointer to the array, the array bounds, and the units conversion index. The array bounds for the 1D implementation are simply the number of cells to output; `WriteStaticV1` will output value one of the arrays specified through the `ncell` value. Again, as in `PrintVarDesc`, the units conversion index is the value returned from `LuMatch`, described below.

The 3D database includes many boundary cells that usually are not output. Thus, `WriteStaticV3` requires that the start and stop locations in each dimension be specified. Local variables have been created in `XtvVs1` to assist in this. Local variables `k0`, `kf0`, and `kx` are used to define the start and stop along the k or z axis. `kf0` indicates the start of the k-face-oriented values as opposed to `k0`, which indicates the start of the cell-centered values. Similarly, variable `jf0` indicates the beginning of j-face-oriented values. The remaining values can be determined from the `vessTab` variables `ic0`, `icx`, `jc0`, and `jcx`.

G.2.5. `XtvBufx` Interface

The interfaces to the output buffer for each level of variable are

```
SUBROUTINE xtvbufs(scalar,uIdx)
SUBROUTINE xtvbuf1(a1d,nc,uIdx)
SUBROUTINE xtvbuf2(a2d,ni,nj,uIdx)
SUBROUTINE xtvbuf3(a3d,k0,kx,jc0,jcx,ic0,icx,uIdx)
```

The `XtvBufx` subroutines are identical in interface to the `WriteStaticVx` subroutines. The only difference is that the `WriteStaticVx` routines output the values in ASCII format as part of the header, whereas the `XtvBufx` routines output binary floats as part of the data stream. The `XtvBufx` routines take three classes of arguments: a pointer to the array of data, the array bounds, and the units conversion index. The `XtvBufx`

routines have both scalar and 2D implementations that the WriteStaticVx routines currently are lacking due to the current mix of output variables.

G.2.6. LuMatch Interface

The LuMatch subroutine is used to obtain the index of a particular variable into the English Units Conversion Table. It has the following form:

```
SUBROUTINE lumatch(label,uTypeIdx, vNameIdx, list)
      CHARACTER*(*), INTENT(IN) ::label
      INTEGER(sik), INTENT(OUT) :: uTypeIdx, vNameIdx
      INTEGER(sik), INTENT(IN) :: list
```

Label is the name of the variable or the units type that is to be indexed.

uTypeIdx is the index into the list of units types and conversions. This value is used by the XTV output routines to perform English units conversions if the namelist variable iogrf=1.

vNameIdx is the index of the variable in the list of convertible variables. XTV does not use this variable.

List is an enumeration of three options: vNameList, if it is a variable name such as 'vlnxr', uTypeList, if it is the name of a units type such as 'luvolume', or allLists to search both lists.

Generally, LuMatch is called in XtvInit for dynamic variables and immediately before PrintVarDesc for static variables. LuMatch is also used by other subroutines in conjunction with input as well as the TRCOUT file.

APPENDIX H
INPUT AND OUTPUT FILE STRUCTURES

H.1. Dump/Restart

The structure of the dump/restart file is described fully in Section 2.6.3.

Note: XTV/XMGR5 Graphics System. TRAC-M/F90 Version 3.0 includes a now-obsolete version of the XTV/XMGR5 graphics system, which is implemented by Fortran module `Xtv`. Module `Xtv` is to be replaced in a future version of TRAC-M/F90 by modules `CXtvXFaces`, `XtvComps`, `XtvData`, `XtvDump`, and `XtvSetup`. The new implementation of the XTV/XMGR5 logic will include many arrays and derived types, all of which will be defined in module `XtvData`.

H.2. Graphics

XTV previously used a separate ASCII file (`xtvgr.t`, or `file.xtvt`) to describe the contents and format of the binary graphics (`xtvgr.b`, or `file.xtvb`) output. With the expansion of the XTV files to include full TRCGRF equivalency, the format of the XTV file has changed. Foremost among these changes is the consolidation of the two files into one container file (`trcxtv` or `file.xtv`). Currently, this file still consists of an ASCII header section and binary data section, which are delineated by the ASCII string "End of Header Block". By combining these two files into one physical file, the possibility of mismatching multiple header and datafiles is eliminated. This new data format is significantly more generalized than the previous formats, enabling many new components to be added with no changes needed to either the graphics file format or the GUI postprocessor.

H.2.1. Overview of Changes in Version 3.0

- Introduction of a component-type identifier and modifiers to describe the generic type of display needed in XTV (e.g., 1D, 3D), instead of the former component name key. This change allows XTV to handle many new component types to be output and displayed in XTV with no additional coding.

- Expansion of the Face flag to a variable-attributes section that encompasses dimension, size, and output frequency.

- Addition of a synchronization value (1.12345E+12) at the beginning of each timestep to validate file parsing.

- Addition of the capability for XTV (or other postprocessor) to perform unit transformations based on the variable unit type used in TRAC (e.g., `luvolume`, `lulength`, and `lunounit`), as well as a text string label (e.g., m/s) for processors without that functionality.

- Addition of short variable names, as well as the longer descriptive labels for each variable.

- Addition of facilities for auxiliary component structures, which can contain the additional information peculiar to a component type.

- Addition of facilities for possible compression of the data stream at some later date to decrease file size and/or speed transfers over a network.

H.2.2. Summary of XTV Header Format

The XTV header, which is modular, comprises a starting block, followed by a block for each component output to XTV, and finally an End of Header Block. There are only three considerations for ordering in the header file: (1) the data segment must follow the order of the header exactly, (2) substructures (i.e., rods or slabs of heat structures) must follow after the master or parent unit, and (3) the order of appearance in the header determines the order in which the components appear in the Component Selection window in the XTV GUI.

In general, the XTV header is space delimited; exceptions to this are character strings that are of varying length, which are delimited by colons, and sets of values that are delimited by commas for clarity. Character strings had been delimited previously by asterisks; this practice has been discontinued, and the asterisk now is reserved as a component marker.

Starting Block

The starting block has increased dramatically in content from previous versions. The original starting block was a one-line title taken from the first title card supplied through TRACIN. In Version 2.1, this was augmented to support version numbers in addition to the title on the first line. In Version 3.0, the starting block now contains a version identifier, flags for compression and precision, the date and time, machine name, platform type, title, and nondefault unit information to this block.

The first line of the starting block contains all of the information minus the title and additional units information. Previously, there was only one type, and thus, the type/version identifier was the string "XTV-TRAC " followed by a number that was the version of XTV at the time of the modifications to the XTV subroutines in TRAC. Two versions now output the xtv graphics file: a Fortran 77 version (currently Version 5.5.3) and a Fortran 90 version (currently TRAC-M, Version 2.125). The string would be either "XTV-TRAC/f77" if it were written by TRAC-P 5.5.x or "XTV-TRAC/f90" if it were written by the modernized Fortran 90 code. The new version number would be the version of XTV needed to display the file, followed by a revision number to identify uniquely the version of TRAC used (and thus the set of output files used) for the original format.

For example, the version of XTV that can read this new format will be designated 3.0; thus, the version number presented in the header file written by TRAC-P will be "XTV-

TRAC/f77 V3.0.0". If two individual changes are made to what TRAC outputs to XTV, the header-file-version identifier written by TRAC would become "XTV-TRAC/f77 V3.0.2". The revision number will be encoded so that it changes with each new version of TRAC, thus allowing the exact TRAC version to be recovered from this version number.

The compression flag will indicate whether the data block is compressed; the header section will never be compressed. How the data section is compressed will be handled later; this specification only adds the flag for that functionality. The date/time, machine, and platform-type information is designed to assist the user in identifying the run, as well as aid the XTV maintainer in diagnosing problems encountered.

The second line in the starting block will be the title for the run. This currently is drawn from the first title card in the TRACIN file.

The remainder of the starting block is devoted to enabling intelligent units conversion. TRAC has an extensive understanding of the type of any particular variable (e.g., `vol` is type volume, `vv` is type velocity) and has facilities for augmenting this list for the particular problem. TRAC will supply the XTV GUI with the user-defined units types as part of the XTV file so that the GUI has a full and complete list of all fundamental units types. In the starting block, the units system is specified, as well as how many problem-specific units types there are, followed by a line for each new unit type, which is derived from the input deck information supplied to TRAC.

Component Block

The component block consists of three sections: a generic component section, a graphics display section, and a variables section. The generic component section is the same for all components. It contains the component name (e.g., TEE), the component ID number, a component-type identifier with modifiers, and the label supplied through TRACIN. The component-type identifier and its modifiers determine the content of the graphics display section, which will be described in more detail below. The format of the variable section is the same for all components, though the types of variables must match the information supplied in the graphics display section (e.g., a 2D graphics display section cannot support a 3D variable).

The generic component section contains nearly all of the information that is common to all components. Key among these are the component-type identifier and the component name. The component-type identifier contains the dimensionality of the component (0, 1, 2, or 3D), as well as auxiliary-structure, parent-and-child, and dynamic-sizing modifiers. Each of these modifiers is discussed in more detail in the detailed header format section.

The current component types and modifiers are defined later in this appendix. The component name allows special differentiation between variables of the same type. A PIPE and a TEE will show up differently, even though they are both 1D components, because the TEE contains a side or branch leg in addition to the primary leg. This will appear regardless of whether they are both named PIPEs. Through the use of name, the

TEE could label the junctions, but not the PIPEs. This facility also allows for significant functionality of new components before modifications to XTV are completed (e.g., in adding a channel component).

The graphics display section contains the information needed to create the graphics template or visual tile in XTV. This information usually includes the total number of cells or nodes, the number of cells in a particular direction, their length and width, the junctions to a component, the number of branches, and descriptions of how to draw degenerate forms of itself (e.g., a 1D representation of a 2D object). The details of this structure vary significantly from none (0D) to many (3D). See the detailed description below for specific information.

The variable description section contains the total number of variables supplied and the number that is supplied only on initialization, followed by a one-line description of the variable and its contents. Currently, each variable is given a name and a Face flag that connotes its length. We propose substantially augmenting this information by adding a short reference name, a units label, a units type identifier, and an expanded variable-attributes section. For the alternate DMTRAC version, there also would be three index values added: the start of variable vector, file vector length, and number of timesteps in the vector. The file vector length will be directly computable from the variable attributes and the number of timesteps if compression is not used. The units label can be used by programs that are not units-type savvy, whereas the units-type identifier can be used by those who want to perform units conversion. The most significant of the changes is the variable-attributes section. Proposed attributes are dimension, frequency, length, color set, and degeneracy. Dimensions are 0, 1, 2, and 3D, as applicable to the component type. Frequency is proposed to be time-dependent or time-independent, but could expand to every other graphics dump, etc. Length contains two parameters: face/cell centered and static/dynamic for fine mesh rezoning in heat structures. Dynamically sized variables require that the variable that contains the current size be output before the dynamically sized variable is output. Color choice determines whether water colors (blue) or hot colors (red) are used to map the display tiles. Degeneracy is used to indicate along which axes the variable is defined. This degeneracy is used mainly for rods that contain data along the axial direction (z).

End of Header Block

The end of header block is composed solely of the string "End of Header Block", followed by a return. This marks the position of the beginning of the data block. Using a file pointer function such as `ftell()` stores the location of one end of the header and facilitates seeking the beginning of the data block for access via the `fseek()` function.

H.2.3. Summary of XTV Data Format

Header/Data Interface

The binary data follow immediately after the header file. To access the data section, open the graphics file in text mode, read in the header, and obtain the location of the end of the header using a function similar to `ftell()` when the "End of Header Block"

marker is found. Reopen the file in binary mode (if text and binary modes differ; they do not differ in Unix) and seek to the end of the header using a function similar to `fseek()`. What follows will be in the form of the timestep-edit format, which is written by TRAC.

Timestep Edit Format

The timestep edit format comprises a short binary header and subsequent component data blocks. What comprises the header will depend on whether compression is selected. The first value of the header is always a synchronization or check value (1.12345E+12), followed by the problem time. If compression is selected, two additional values are specified: the length of the compressed edit and the expanded length so that the proper data segment may be loaded and an appropriately sized buffer created.

Each component data block will be in the order presented in the index. Multidimensional arrays are output as they occur naturally in FORTRAN, i.e., the first index changes the most rapidly (Row-Major), followed by the second index, then the third, and so on.

To read in all values of a particular variable, typically you would read in each successive timestep edit, obtain the value desired, then discard the remaining data, then read in the next edit and repeat the procedure. This is most particularly true if the data are compressed or have dynamically dimensioned output, in which case there is little choice. If the data are not compressed and there are no dynamically dimensioned variables before the desired one, then the offset can be computed into the edit and the file can be stepped through with little processing. Currently, the heat structures output fixed-length, dynamically used (some space is undefined/unused) variables, so each timestep edit is exactly the same length. It may be advantageous in terms of space to switch to dynamically dimensioned variables, in which case the access time will be significantly increased.

To read in many values at the same timestep, e.g., for the void fraction profile in a PIPE, step through the data block to the relevant timestep and extract the values from the array `alpha(x)`. Typically, batch files and users access one variable at a time, so generally, all elements desired will be in the same array.

H.2.4. Detailed Header File Format

Starting Block

Line 1:	XTV-[TRAC | DMTRAC]/[f77 | f90] V<n>.<m>.<r>[-<nX>.<mX>] <compression> <machine> <platform> : <Date/Time>
Line 2:	<title>
Line 3:	<Units type> <# of supplied units types>
Lines 4-3+n:	<lulabel> <SI label> <alt units label> <multiplier> <offset>

TRAC used to denote file is in *timestep edit format*

DMTRAC	used to denote that file is in *variable vector format*
f77	used to denote that file was originally written by TRAC-M/F77
f90	used to denote that file was originally written by TRAC-M/F90
n	major XTV version
m	minor XTV version
r	revision number = current TRAC version # – TRAC version # when XTV was last changed for input handling
nX	major XTV version of executable that was used to convert to *variable vector format* (only for variable vector format headers)
mX	minor XTV version of executable that was used to convert to *variable vector format* (only for variable vector format headers)
compression	code for whether data section is compressed: CP data section compressed using `zlib` (deflate algorithm) UC data section is uncompressed
machine	result of `gethostname()` on POSIX compliant systems, e.g., `startrac`
platform	system name and OS release number "uname –sr"
Date/Time	date and time as returned by the operating system "date" or function ctime()
title	First title card in TRACIN input file.
Units type	SI or ENG or ALT

Component Block

*<component name> <component ID> <component type id> :<component label>

Graphics Display Block

Variable Definition Block

Graphics Display Block

0D

There is no block for an unmodified 0D (scalar) component

1D

Line 1: <nCells> <nJun> <nLegs>

nCells: Number of values along flow direction

nJun: Number of junctions on this component

nLegs: Number of side legs on this component. currently, plen = 0, tee = 1

Line set 2: <junID> <jCell > <jFace>... [for each `jun`, one `jun` per line]

junID: Identifier for this junction

jCell: Cell number where junction occurs

jFace: Code for which face junction attaches to. I = downstream (increasing cell numbers), i = upstream (decreasing cell numbers)

Note: only the first junction may attach to the "upstream" face; all others attach to the downstream face. (The one exception is PLENUM components, where the faces can be divided into either group.) Face noding always proceeds away from primary leg source and goes from the primary leg downside leg(s), even if the TEE legs are nominally sources.

Line set 3: <first cell> <last cell> <connects at> (for each leg, one leg per line)

Line set 4: <x Upper coordinate> (for each cell, eight per line)

Line set 5: <grav> (for each cell face, eight per line)

Line set 6: <fa> (for each cell face, eight per line)

2D

Line 1: <nCell-i> <nCell-j> <nJun> <nLegs> <CoordFlag>

CoordFlag: Code for coordinate system,

CART Cartesian (x,y)

CYLrt Cylindrical (r,θ)

CYLrz Cylindrical (r,z)

CYLtz Cylindrical (θ,z)

Line set 2: <junID> <jCell-i > <jCell-j> <jFace>... (for each `jun`, one `jun` per line)

junID: Identifier for this junction

jCell(i,j): (i,j) coordinates where junction occurs

jFace: Code for which face the junction attaches to. I/i = +/- face in first axis direction (see `CoordFlag`), J/j= +/- in second axis direction. C = cell center, `Jun` does not attach at the face

Line set 3: <first cell j> <last cell j> <connects at j> (for each side leg, one leg per line)

note: for now, all 2D legs break at j = value (e.g., z = const.)

Line set 4: <Radial segment outer radius/ x dimen.> (at each cell in r/x, eight per line)

Line set 5: <Axial segment upper elevations/y dimen.> (at each cell in z/y, eight per line)

Line set 6: <grav-i> ... (for each cell face in r/x, eight per line)

Line set 7: <grav-j> ... (for each cell face in z/y, eight per line)

Line set 8: <fa i> (for each cell face, eight per line)

Line set 9: <fa j> (for each cell face, eight per line)

3D

Line 1: <nCell-i> <nCell-j> <nCell-k> <nJun> <nLegs> <CoordFlag>

CoordFlag: Code for coordinate system:

CART Cartesian (x,y,z)

CYL Cylindrical(r,θ,z)

Line set 2: <junID> <jCell-i> <jCell-j> <jCell-k> <jFace>

junID: Identifier for this junction

jCell(i,j,k): (i,j,k) coordinates where junction occurs

jFace: Code for which face the junction attaches to. I/i = +/- face on x/r axis J/j = +/- face on y/t axis. K/k = +/- on z axis. C = cell center, Jun does not attach at the face.

Line set 3: <1st cell k> <last cell k> <connects at k> (for each side leg, one leg per line)

note: Currently, all 3D legs break at k = value (i.e., z = const.)

Line set 4: <Radial segment outer radius/x dimension> (at each cell in r/x, eight per line)

Line set 5: <Theta segment angle/y dimension>(at each cell in t/y, eight per line)

Line set 6: <Axial segment upper elevations/z dimen.>(at each cell in z, eight per line)

Line set 7: <grav-i> ... (for each cell face in r/x, eight per line)

Line set 8: <grav-j> ...(for each cell face in t/y, eight per line)

Line set 9: <grav-k> ... (for each cell face in z, eight per line)

Line set 10: <fa-x/r> flow area x/r face (for each cell face, eight per line)

Line set 11: <fa-y/t> (for each cell face, eight per line)

Line set 12: <fa-z> (for each cell face, eight per line)

Modifiers

A: Auxillary Structure. Denotes that there is a special-purpose auxiliary structure that follows the regular Graphics display block. The first line of that structure will contain two or three items: the name of the auxiliary structure, the number of lines it occupies, and, optionally, a version number.

There is only one auxiliary information structure currently defined, PlenAux, which contains the lengths of each plenum junction.

P: Parent (e.g., 0DP – HTSTR Parent)
Identifies component as parent with substructures that may be 0, 1, 2 or 3D.

Line 1a: <nchildren>

C: Child (e.g., 2DC—HTSTR Child)
Identifies structure as a child of a parent component. No new variables are associated with this, but the `compID` value is special. `compID` is defined for children as parent# - child#. If a heat structure with two rods, one average and one hot, were component 902, then the first (average) rod would be 902-1 and the second (hot) rod would be 902-2.

S: Dynamically Sized Axis(es).
Used on components such as TRAC HTSTRs, where one or more of the axes are typically dynamically sized in one manner between a minimum and maximum number of elements. This allows for any of the variables to be declared as "sized by the component" (DC) rather than individually defined for each variable. This also obviates the need for dimensional information on the dynamically sized axis if the minimum is 1. (Line sets 3 or 4 in 2D; line sets 3, 4, 5 in 3D as appropriate.) The dimensions on line 1 of the structures must still be entered; they are the maximum number of datapoints in that dimension.

Line 1a: <# of dynamically Sized Axes> <axis1> <VarType> <Var Name> <Minimum>, <axis2>...

VarType: Variable Dynamic Sizing Attribute as defined in Variable Attributes Section (this variable cannot be type DC) for the variable that determines the size of the other variables.

VarName: name of the variable that controls the size of the dynamically sized variables. Note that this ideally should come before the dynamically dimensioned variables and must come before any variables where the output length varies. (Some variables are output at a fixed length, but only a fraction of them contain real data.)

Minimum: Minimum number of elements present. If this value is

greater than 1, then the axis dimensional information should be given for the minimum number of elements.

If both P and S are present, the entries appear alphabetically, i.e.:

Line 1a: <nChildren> (P)
Line 1b: <# of dyn Sized Axes> <axis1> <VarType> <Var Name> <Minimum>, <axis2>...

Auxiliary Component Structures

Auxiliary component structures allow for the customization of information within a generic type. Because components generally are classed by their primary dimension, adding special information, such as signal variable type, can be a problem. The solution is to use the Auxiliary Component Structure. This is a self-typing (not self-describing) structure that can be bypassed if the reader has not been programmed to understand the component type.

Line 1: <Aux Struct Name> <lines occupied by the structure> [<vers string>]
Lines 2-nn: Determined by individual structure

PlenAux

Lines 2-(njun/8+1): dx jun(i) for all junctions (8/line)

Variable Definition Block

Line 1: <number of total variables> <number of static (time-independent) variables>

Timestep Edit Format

Line n: <VarName> <VarType>: <variable attributes>: <units label>: <VarString>

Variable Vector Format

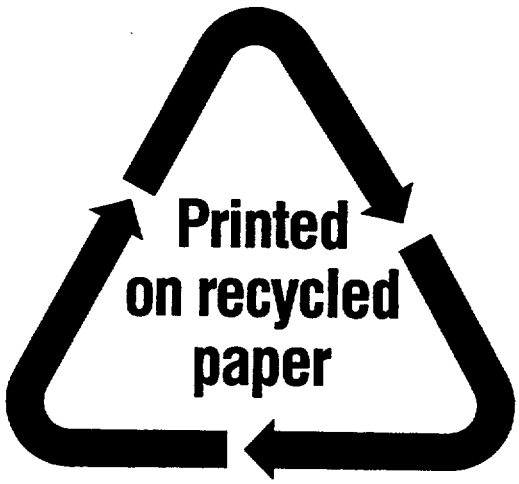
Line n: <VarName> <StartIndex> <FileLength> <Nsteps> <VarType>: <variable attributes>: <units label>: <VarString>

VarName: Short name of variable for index and quick reference (e.g., `rhol`)
VarType: Unit type identifier from TRAC (e.g., `luden`)
variable attributes: Space-delineated list of dimension, frequency, length, and special-option attributes. See table below for attribute, codes, and meanings.
units label: Label for use in graphs and printouts (e.g., kg/m^3)
VarString: Descriptive label for the variable (e.g., liquid density)

Attribute	Value	Description
Dimension	This gives the dimension of indexing used. Subdimensional variable must use the correct value here and specify the degeneracy attribute for proper array-length specification	
	0D	scalar value (not an array)
	1D	1D (linear) array
	2D	2D array [indexed as (i,j)]
	3D	3D array [indexed as (i,j,k)]
Frequency	This provides the frequency of output in the graphics file. Later, options for reduced resolution may be specified (e.g., every other edit, every third edit, etc.)	
	TI	Time-independent value (output on first edit only)
	TD*	Time-dependent value (output every edit)
Data Position	This provides the position of data within the cell	
	CC	Cell-centered value array (same number of values as cells)
	FA<ax>	Face-valued array (one more value than cells on face axis for the array); 1D arrays do not specify axis; 2D and 3D axes specify i,j,k for first, second, or third axis (x/r,y/t,z), e.g., FAi
Length	Provides the length in terms of known dimensions of the array	
	FX	Fixed-dimension array. Array length given by cell dimensions +1 if it is a face-valued array and not an azimuthal variable
	DC	Dynamically dimensioned at the component level. See the S option for component types
	DM	Fixed-dimension array that has dimension of minimum value for component-defined, dynamically scaled axis
	DD<ax> <var1>... <var2>...	Dynamically dimensioned array. ax specifies the axes that are dynamic (i,j,k). Array length given by variable `var1` specified earlier this edit. The length of each cell is defined by `var2`. One variable per dimension must be supplied
	DL<ax> <var1>... <var2>...	Dynamic/fixed-length variable. ax specifies the axes that are dynamic (i,j,k). Dimension specifiers give length written; actual data are of length `var1`, which is of type 0D. The length of each cell is defined by `var2`. One variable per dimension must be supplied

	DN<ax> <var>...	Dynamic/fixed-length variable array. ax specifies the axes that are dynamic (i,j,k). Dimension specifiers give length written, first negative value gives end of actual data. The length of each cell is defined by `var`. One variable per dimension must be supplied
	DR <var>	Dynamic/fixed-length variable. Dimension specifiers give length written; actual data are the same length as `var` for all dimensions. The length of each cell is equivalent ot the length of the equivalent cell in `var`
Degeneracy	This specifies along which axis (es) the degenerate array lies. For example, all degenerate rod variables lie along the k (z) axis. This is used for selecting the dimension	
	i,j,k	Standard single-axis identifiers
	ij,ik,jk	2D degenerate identifiers
Color Mapping	Specifies what color set to use for the visualization	
	WC*	Use watercolors (blues)
	HC	Use hot colors (reds)
Special options	These codes are special-purpose options. See the individual explanations	
	VI <name>*	i component of vector-valued function (i,j,k must be sequential). Note that the name immediately follows this identifier and is terminated by an asterisk(*). This name appears in the vectors submenu of XTV. The rank of the vector is equal to the rank of the component. It exists at as many dimensions as are defined by the length attribute of the component (thus, 2D components will not have VK defined). E.g., `vgxr luvelocity` :3D FAi VI Vapor Vectors* FX:m/s: radial gas velocity. This is the first of three variables (a 3D variable must belong to a 3D component) that form the vector named "Vapor Vectors". It has (ni+1) *nj *nk elements ...
	VJ	j component of vector-valued function (i,j,k must be sequential)
	VK	k component of vector-valued function (i,j,k must be sequential)

	IT <name>*	i row of tensor function (i,j,k must be sequential). Note that the name immediately follows this identifier and is terminated by an asterisk(*). This name appears in the vectors submenu of XTV. The rank of the tensor is given by the component type. Thus, a scalar tensor of a 3D component has three values: e.g., `tensor1i lunounit :0D IT Tensor1*:-:Thermal Coupling`, and a 1D tensor (a tensor at each value along an axis) might be `tensor2i lunounit: 1D i IT Tensor2* FA:-:Reactivity Fdbk`. This would be an array (ni+1) * 2 elements long if it belonged to a 2D component.
	JT	j row of tensor function (i,j,k must be sequential)
	KT	k row of tensor function (i,j,k must be sequential)
	ID	(inset display) Display as an inset value (as wall temperatures are now)
	UV	(unlisted value) Do not place on variable selection list. (Typically used for variable use for dimensioning dynamic arrays)



Federal Recycling Program

NRC FORM 335 (2-89) NRCM 1102, 3201, 3202

U.S. NUCLEAR REGULATORY COMMISSION

BIBLIOGRAPHIC DATA SHEET

(See instructions on the reverse)

1. REPORT NUMBER (Assigned by NRC, Add Vol., Supp., Rev., and Addendum Numbers, if any.)

NUREG/CR-6725

2. TITLE AND SUBTITLE

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

3. DATE REPORT PUBLISHED

MONTH	YEAR
May	2001

4. FIN OR GRANT NUMBER

W6245

5. AUTHOR(S)

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

J.H. Mahaffy, C. Murray, PSU

6. TYPE OF REPORT

Technical

7. PERIOD COVERED *(Inclusive Dates)*

8. PERFORMING ORGANIZATION - NAME AND ADDRESS *(If NRC, provide Division, Office or Region, U.S. Nuclear Regulatory Commission, and mailing address; if contractor, provide name and mailing address.)*

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Pennsylvania State University
University Park, PA 16802

9. SPONSORING ORGANIZATION - NAME AND ADDRESS *(If NRC, type "Same as above"; if contractor, provide NRC Division, Office or Region, U.S. Nuclear Regulatory Commission, and mailing address.)*

Division of Systems Analysis and Regulatory Effectiveness
Office of Nuclear Regulatory Research
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001

10. SUPPLEMENTARY NOTES

F. Odar, NRC Project Manager

11. ABSTRACT *(200 words or less)*

The Transient Reactor Analysis Code (TRAC) was developed to provide advanced best-estimate predictions of postulated accidents in light-water reactors. The TRAC-P program has provided this capability for pressurized water reactors and for many thermal-hydraulic test facilities for approximately 20 years. However, the maintenance and portability of TRAC-P had become cumbersome because of the historical nature of the code and the inconsistent use of standardized FORTRAN. Thus, the Modernized TRAC (TRAC-M) was developed by recoding the TRAC-P algorithms to take advantage of the advanced features available in the FORTRAN 90 programming language while conserving the computational models available in the original code. The Programmer's Manual is one of four-volume set of documents on TRAC-M, and was developed to assist a programmer and contains information on the TRAC-M code and data structure, the TRAC-M calculation sequence, memory management, and data precision. This document provides a code developer with a single source of information to allow either modification of or addition to the code. Sufficient information is provided to permit replacement of modification of physical models and correlations.

12. KEY WORDS/DESCRIPTORS *(List words or phrases that will assist researchers in locating the report.)*

TRAC
TRAC-M
FORTRAN 90
thermal hydraulics
pressurized water reactor

13. AVAILABILITY STATEMENT

unlimited

14. SECURITY CLASSIFICATION

(This Page)
unclassified

(This Report)
unclassified

15. NUMBER OF PAGES

16. PRICE

UNITED STATES
NUCLEAR REGULATORY COMMISSION
WASHINGTON, DC 20555-0001

OFFICIAL BUSINESS
PENALTY FOR PRIVATE USE, $300