TRNSYS 16

a TRaNsient SYstem Simulation program

Volume 8

Programmer's Guide

Solar Energy Laboratory, Univ. of Wisconsin-Madison
http://sel.me.wisc.edu/trnsys

TRANSSOLAR Energietechnik GmbH
http://www.transsolar.com

CSTB – Centre Scientifique et Technique du Bâtiment
http://software.cstb.fr

TESS – Thermal Energy System Specialists, LLC
http://www.tess-inc.com

About This Manual

The information presented in this manual is intended to provide a reference to users who want to create their own TRNSYS 16 components or want to understand the TRNSYS 16 simulation engine. This manual is not intended to provide information about other utility programs in TRNSYS 16. More information concerning these programs can be found in other parts of the TRNSYS documentation set. The latest version of this manual is always available for registered users on the TRNSYS website (see here below).

Revision history

- 2004-09 For TRNSYS 16.00.0000
- 2005-02 For TRNSYS 16.00.0037
- 2006-01 For TRNSYS 16.01.0000
- 2007-02 For TRNSYS 16.01.0003

Where to find more information

Further information about the program and its availability can be obtained from the TRNSYS website or from the TRNSYS coordinator at the Solar Energy Lab:

TRNSYS Coordinator Solar Energy Laboratory, University of Wisconsin-Madison 1500 Engineering Drive, 1303 Engineering Research Building Madison, WI 53706 – U.S.A.	Email: trnsys@engr.wisc.edu Phone: +1 (608) 263 1586 Fax: +1 (608) 262 8464
TRNSYS website: http://sel.me.wisc.edu/trnsys	

Notice

TRNSYS Contributors

S.A. Klein	W.A. Beckman	J.W. Mitchell
J.A. Duffie	N.A. Duffie	T.L. Freeman
J.C. Mitchell	J.E. Braun	B.L. Evans
J.P. Kummer	R.E. Urban	A. Fiksel
J.W. Thornton	N.J. Blair	P.M. Williams
D.E. Bradley	T.P. McDowell	M. Kummert
D.A. Arias		

Additional contributors who developed components that have been included in the Standard Library are listed in Volume 5.

Contributors to the building model (Type 56) and its interface (TRNBuild) are listed in Volume 6.

Contributors to the TRNSYS Simulation Studio are listed in Volume 2.

Table of contents

8. PROGRAMMER'S GUIDE

8.1. Introduction

There are 3 main sections in this guide:

- Section 8.2 explains how to update TRNSYS 15 components in order to use them in TRNSYS 16. All TRNSYS 15 Types can be used in legacy mode after a few steps that do not require any programming knowledge.
- Section 8.3 describes the process of creating a new component from scratch
- Section 8.4 Provides detailed reference on the TRNSYS kernel:
  - Calling sequence of Types and structure of the INFO array (8.4.3)
  - Available global constants (8.4.1). Some constants are especially interesting for programmers since they set standard string lengths in TRNSYS (Fortran 90 does not handle variable-length strings), see section 8.4.1.2.
  - Access functions that allow Types to access kernel variables (8.4.2)
  - Utility subroutines (0)
  - Instructions to recompile the TRNSYS DLL and add new DLL's to UserLib using different compilers (Sections 8.5, 8.5.2 and above)

8.2. How to update TRNSYS 15 components

Existing TRNSYS 15 components can be updated to TRNSYS 16 in two ways:

- **After 2 easy steps that do not require any programming knowledge, you can run existing Types in a "legacy TRNSYS 15" mode**, where all the information exchange with the Kernel is identical to what was happening in TRNSYS 15. This implies that you will not benefit from some of the new features:
  - General use of double precision
  - Added security and user-friendliness thanks to access functions, etc.
  - Drop-in DLL's: **Types running in "legacy TRNSYS 15 mode" must be linked to the main TRNSYS DLL (TRNDll.dll).**

  You will also have to be careful if you are interested in the system response just after initialization: Because of the change in the definition of the initial time step, TRNSYS 15 Types might give an incorrect answer just after the simulation start.

  Simple components can be put into external DLLs in legacy mode. However, if the Type needs to call one of the TRNSYS utility routines, it needs to be fully updated so that it can call the double precision versions of those routines instead of the outdated single precision versions.
- **You can also fully convert an existing component to the TRNSYS 16 standard by following a systematic procedure**. The difficulty of that procedure and the time required to perform the conversion will depend on the way the existing Type is coded: heavy access to Kernel variables through common blocks, use of Fortran functions specific to single precision variables, etc.

8.2.1. Steps to use TRNSYS 15 components in "legacy mode"

The following 2 steps are **required** in order to use TRNSYS 15 Types in TRNSYS 16. If you want to use your Type in a "TRNSYS 15 legacy mode", this is all you need to do. If you want to fully convert your Types to the TRNSYS 16 standard, you will find additional instructions in section 8.2.1.3. Note that the steps listed in this section are also required to fully convert existing TRNSYS 15 Types. **NOTE: If you choose to use your component in "legacy mode" the Type *cannot* be placed in an external DLL; it *must* be compiled and linked into TRNDll.dll along with TRNSYS's standard components.**

8.2.1.1. Export the name of the routine (Typennn)

In order for your component to be found by the TRNSYS Kernel, it needs to broadcast its name. This is done by adding the following syntax to the component just underneath the subroutine line at the top of the file.

```
!DEC$ATTRIBUTES DLLEXPORT :: TYPEnnn
```

Where **nnn is the Type number**. Thus Type205 would have the following line added:

```
!DEC$ATTRIBUTES DLLEXPORT :: TYPE205
```

It is important that the syntax begin in the leftmost column. In other words, do not put any tabs or spaces to the left of the first exclamation point. As far as the Fortran compiler itself is concerned, the line that has just been added is a comment. However, the pre-processor recognizes the syntax and includes the Type in a file called an export library. Essentially this means that the subroutine will be accessible by other subroutines not contained within the same DLL.

Note: The line here above is required due to the new calling mechanism in TRNSYS 16, even though TRNSYS 15 Types have to be linked to the main DLL. The "DLLEXPORT" directive is required but it does not mean your Type will be usable in an external DLL.

8.2.1.2. Version sign the Type

A number of structural changes have been made to TRNSYS components. Types in TRNSYS 16 are called in a different manner and with new INFO array codes, allowing them much more flexibility in when and how they perform their calculations. Many of these changes are not compatible with Types written for TRNSYS 15 or earlier so to avoid forcing users to completely rewrite all their components, TRNSYS 16 contains two parallel calling structures, one for new TRNSYS 16 components, and one for legacy components. In order for TRNSYS to determine the manner in which it should call a component, each component must be "signed" with a version number for which it was written. To sign your component, you need to add the following syntax to it, at the top of the executable section (just below the variable declarations)

```
! Set the version information for TRNSYS
      if (INFO(7) == -2) then
           INFO(12) = 15
           return 1
      endif
```

By adding the above, you have signaled to TRNSYS that your component was written using the TRNSYS 15 calling structure. TRNSYS will treat it accordingly. There should be no further changes that you need to make to your Type in order for it to run under version 16.

8.2.1.3. Additional considerations

If your Type refers to "param.inc", the include file that declares global TRNSYS 15 constants, you need to update the path to that file. Param.inc is now located in %TRNSYS16%\SourceCode\Include (%TRNSYS16% is your installation directory). Note that Param.inc is obsolete and that TRNSYS 16 Types should use the global constants declared in the TrnsysConstants module (see section 8.2.2.4 for details).

8.2.2. Instructions to convert TRNSYS 15 Types to the TRNSYS 16 standard

The steps required in order to convert an existing Type to the TRNSYS 16 standard are given in the following sections. The first two steps are the same as the steps required to run TRNSYS 15 Types in "legacy mode" (except that the Type is signed as a "Version 16" Type). Please note that if you have a Type that was written for a TRNSYS version prior to TRNSYS 15, steps for converting those Types are listed in the Reference section of this manual.

8.2.2.1. Export the name of the routine (Typennn)

See section 8.2.1.1 here above for details. You need to add the following syntax to the component just underneath the subroutine line at the top of the file.

```
!DEC$ATTRIBUTES DLLEXPORT :: TYPEnnn
```

Where **nnn is the Type number**. Thus Type205 would have the following line added:

```
!DEC$ATTRIBUTES DLLEXPORT :: TYPE205
```

It is important that the syntax begin in the leftmost column. In other words, do not put any tabs or spaces to the left of the first exclamation point.

8.2.2.2. Special calls identified by INFO(7:8) and INFO (13)

Note that detailed explanations of the INFO array and a full example of a typical calling sequence for an iterative call are provided in section 8.4.3. The following sections just outline the changes to existing TRNSYS 15 Types.

VERSION SIGN CALL (INFO(7) = -2)

See section 8.2.1.2, here above, for details. To sign your component, you need to add the following syntax to it, at the top of the executable section (just below the variable declarations):

```
! Set the version information for TRNSYS
      if (INFO(7) == -2) then
           INFO(12) = 16
           return 1
      endif
```

By adding the above, you have signaled to TRNSYS that your component was written using the TRNSYS 16 standard.

INITIALIZATION (PRE-SIMULATION) CALL (INFO(7) = -1)

The INFO(7) = -1 call is still intended to allow components to perform initializations, as in TRNSYS 15. However, due to the new specification of TIME in TRNSYS, the manipulations required at INFO(7) = -1 have changed. The manipulations that your component should do at INFO(7) = -1 are the following:

- Set INFO(6) to the number of output spots needed by the component.
- Set INFO(9) so that your Type will be called in the proper manner (see section 8.4.3 for details).
- Handle data storage initialization:
  - Set INFO(10) to the number of storage spots required in the single precision storage structure (in other words, uses the S array). (***Deprecated***)
  - Call `SetStorageSize` to reserve space in the double precision storage structure. See section 8.2.2.6 for more information.
- Call TYPECK to have TRNSYS check whether the correct number of Inputs, Outputs and Parameters were specified in the deck.
- Call RCHECK to have TRNSYS check the units of the Input Output connections in the deck.
- `return 1`: no other manipulations should be made during this call.

Note that reading parameters and process them can be realized either during this call or during the initial time step (see "TIME = TIME0 call" here below). ***You should be careful to handle all operations that must be done only once (e.g. file opening) in only one of those initial calls***.

NEW SIGNIFICANCE OF THE FIRST TIME STEP (TIME = TIME0)

In TRNSYS 16, the time specified in the simulation cards ("Simulation Start Time" in the Simulation Studio) is the exact moment at which the simulation starts. This is different from TRNSYS 15, where the so-called simulation start was the time at the end of the first time step. In TRNSYS 16, components are called at TIME = TIME0 and they should just output their initial values at that call. ***There are no iterations for that time step***.

The manipulations that a component should make at TIME = TIME0 are:

- read parameter values and set them to local variable names
- check parameters for validity. For example, you might want to check to see if the specified specific heat is negative and generate an error if it is. Note that
- set outputs to initial conditions or 0
- `return 1`: no other manipulations should be made during this call.

"POST CONVERGENCE" CALL (INFO(13) = 1)

All components that are signed as having been written for version 16 are called once more at the end of every time step after convergence has been reached. This modification was made in order to simplify the use of storage (the TRNSYS 15 S array) and other operations that must be done after convergence has been reached, such as printing or integration. If your component does not use storage and is not a printer or an integrator, then you may simply add the lines:

```
!    Perform post-convergence operations
      if(INFO(13) > 0) then
        return 1
      endif
```

If your component does use storage, then you should add lines that set up your local storage array, then send the values to global storage using a call to the setStorageVars subroutine as explained in section 8.2.2.6.

FINAL CALL AFTER THE SIMULATION IS DONE (INFO(8) = -1)

This call has not changed between TRNSYS 15 and 16. At the very end of a TRNSYS simulation, each Type is called one last time with INFO(8) = -1. If your Type does not contain any handling of INFO(8) = -1, it will simply run through its calculations one last time and probably no harm will come of it. However, it is a good idea to specifically handle INFO(8). You can add the following lines if your component does not have to do anything:

```
!    Perform last call manipulations
      if(INFO(8) == -1) then
        return 1
      endif
```

You may want your components to actually do something at the end of the simulation. They could, for instances print a message to the list file or close logical units that were used during the simulation. Standard component Type 22 (Iterative Feedback controller) performs end of the simulation manipulations that you might use as an example.

The INFO(8)=-1 call also occurs if the simulation terminates with a fatal error. In that case, the component that generates the error returns control to TRNSYS, which calls all components one last time in order to perform their "end of simulation" operations. You can check if the very last call occurs because of an error or as part of the normal simulation process by calling getNumberOfErrors(). Some end of simulation operations are unnecessary or might crash TRNSYS if the simulation ends with a fatal error. E.g. Type 22 handles INFO(8)=1 as follows:

```
if (info(8) == -1) then
    ! Exit immediately if this call is the result of a fatal error
    if (getNumberOfErrors() > 0) then
        return 1
    ! Otherwise, print the nb of "stuck" timesteps
    else
        ...
        printing manipulations
        ...
    endif
    return 1    ! Exit
endif
```

Notes:

- If the simulation ends without errors, the INFO(8) = -1 call happens ***after the user has allowed the simulation to terminate*** by clicking on the "yes" or "continue" button at the end of the simulation
- The lines of code here above should be placed ***before the normal instructions*** so the return occurs before those instructions are executed.

8.2.2.3. Double precision variables

One of the other focuses of version 16 development was to move away from single precision variables toward double precision variables. Four of the arguments sent to each Type are now double precision (TIME, PAR, T, and DTDT). Previously only XIN and OUT were double precision. You will need to go through and declare TIME, PAR, T and DTDT as double precision instead of real. It is also recommended that you do this for all your variables. If you prefer to keep your variables single precision, it is important that you help Fortran make the conversion correctly. The following incorrect code will not be caught by the Fortran compiler and may result in incorrect answers:

```
RealVariable = DoublePrecisionVariable
```

The correct way of performing the above is the following:

```
RealVariable = sngl(DoublePrecisionVariable)
```

To go the other way, you need the following:

```
DoublePrecisionVariable = dble(RealVariable)
```

8.2.2.4. Access to global constants: the "TrnsysConstants" module

TRNSYS 16 Types should access global constants through the "TrnsysConstants" data module. The available constants are described in section 8.4.1. Many new constants have been added in order to promote consistency in TRNSYS (e.g. by making sure strings are sized in a consistent way), and the existing constants previously found in the "param.inc" file have been transferred into the new data module. Note that "param.inc" is still included in the TRNSYS distribution for backwards compatibility, but it should not be used by TRNSYS 16 Types. To use the "TrnsysConstants" module in a Type, add the following line at the top of the declaration section (i.e. just below the !DEC$ATTRIBUTES instruction):

```
use TrnsysConstants
```

This will declare all the constants in the module in your Type. Alternatively, if you only want to use the constant defining the maximum length of a Label string (maxLabelLength), you can use the following syntax:

```
use TrnsysConstants, only: maxLabelLength
```

This will only declare the constant(s) listed after the "only" keyword. You can list several constants in one "only:" instruction by using a comma-separated list.

A NOTE ON STRINGS

Fortran 77/90/95 do not provide variable-length strings, so each individual string variable must be sized according to its expected maximum length. This can cause problems when different parts of the code expect different maximum lengths for the same string. If your Type manipulates strings, we strongly recommend that you check section 8.4.1 to see if a suitable constant is already available for the size of your string, rather than using a new constant or a hard-coded value.

It is also recommended to use the generic "a" format when printing strings, rather than "ann" where nn is the length of the string. Where an old print instructions for a variable descriptor (column title in a printer) might say:

```
      character*10 myString
      myString = 'Hello world'
      write(*,1000) myString
1000  format(a20)
```

The recommended way is now:

```
use TrnsysConstants    ! defines integer, parameter ::maxDescripLength
character (len=maxDescripLength) :: myString
myString = 'Hello world'
write(*,'(a)') trim(myString)
```

Note that the generic "a" format does not work for character arrays (which are not exactly the same as strings). In that case it is necessary to adjust the format instruction with a variable, e.g.:

```
use TrnsysConstants    ! defines integer, parameter ::maxDescripLength
character (len=100) :: myFormatString
character :: myString(maxDescripLength)    ! myString is a character array,
                                             not a string
write(myFormatString,'(i)') maxDescripLength
myFormatString = '(' // trim(adjustl(myFormatString)) // 'a1)'
myString = 'Hello world'
write(*,myFormatString) trim(myString)
```

We recommend that you use string variables rather than character arrays when possible.

8.2.2.5. Access to Kernel variables: Access Functions and the "TrnsysFunctions" module

WHY ACCESS FUNCTIONS?

TRNSYS 16 Types should access global (Kernel) variables through access functions only. Access functions allow Types to use Kernel variables in a much safer way than the Common Blocks that were used in TRNSYS 15. Common Blocks are inherently dangerous because they did not ensure name nor even type consistency for Kernel variables among the Types. There is also no way of preventing a Type from changing the simulation stop time, or even the current simulation time, with Common Blocks.

The principle of Access Functions is to provide functions that allow access to Kernel variables in the way they are intended to be used. Foe example, a function is provided to "Get" the simulation time step, but no function is provided to change it.

All access functions are listed in section 8.4.2.

HOW TO USE ACCESS FUNCTIONS

First, the module with all the function declarations must be Used in the Type:

```
use TrnsysFunctions
```

That line should be inserted with the other "Use" instructions at the top of the declaration section. Here again, it is possible to add "only:" instructions if only some of the functions are needed and if name conflicts may occur (it is **strongly** discouraged to re-use TRNSYS function names in Types)

Then, when a Kernel variable is needed, it can be accessed with the corresponding function. For example, the simulation time step (DELT variable in the Kernel) is accessed through the "getSimulationTimeStep()" function. Let's say a Type is integrating a value x over time, the result being put in "integral".

TRNSYS 15

```
subroutine Type205(TIME,XIN,OUT,T,DTDT,PAR,INFO,ICNTRL,*)
common/SIM/TIME0,TFINAL,DELT,IWARN
real TIME0,TFINAL,DELT
integer IWARN
…
real x,s
…
s = s+x*DELT
```

TRNSYS 16

```
subroutine Type205(TIME,XIN,OUT,T,DTDT,PAR,INFO,ICNTRL,*)
use TrnsysFunctions

double precision :: delt, x, s
…
delt = getSimulationTimeStep()
…
s = s+x*delt
```

NOTE ON EXTERNAL DLL'S

The TRNSYS 15 Common Blocks are still present in TRNSYS 16 in order to allow existing Types to run in "legacy mode". However, they are **not** exported from the main DLL (TRNDll.dll). If you do not replace Common Blocks with the use of Access Functions, you will not be able to use your Type in an external DLL.

8.2.2.6. Data storage (the "S array")

STORAGE IN TRNSYS 15

One of the most unintuitive utilities in TRNSYS 15 was storage using the S array. The underlying idea of storage is that sometimes you want your calculations not to be based upon the value of a variable at the last iteration but the value of that variable at the end of the last time step. To use the S array, one would bring it in as part of a COMMON block and declare how many places were needed by setting INFO(10) to that number. Then every time a stored variable was needed, the user would get a pointer into the S array by setting INFO(10) to a local integer variable (such as ISTORE), then would start pulling out the stored values by referencing them as S(ISTORE), S(ISTORE+1) etc. At the end of a time step, the user would do the reverse, again getting a pointer into the S array, then setting its values to the correct local variables.

The problem was that Types did not know when it was the end of a time step, only that a new time step had begun. Consequently, users had to notice that a new time step had started, then update their S array with the most recent values (from the end of the last time step).In addition, there was no check that a given Unit was not writing values in parts of the S array that had been allocated to a different Unit.

STORAGE IN TRNSYS 16

In TRNSYS 16, the S array still exists in the COMMON block /STORE/ for legacy components. However, three new utility subroutines were created, allowing for a hopefully more intuitive approach to variable storage. The functions are described in details in section 8.4.2.1.

- At the initialization call (INFO(7) = -1), the Type should call `SetStorageSize` to allocate storage for the current Unit.
- When the user wants to set storage at the end of a time step, a call is made to the `setStorageVars` subroutine when INFO(13) = 1
- When the user wants to obtain a previously stored value, a call is made to the `getStorageVars` subroutine. This usually occurs during the first call of a new time step, identified by INFO(7) = 0.

8.2.2.7. The "Messages" subroutine

One of the enhancements in TRNSYS 16 is the introduction of a Messages subroutine that handles all notices and error messages that component print to the listing and log files. This makes it easier to parse a listing or log file for errors and it ensures a consistent handling of messages by the Simulation Studio's "Error Manager".

Where Type previously wrote error messages to the listing file (accessed throught the logical unit LUW in the LUNITS Common Block), they should now call the Messages subroutine:

```
call Messages(errorCode,message,severity,unitNo,typeNo)
```

errorCode is a standard TRNSYS error number (if available), message is the string that has to be printed, severity indicates the severity of the message (e.g. warning or error) and UnitNo and TypeNo indicate the calling unit and Type numbers.

Please refer to section 8.4.4.9 for more information on the Messages subroutine.

8.2.2.8. Utility subroutine calls

One of the focuses of version 16 development was to move away from single precision variables toward double precision variables. As a consequence, it is important to correctly handle the arguments to utility subroutines (such as PSYCH) that your components may be calling. In TRNSYS 15, the call to PSYCH was:

```
call PSYCH(TIME,INFO,IUNITS,MODE,WBMODE,PSYDAT,0,STATUS,N)
```

The arguments TIME and PSYDAT were both declared as real (single precision). In TRNSYS 16, both TIME and PSYDAT should be declared in components as double precision and the call should be made to the double precision version of PSYCH, which is called Psychrometrics. Thus the syntax would be:

```
call Psychrometrics(TIME,INFO,IUNITS,MODE,WBMODE,PSYDAT,0,STATUS,N)
```

The single precision PSYCH subroutine still exists in TRNSYS 16 but it is merely a shell that correctly converts single precision arguments to double precision, calls Psychrometrics, and reconverts the results from double precision back to single precision. Table 8.2.2.8-1 shows the TRNSYS 15 utility subroutine names in column 1, the corresponding TRNSYS 16 utility subroutine name in column 2 and the arguments that must be converted from single to double precision in column 3.

Table 8.2.2.8-1: TRNSYS 15 (single precision) and TRNSYS 16 (double precision) utility routines

Single precision routine	**Double precision routine**	**Arguments that need to be changed from single to double precision**
PSYCH	Psychrometrics	TIME, PSYDAT
DATA	DynamicData	X, Y
TALF	TauAlpha	THETA,XKL, REFRIN, ALPHA, RHOD
DIFFEQ	Differential_Eqn	TIME, AA, BB, TI, TF, TBAR
ENCL	Enclosure	SPAR, WPAR, FV
VIEW	ViewFactor	A, B, C, D, E, F, G, H, X3, Y3, Z3
TABLE	Table_Coefs	B, C, D
INVERT	Matrix_Invert	A

DINVRT	Matrix_Invert	A
FLUIDS	Fluid_Props	PROP
STEAM	Steam_Props	PROP

Additional information on Utility Subroutines can be found in section 0.

8.3. How to create new components

One of TRNSYS's major strengths is the ease with which users may write new components to expand upon the capabilities of the program. Three features provide the foundation of this expandability. First is TRNSYS's open architecture. With one exception, all standard components are provided along with their source code to act as a reference and to act as the basis for adding new components. Second, all components whether standard components or user written components are formulated in the same manner and follow the same steps and progression throughout the code; writing new components becomes a matter of using a template and adding the appropriate functions, utility calls and equations to write a model of your own. Thirdly, and perhaps most importantly, the TRNSYS kernel does not impose any hierarchy whatsoever on the components that are used in a simulation. Nor does it make any assumptions regarding the order in which components should be solved to simulate the system. Many other simulation packages do impose such a hierarchy. To take the example of simulating a building, they often solve the building loads using one part of their kernel, then proceed to solve the system with another part of their kernel and finally solve the plant aspect of the system last. Many do not revise a previous step based on calculations in a subsequent step. Thus the system could well not meet the load at a given time step. However, the building, at the next time step, has no idea that the load was not met. TRNSYS, by contrast, continues to iterate through all components (whether by successive substitution or Powell's method) until convergence is reached. While this can make controls in a TRNSYS simulation somewhat more cumbersome, it has a major advantage from the point of view of someone writing a new component; that is that there is no need for the component writer to modify the TRNSYS kernel in any way in order to accommodate and include their new component. There is no need, for example to decide where in the management hierarchy the new component that you are writing fits best. Components in TRNSYS are automatically called by the kernel as soon as they are found in a given simulation. By assigning a number to each component, the TRNSYS kernel anticipates and automatically calls all component numbers that it finds in a simulation input file. If a particular Type number is found in the input file but is not found in the compiled code by the TRNSYS kernel, a dummy place holding component is called in its stead and the simulation ends with an error.

The following sections of this manual will step the new TRNSYS Type programmer through the process of filling in a Type template in order to create a new model for use in TRNSYS simulations.

8.3.1. Fundamentals

At the most fundamental level, a component (referred to as a Type) in TRNSYS is merely a black box. The TRNSYS kernel feeds inputs to the black box and in turn, the black box produces outputs. To delve a little deeper, however, TRNSYS makes a distinction between inputs that change with time and inputs that do not change with time. Examples of inputs that might change with time are temperature, flow rate, or voltage. Example of inputs that do not change with time are area or a rated capacity. In the early days of TRNSYS, this distinction was critical because computing time was very costly and inputs that do not change with time can be set once at the beginning of a simulation thereby saving a good bit of calculation time. With modern computing power, this distinction is less critical but TRNSYS retains the designation of time dependent inputs and time independent inputs. Time dependent inputs are referred to as INPUTS while time independent inputs are referred to as PARAMETERS.

At each iteration and at each time step, a component turns the current values of the INPUTS and PARAMETERS into OUTPUTS. No distinction is made among OUTPUTS; all OUTPUTS are assumed to be time dependent and are recomputed by a component whenever appropriate.

TRNSYS has one more input / output distinction; that is DERIVATIVES. Components that solve differential equations numerically will often have DERIVATIVES as well as INPUTS, OUTPUTS and PARAMETERS. From the simulation user's point of view, the DERIVATIVES for a given Type specify initial values, such as the initial temperatures of various nodes in a thermal storage tank or the initial zone temperatures in a multi zone building. At any given point in time, the DERIVATIVES of a Type hold the results of the solved differential equation. If these DERIVATIVES are set in a component as OUTPUTS, the simulation user can see, plot and output these values.

8.3.1.1. The INFO Array and component control

Much of the control as to what a component is supposed to do at a given point in the simulation (particularly during initialization) is controlled by the contents of a data structure called the "INFO array." This array carries in its 15 spots, a great deal of information about what is going on globally in the simulation at a given time. For example, the first INFO spot (INFO(1)) carries the current UNIT number that the kernel is calling. The second INFO spot (INFO(2)) carries the current TYPE number that the kernel is calling. The seventh INFO spot has a value of –1 if the current call from the kernel intends for the called Type to initialize itself. The same spot has a value of 0 at the very first call to a Type in a given time step. Thereafter it carries the total number of iterations that have gone by in the current time step. The eighth INFO spot carries the total number of calls to each Type that have gone by in the simulation at any given time. The thirteenth INFO spot is normally 0 but is set to 1 after convergence has been reached in a given time step. A complete reference to the contents of the INFO array is available in section 8.4.3 of this manual. The INFO array will be used heavily in this section as a method for controlling the functionality of a Type at various points in a simulation.

The remainder of this section of the Programmer's Guide will use the example of a simple liquid heating device to illustrate the process of writing a new component for TRNSYS.

Consider a simple heater that raises the variable inlet temperature of a flowing liquid to a user defined temperature T_{set}. Writing an energy balance on the fluid of inlet temperature, T_{in}, at a mass flow rate $\dot{m}$:

$$\dot{Q} = \dot{m} C_p (T_{set} - T_{in}) \qquad \textbf{Eq. 8.3.1-1}$$

The first decision that must be made by the Type programmer is: what are going to be the PARAMETERS, INPUTS and OUTPUTS of the model. Two possibilities become apparent. If the goal of the component is for the end user to be able to specify an inlet temperature and a desired outlet temperature, and in return, find out how much energy was required to bring the liquid from its inlet condition to its outlet condition, then $\dot{Q}$ should obviously be an output, while $\dot{m}$, T_{set}, and T_{in} should be INPUTS. C_p could be designated as either a PARAMETER or as an INPUT depending upon whether or not the liquid specific heat can be assumed to be constant or whether it varies significantly with liquid temperature. If, on the other hand, the goal of the component is for the end user to provide inlet conditions, a control signal and a heater output that is full ON whenever the control signal is set to ON, then perhaps the output of the model would be Tout, and the inputs would be $\dot{m}$, T_{in}, and $\dot{Q}$.

For the purposes of this example, the INPUTS will be T_{in}, T_{set}, and $\dot{m}$, the OUTPUTS of interest will be $\dot{Q}$, the instantaneous heating rate and the outlet temperature, T_{out}. The PARAMETERS characterizing the heater will be Tset, Cpf, and Cap, allowing the heater to be capacity limited in the amount of energy that it can deliver to the liquid stream. Note that $\dot{m}$ will also be an OUTPUT so that this information is available to other components. We begin at the top of the new component subroutine.

8.3.2. The first line

Every Type begins with the same line:

```
SUBROUTINE TYPEn(TIME,XIN,OUT,T,DTDT,PAR,INFO,ICNTRL,*)
```

Where “n” in TYPEn is a number between 1 and 999. To avoid problems with more than one component being assigned the same number, the following conventions have been adopted:

- Type1 – 150: reserved for library of Standard TRNSYS Components
- Type151 – 200 : reserved for user written components
- Type201 – 300: reserved for add on components commercially available from Transsolar
- Type301 – 400: reserved for add on components commercially available from CSTB
- Type401 – 500: reserved for add on components commercially available from Aiguasol
- Type501 – Type800: reserved for add on components commercially available from Thermal Energy System Specialists
- Type801 – Type999: not reserved

For this example, Type number 151 will arbitrarily be chosen. Thus the first line of the subroutine should be:

```
SUBROUTINE TYPE151(TIME,XIN,OUT,T,DTDT,PAR,INFO,ICNTRL,*)
```

In the subroutine parameter sequence shown above, the following variables are used:

TIME	(double precision) the simulation time
XIN	the double precision array containing the component INPUTS
OUT	the double precision array which the component fills with its appropriate OUTPUTS
T	the double precision array containing the dependent variables for which the derivatives are evaluated in the particular model
DTDT	the double precision array containing the derivatives of T which are evaluated by the model
PAR	the double precision array containing the PARAMETERS that characterize the component
INFO	the integer array described in Section 8.4.3
ICNTRL	the integer array described in Section 8.4.7
*	the alternate return described in Section 8.4.6

8.3.3. DLL Export

With the release of TRNSYS v. 16.0, a great deal of effort was put into allowing Types to exist either in the standard TRNDll.dll along with the other standard Types or to exist in an External DLL. The TRNSYS 16 kernel examines the contents of a particular user library directory at the beginning of each simulation where all external DLLs must be located if they are to be used. It then loads all files with the extension *.dll, and examines them to find out if there are any exported Type subroutines contained therein. If it finds exported Type routines it loads them into memory for the duration of the simulation. The main advantage to end users is that they no longer have to recompile the TRNDll.dll file when they receive a new component that they wish to use in their simulation. Instead, they merely need to drop a DLL containing that Type into the appropriate directory. As a Type programmer, there are certain steps that you must follow in order to compile your Type and link it as an external DLL. For more information on linking your component into a DLL, please refer to sections 8.3.16 and 8.5.2 of this manual. From a code point of view, however, you need only add one line to your component, right at the top of the file after the SUBROUTINE declaration and before any USE statements. The syntax of the line is:

```
!DEC$ATTRIBUTES DLLEXPORT :: TYPEn
```

Where n is the Type number. For our example (Type151) we would have the following line added:

```
...
!DEC$ATTRIBUTES DLLEXPORT :: TYPE151
...
```

It is important that the syntax begin in the leftmost column. In other words, do not put any tabs or spaces to the left of the first exclamation point. The exclamation mark is recognized as a comment by the Fortran code. It is, however, noticed by the Fortran compiler as an indication that this subroutine should be exported for access by other DLLs.

8.3.4. USE Statements

Whereas previous versions of TRNSYS relied heavily on Fortran COMMON blocks, TRNSYS 16 (and later version) Types should access global constants through various Data Modules that are included with the source code. A Data Module is really nothing more than a file that contains a series of variable declarations; they are a convenient way of declaring all in one place, the variables that are accessible by various subroutines (Types, and kernel routines) within a program. If a variable contained within a data module is needed by a given component, it is accessed by adding a “USE” statement directly after the DLL Export instruction (see section 8.3.3 above). The syntax of a USE statement is as follows:

```
USE NameOfDataModule
```

The above has the effect of declaring ALL of the variables contained within the module in your Type. Alternatively, if you only want to use a particular variable that is contained within the module but do not want access to all of the variables therein, you can use the syntax:

```
USE NameOfDataModule, ONLY: nameOfVariable1,nameOfVariable2
```

This will only declare the variable(s) listed after the "ONLY" keyword. The advantage of listing only those variables to which we want access is that we can reuse the names of global variables locally as long as we do not request access to them through a USE statement. A complete list of Data Modules accessible from a given Type or subroutine of a Type are listed in sections 8.4.1, 8.4.2. The most common Data Modules that you will need to access from your Type are: TrnsysFunctions, TrnsysConstants, and TrnsysData. TrnsysFunctions contains a series of functions that return commonly useful data such as the simulation start time or time step, or which perform common functions such as reserving a new logical unit number in a global list for use by an external file accessed by your Type. The TrnsysConstants module contains values that do not change throughout a simulation but which are again, of common interest. The TrnsysData module carries a series of data structure sizing constants such as the number of allowable UNITS in a given simulation or the number of allowable EQUATIONS in a given simulation.

For the purposes of our example, the liquid heater will only need some of the TRNSYS's Access Functions. Thus, in the USE statement section of the code, we need only add the following line:

```
        ...
! USE STATEMENTS
        USE TrnsysFunctions
        ...
```

As previously mentioned, it is permissible to list the names of those functions that we wish to access from the TrnsysFunctions module. However, since there is little chance that we will be writing Functions of our own in this component, the chances that the name of a local function and the name of a global function will conflict, is minimal.

8.3.5. Variable Declarations

Having defined the subroutine arguments, exported the component, and given the component access to global TRNSYS variables, we next need to declare all of the local variables that we intend to use in the Type. Local variables can be split into two categories; local variables that are required by every Type, and local variables that are only required by a given Type. Those local variables that are required by every Type include the array that contains the Type's parameters, the array that contains the Types inputs, the array that contains the Type's outputs, the simulation time, etc. This block of variable declarations can simply be copied from an existing Type and can be pasted into your new one. You only need make one modification to the following; that is to change the values of NP, NI, NO, ND, and NS to match the number of parameters, inputs, outputs, derivatives, and storage spots that will be required for your Type. The block of local variables required by every Type is as follows.

```
        ...
        IMPLICIT NONE !FORCE DECLARATION OF ALL VARIABLES

!       TRNSYS DECLARATIONS
        DOUBLE PRECISION XIN,OUT,TIME,PAR,T,DTDT,TIME0, DELT,STORED
        INTEGER*4 INFO(15),NP,NI,NOUT,ND,IUNIT,ITYPE,ICNTRL
        CHARACTER*3 OCHECK,YCHECK

!       SET THE MAXIMUM NUMBER OF PARAMETERS(NP),INPUTS(NI),OUTPUTS(NO),
!       DERIVATIVES(ND), AND STORAGE SPOTS(NS) THAT MAY BE SUPPLIED FOR THIS TYPE
        PARAMETER (NP=a,NI=b,NO=c,ND=d,NS=e)

!       REQUIRED TRNSYS DIMENSIONS
        DIMENSION XIN(NI),OUT(NO),PAR(NP),YCHECK(NI),OCHECK(NO),STORED(NS)
        ...
```

For the purposes of our liquid heater component, NP will be set to 4 (four parameters are required), NI will be set to 5 (five inputs are required), NO will be set to 5 (five outputs will be calculated), ND will be set to 0 (no derivatives are required), and NS will be set to 0 (no storage spots are required).

With the local variables that apply to all components declared, we next declare the variables that pertain only to our liquid heater. From a practical point of view, there is no way to anticipate and simply enter all of the variables that will be needed during the course of developing a new component. The following list is almost always created as the Type is being written and as local variables are needed.

```
        ...
!       LOCAL VARIABLE DECLARATIONS
        INTEGER IGAM
        DOUBLE PRECISION TIN, & !temperature of fluid at heater inlet [C]
        TOUT, &   !temperature of fluid at heater outlet [C]
        TBAR, &   !average temperature of fluid in heater [C]
        TAMB, &   !ambient temperature of heater surroundings [C]
        TSET, &   !heater setpoint temperature [C]
        TON, &    !set temporarily to TOUT before check on TOUT>TSET [C]
        QMAX, &   !heater capacity [kJ/hr]
        QAUX, &   !required heating rate [kJ/hr]
        QLOSS, &  !rate of thermal losses to surroundings [kJ/hr]
        FLOW, &   !fluid flow rate through heater [kg/hr]
        CP, &     !fluid specific heat [kJ/kg.K]
        HTREFF, & !heater efficiency [-]
        UA, &     !overall loss coeff. for heater during operation [kJ/hr.K]
        QFLUID    !rate of energy delivered to fluid [kJ/hr]
        ...
```

8.3.6. *Data Statements*

It is advantageous in some Types to set some constants that will be used throughout the Type outside of the executable portion of the code using a DATA statement. The advantage of doing so is that by defining a constant in a DATA statement, that definition is loaded at the simulation start and never is there a time when simulation time has to be spent setting or resetting those values. Often times Type programmers will set a variable called PI to the value 3.1415926 in a DATA statement, or they will set a variable for the conversion factor between degrees and radians in a DATA statement. Any variable whose value is set in a DATA statement must be declared among the Local Variable declarations discussed in section 8.3.5 above.

One other common set of constants that are declared in DATA statements before the beginning of the executable portion of a Type are the three letter codes that set the units of the Type's INPUT and OUTPUT variables. Refer to section 8.4.4.12 for additional information on these three letter codes.

We have already declared and sized the two CHARACTER*3 arrays that will hold the input and output unit codes for the liquid heater among the definition of local variables that are required by all Types (see section 8.3.5). Our list of inputs, their units, and our list of outputs and their units is as follows:

Table 8.3.1.1-1: Inputs, Outputs, and their Units

Input Number	Input Description	Units	Output Number	Output Description	Units
1	Liquid inlet temperature	C	1	Liquid outlet temperature	C
2	Liquid inlet mass flow rate	kg/hr	2	Liquid outlet flow rate	kg/hr
3	Control signal	0..1	3	Required heating rate	kJ/hr
4	Temperature set point	C	4	Energy lost to ambient	kJ/hr
5	Ambient temperature	C	5	Energy to liquid stream	kJ/hr

Referring to Table 8.3.1.1-1 and to Table 8.4.4.12-1 through Table 8.4.4.12-22, the following DATA statements can be added to the Type, thus setting the units of the first input as a temperature in degrees C, the units of the second input as mass flow rate in kilograms per hour, the third input as a control signal having a value between 0 and 1, etc.

```
       ...
!      DATA STATEMENTS
       DATA YCHECK/'TE1','MF1','CF1','TE1','TE1'/
       DATA OCHECK/'TE1','MF1','PW1','PW1','PW1'/
       ...
```

8.3.7. Global Constants and Variables

We have now arrived at the executable section of the Type; from here on, each line of the code will be executed sequentially, starting at the top. It is therefore necessary to put those lines that should be executed every single time step and every single iteration right at the beginning of the executable section. In the case of almost all Types, the only thing that needs to be done this regularly is to retrieve the values of some global variables that are used in controlling what the Type does at what point in the simulation. Much of this information has already been passed to the component. Variables such as TIME, and arrays such as INFO, PAR, and XIN are already passed to us. Others, such as the time step, the simulation start time, and the simulation stop time are not passed but must be retrieved on an "as needed" basis. This is the point in the Type where AccessFunctions will be used most heavily. Almost all Types, this one included, need to know the simulation start time and the simulation time step. These two variables are used later to control Type functionality (see section 8.3.12 here below). Again, the local variable names for these two variables have already been declared in the block of local variables that are required by every Type. The local variable name for the simulation time step is "DELT" and the local variable name for the simulation start time is "TIME0" The code that should be added to Type151 is as follows:

```
!-----------------------------------------------------------------------------
!      GET GLOBAL TRNSYS SIMULATION VARIABLES
       TIME0   = getSimulationStartTime()
       DELT    = getSimulationTimeStep()
```

8.3.8. Version Signing

Version signing is one of a series of special calls that is made to a Type when the Type is not intended to perform any manipulations of inputs or outputs. Once, at the very beginning of a simulation, a Type needs to log itself as having been written for a given version of TRNSYS. Throughout TRNSYS's history, new requirements have periodically been added to the recommended practices in coding components. These requirements reflect new enhancements that have been added or reflect changes is recommended coding practices that come from the developers of Fortran compilers. Prior to the release of TRNSYS version 16, it was left to the Type programmer to simply make the required modifications before being able to use an existing Type with the new version of TRNSYS. With the release of TRNSYS 16.0, an effort was made to allow for backwards compatibility in Types. That is that the TRNSYS 16 kernel was designed in such a way that it was able to deal with Types that were written to conform with the TRNSYS 16 coding requirements, and to deal with Types that had been written for TRNSYS 15 (which have different coding requirements). In order for the kernel to know how to treat a Type, however, each Type must log itself as having been written with TRNSYS 16 conventions or as having been written with TRNSYS 15 (and earlier) conventions. For additional information about the differences in conventions, please refer to section 8.2 of this manual.

For the purposes of the liquid heater example, we are writing a TRNSYS 16 style component so we must add lines of code that "version sign" our Type as complying with the TRNSYS 16 coding requirements. The code lines that must be added are:

```
...
!-----------------------------------------------------------------------------
```

```
!    SET THE VERSION INFORMATION FOR TRNSYS
      IF(INFO(7).EQ.-2) THEN
         INFO(12)=16
         RETURN 1
       ENDIF
...
```

At the very beginning of a TRNSYS simulation, the TRNSYS kernel calls each Type present in the input file with the seventh spot of the INFO array set to a value of –2. Every Type should therefore have a line in it that watches specifically for this situation. The only thing that the Type should do when it realizes that INFO(7) = -2 is to set the twelfth input spot to a value of 16 if the Type was written using TRNSYS 16 coding conventions or to a value of 15 if the Type was written using previous coding conventions. If a Type does not handle the INFO(7) = -2 call correctly (does not set INFO(12) to either 15 or 16) the TRNSYS kernel will not allow the simulation to proceed and will generate an error, writing it to the simulation list and log files.

8.3.9. *Last Call Manipulations*

The second special case call to Types occurs at the very END of each simulation. Whether the simulation ends correctly or ends in error, each Type is recalled by the TRNSYS kernel before the simulation shuts down. This call allows Types to perform any “last call” manipulations that may be necessary. These may include closing external data files that were opened by the Type during the course of a simulation, or calculating summary information that may have been collected during a simulation. The Type that we are writing, does not require any such last call manipulations, so it is advisable to simply return control directly to the TRNSYS kernel. The code that is added to Type151 is as follows:

```
...
!-----------------------------------------------------------------------------
!      PERFORM LAST CALL MANIPULATIONS
       IF (INFO(8).EQ.-1) THEN
         RETURN 1
       ENDIF
...
```

You may want your component to actually do something at the end of the simulation. It could, for instances print a message to the list file or close logical units that were used during the simulation. Standard component Type 22 (Iterative Feedback controller) performs end of the simulation manipulations that you might use as an example.

The INFO(8)=-1 call also occurs if the simulation terminates with a fatal error. In that case, the component that generates the error returns control to TRNSYS, which calls all components one last time in order to perform their "end of simulation" operations. You can check if the very last call occurs because of an error or as part of the normal simulation process by calling getNumberOfErrors(). Some end of simulation operations are unnecessary or might crash TRNSYS if the simulation ends with a fatal error. E.g. Type 22 handles INFO(8)=1 as follows:

```
if (info(8) == -1) then
    ! Exit immediately if this call is the result of a fatal error
    if (getNumberOfErrors() > 0) then
        return 1
    ! Otherwise, print the nb of "stuck" timesteps
    else
        ... printing manipulations
        ...
    endif
    return 1    ! Exit
endif
```

Notes:

- If the simulation ends without errors, the INFO(8) = -1 call happens ***after the user has allowed the simulation to terminate*** by clicking on the "yes" or "continue" button at the end of the simulation
- The lines of code here above should be placed ***before the normal instructions*** so the return occurs before those instructions are executed.

8.3.10. Post Convergence Manipulations

At the end of each time step, each Type in a simulation is recalled with the thirteenth spot in the INFO( ) array set to a value of 1. During all other calls (during the iterations of a given Type during a given time step) this value is set to zero. In this case of the liquid heater example, no post convergence manipulations are needed. Therefore, we can enter the following code at this point in the component:

```
...
!-----------------------------------------------------------------------------
!      PERFORM POST CONVERGENCE MANIPULATIONS
       IF(INFO(13).GT.0) THEN
         RETURN 1
       ENDIF
...
```

In more complex components than the one detailed by this example, the post convergence call allows a number of different manipulations to be performed. They are resetting counters and updating storage variables.

8.3.10.1. Resetting counters

It is sometimes advantageous to count the number of times that a particular event has occurred during a given time step. One example might be a controller that counts the number of times that it has calculated a different value of its control signal during a given time step. After a certain number of different decisions, the controller “sticks” and the calculated output state no longer changes. At the end of the time step after all components have converged, it is necessary to reset the iteration counter to zero. This could be done as one of the Post Convergence Manipulations when INFO(13) is set to 1.

8.3.10.2. Updating storage variables

One of the most common Post Convergence Manipulations performed by Types is that of updating storage variables. For background information on updating storage variables, please refer to section 8.4.4.16. One common application of updating storage variables at the end of a time step is for a Type to store two values of a variable; the initial value of the variable (at the beginning of the time step) on which all calculations at the subsequent time step are based and the final value of the variable, which is reevaluated at every iteration. At the end of the time step, when all components have converged, it is necessary to replace the initial value with the converged final value. The code employed might look like the following:

```
...
!-----------------------------------------------------------------------------
!      PERFORM POST CONVERGENCE MANIPULATIONS
       IF(INFO(13).GT.0) THEN
         CALL getStorageVars(Stored,2,INFO)
         OldInitialValue = Stored(1)
         NewFinalValue = Stored(2)  !NewFinalValue was calculated by the
                                    !component at each iteration in the time
                                    !step and was stored in the Stored(2) spot.
         Stored(1) = NewFinalValue
         CALL setStorageVars(Stored,2,INFO)
         RETURN 1
       ENDIF
...
```

8.3.11. Initialization Call Manipulations

At the very start of each simulation, after all Types have version signed themselves (see section 8.3.8 or section 8.4.3.3 for more information), each Type is called with the seventh spot of the INFO( ) array set to –1, indicating to all Types that they should initialize themselves. There is a very specific set of steps that each Type should take at this point in the simulation. With the release of TRNSYS version 16, the steps taken by a Type during the initialization step changed from what had been done previously. The most important thing to keep in mind concerning the initialization call is that a Type should *NOT* read its parameter list, nor should it perform *ANY* calculation of output values. It should perform the following operations

Reserve the required amount of space in the output array using INFO(6). Since we have already set the expected number of outputs as a Fortran parameter in section 8.3.5, there is no need to do anything except add a line that says:

```
INFO(6) = NO
```

Set how the Type should be called using INFO(9). Most Types will set INFO(9) to a value of 1, indicating that the Type should be called every iteration whether or not its input values have changed. Types that are integrators or printers use different INFO(9) settings. Users programming this kind of Type should refer to section 8.4.3.5 for more information.

Set the number of single precision storage spots required using INFO(10). It is highly recommended that all components make use of DOUBLE, not SINGLE precision storage structures. Therefore, in all likelihood, you should simply set INFO(10) to a value of 0.

Set the number of double precision storage spots required by calling the setStorageSize subroutine. Information on calling setStorageSize can be found in section 8.4.4.16. It is again important to bear in mind that we are not setting initial values of the storage variables at this time, we are merely reserving space for later usage.

Call the TYPECK routine to make sure that the TRNSYS user has supplied the correct number of parameters, inputs, and derivatives in the input file. Again, since we have already set the expected number of parameters, inputs and derivatives using a Fortran parameter statement in section 8.3.5, we need only paste in a line of the proper syntax at this point as shown below. The 1 in the fist argument of the call to TYPECK is an indicator of the action that TYPECK should take in this particular case. For additional information on the arguments to TYPECK and their meanings, please refer to section 8.4.4.19.

```
CALL TYPECK(1,INFO,NI,NP,ND)
```

Call the RCHECK routine to make sure that the units of the outputs that are connected to the inputs of this Type are correct. In section 8.3.6, we listed three letter codes for the units of each of the inputs and outputs of the Type151 that we are writing. By calling RCHECK, we are allowing TRNSYS to check the users connections between other Types and this one to make sure that they did not inadvertently connect an output with units of power (‘PW1’ for example) to an input with units of temperature (‘TE1’ perhaps)

Return control to the TRNSYS kernel. Again, there are NO iterations performed during this time step. Therefore there should be NO calculations performed.

The code required for the above steps in the context of Type151 is as follows. Please note that the following code is quite generic and can be copied for any Type that does not have a variable number of parameters, inputs, or derivatives allowed in the input file.

```
...
!-----------------------------------------------------------------------------
!      PERFORM INITIALIZATION MANIPULATIONS
       IF (INFO(7).EQ.-1) THEN
          !retrieve unit and type number for this component from the INFO array
         IUNIT=INFO(1)
         ITYPE=INFO(2)
         !reserve space in the global OUT array
         INFO(6) = NO
         ! this TYPE should be called at each iteration whether or not its
         ! inputs change
         INFO(9) = 1
         ! reserve space in the single precision storage structure
         INFO(10) = 0
         ! call the Type check subroutine to compare what this Type requires to
         ! what has been supplied in the input file.
         CALL TYPECK(1,INFO,NI,NP,ND)
         ! call the input-output check subroutine to set the correct input and
         !output variable units.
         CALL RCHECK(INFO,YCHECK,OCHECK)
         ! return to the calling program.
         RETURN 1
       ENDIF
...
```

8.3.12. TIME = TIME0 Manipulations

Once the initialization steps have been completed, control is returned to the TRNSYS kernel. The next time that each Type is called is when the simulation time is equal to the simulation start time. Throughout much of TRNSYS’s history (through version 15) there was no call to components at the simulation start time. After being called with INFO(7) = -1 for initialization (as described in section 8.3.11) Types were next called at the end of the first time step (TIME = InitialTime + TimeStep). With the release of TRNSYS 16.0, an important distinction was made by now calling all Types once after they have been initialized but before time has advanced in the simulation. The point of this initial time call is to perform the following actions:

First, read the local parameter list and set each parameter to a local variable. As described in section 8.3.2, one of the data structures passed by the TRNSYS kernel to each Type is a one dimensional array containing that Type’s parameter values. Parameter values can (and should) be read only at two places in a Type. The first is during this time = time0 call before time has progressed. The other is during the “multiple unit manipulations” section (discussed below in section 8.3.13) Only reading parameters at these two stages of the simulation saves a great deal of computational time because in this manner parameters (which do not change with time) are read only when they must be. In each of these two sections, each parameter value is set equal to a local variable name that has been declared as a double precision variable among the Type’s local variables as discussed in section 8.3.5. The generalized code for setting a parameter value to a local variable name is

```
LocalVariable = PAR(n)
```

Type151 has four parameters, one each for the maximum heating rate, the specific heat of the working liquid, the overall loss coefficient of the heater, and the heater efficiency. Referring to the code at the end of this section, each of these four parameters (PAR(1) through PAR(4) is set to a local variable name: QMAX, CP, UA, and HTREFF respectively.

The second step performed during the “time = time0 manipulations” section is to check each of the parameter values for validity. Parameters whose values cannot be negative, or whose values must be between 0 and 1, can be flagged as an error using a call to the TYPECK subroutine. Take, for example, the specific heat of the working liquid for the heater. It does not make any physical sense to allow the user to enter a negative value of specific heat. We could use the following code to detect and report the problem, assuming that the second parameter is the specific heat and has been set to a local variable called SpecHeat.

```
...
SpecHeat = PAR(2)
IF (SpecHeat.LT.0.0) CALL TYPECK(-4,INFO,0,2,0)
...
```

The “-4” argument in the call to TYPECK indicates to TYPECK that we are flagging an incorrect parameter value and that we want TYPECK to report the error for us using its generic error message. The “2” argument indicates to TYPECK that it is the second parameter that is a problem.

Once TYPECK has reported the flagged parameter and printed its error message to the simulation log and list files, it returns control to the calling Type but unfortunately does not return an argument that tells the Type that an error message was written. It is good practice for a Type that has called TYPECK with a bad parameter to immediately return control to the TRNSYS kernel without completing its TIME = TIME0 manipulations in case the bad parameter could cause other problems before the end of the call. It is therefore recommended that once all the parameters have been checked, programmers use the ErrorFound( ) Access Function to determine whether TYPECK wrote any error messages to the simulation list and log files. For more information on the ErrorFound( ) Access Function, please refer to section 8.4.2.6. The code for returning control in this manner is:

```
...
IF (ErrorFound()) RETURN 1
...
```

Next, it is necessary to set the initial values of any storage variables that are used by the Type. This is accomplished simply by assigning local variable values to each of the spots in the array that is used to transfer information to and from the double precision storage structure, then calling the setStorageVars subroutine. The code used to set two initial storage values might look like the following:

```
...
storageArray(1) = LocalVar1
storageArray(2) = LocalVar2
CALL setStorageVars(storageArray,2,INFO)
...
```

Since our Type151 does not require any storage variable spots, there is no need to include the above step in this section of the Type151 code.

The final step performed during the TIME = TIME0 Manipulations section is to set the initial values of the outputs. It is essential that you set initial values here, your Type should *not* perform any of its calculations at this point in time. If it does, then it will likely get one time step ahead of where it is supposed to be at this point in the simulation. Because of this restriction on performing calculations, it is often difficult to know what value to choose for an output initial value. A good default value is zero since that usually indicates to the user that the output has not yet been calculated. In the case of devices that have fluid flow through them, the outlet temperature and outlet flow rate of the fluid can often be set to the inlet temperature and inlet flow rate respectively. In the case of Type151, the first two outputs can be set to the input initial values in this manner. The other outputs cannot be calculated easily and so are simply set to zero.

Below is the entire code for the TIME = TIME0 Manipulations section of Type151.

```
...
!-----------------------------------------------------------------------------
!      PERFORM INITIAL TIMESTEP MANIPULATIONS
       IF(TIME.LT.(TIME0+DELT/2.)) THEN
         !set the UNIT and TYPE numbers
         IUNIT = INFO(1)
         ITYPE = INFO(2)
         !read parameter values
         QMAX   = PAR(1)
         CP     = PAR(2)
         UA     = PAR(3)
         HTREFF = PAR(4)
         !check the parameters for problems and RETURN if any are found
         IF(QMAX.LT.0.) CALL TYPECK(-4,INFO,0,1,0)
         IF(CP.LT.0.)   CALL TYPECK(-4,INFO,0,2,0)
         IF(UA.LT.0.)   CALL TYPECK(-4,INFO,0,3,0)
         IF((HTREFF.GT.1.).OR.(HTREFF.LT.0.)) CALL TYPECK(-4,INFO,0,4,0)
         IF (ErrorFound()) RETURN 1
         !perform any required calculations, set the outputs to appropriate
         ! initial values.
         OUT(1) = XIN(1) !outlet temperature = inlet temperature [C]
         OUT(2) = XIN(2) !mass flowrate out = mass flowrate in [kg/hr]
         OUT(3) = 0.     !required heating rate [kJ/hr]
         OUT(4) = 0.     !rate of losses to environment [kJ/hr]
         OUT(5) = 0.     !rate of energy delivered to stream [kJ/hr]
         RETURN 1        !the first timestep is for initialization - exit.
      ENDIF
...
```

8.3.13. Multiple Unit Manipulations

Only one more special manipulations section is required before we begin actually simulating the performance of the liquid heater. In this Multiple Unit Manipulations section, we must reread our components parameter list if and only if there is more than one instance of the component in the users simulation.

You may have noticed in some of the previous sections that often, one of the first few lines of code in each section says:

```
!set the UNIT and TYPE numbers
IUNIT = INFO(1)
ITYPE = INFO(2)
```

What this does is take the first two spots of the INFO( ) array each and every time the Type is called and save the current Type number and the current Unit number to local variables ITYPE and IUNIT respectively. If there is only one instance of a given Type in a simulation, then the value of IUNIT will always be equal to the value of INFO(1). Conversely, if there is more than one instance of a given Type in a simulation, then whenever we begin simulating a new instance of that Type, the value of IUNIT will not match the value of INFO(1). We make use of this very feature at this stage of the manipulations. If the unit number has changed (IUNIT does not equal INFO(1) then we know that we need to reread that parameter list so that we are sure that all local variables are set to the current instance of the Type's parameter lists. The code is as follows:

```
...
!-----------------------------------------------------------------------------
!      RE-READ THE PARAMETERS IF ANOTHER UNIT OF THIS TYPE HAS BEEN CALLED SINCE
!      THE LAST TIME THEY WERE READ IN
       IF(INFO(1).NE.IUNIT) THEN
         !reset the unit number
         IUNIT  = INFO(1)
         ITYPE  = INFO(2)
         !read the parameter values
         QMAX   = PAR(1)
         CP     = PAR(2)
         UA     = PAR(3)
         HTREFF = PAR(4)
       ENDIF
       ...
```

There is no need to recheck the parameters for validity since this was done already during the TIME = TIME0 manipulations section.

8.3.14. Every Time Step Manipulations

At last we have come to the meat of writing new components. It is in this final section that the performance of the component is simulated. Because every component is different, the rules for what steps go in this section become much more flexible. That said, the “Every Time Step Manipulations” section can be broken up into four basic categories which proceed in order as follows.

8.3.14.1. Retrieve Stored Values

At each iteration, a call should be made to the getStorageVars subroutine to retrieve stored values that are needed in calculations. The call might look like the following:

```
...
CALL getStorageVars(storageArray,NS,INFO)
LocalVar1 = storageArray(1)
LocalVar2 = storageArray(2)
...
```

8.3.14.2. Retrieve Input Values

Retrieving input values follows much the same pattern as that of retrieving parameter values. The inputs are passed to each Type in a double precision, one dimensional array called XIN( ). The values of this array are typically read to local variable names for convenience and then can be checked for validity in much the same way as parameters.

Our Type151 has five inputs. However, only two of them have restrictions on their values. Namely, the flow rate should not be negtive, and the heater control signal cannot be less than zero or greater than one. The section that reads them from the XIN( ) array, sets the inputs to local variables and checks them might look like the following.

```
...
!-----------------------------------------------------------------------------
!      RETRIEVE THE CURRENT INPUT VALUES FROM THE XIN ARRAY
       TIN  = XIN(1)
       FLOW = XIN(2)
       IGAM = JFIX(XIN(3)+0.1)
```

```
      TSET = XIN(4)
      TAMB = XIN(5)
!     CHECK THE INPUTS FOR VALIDITY AND RETURN IF PROBLEMS ARE FOUND
      IF (FLOW.LT.0.)                         CALL TYPECK(-3,INFO,2,0,0)
      IF ((IGAM.GT.1.).OR.( IGAM.LT.0.))      CALL TYPECK(-3,INFO,3,0,0)
      IF (ErrorFound() ) RETURN 1
...
```

Here the –3 in the TYPECK arguments indicates that we are flagging a bad input value (previously we used –4 to indicate that we were flagging a bad parameter value.). The 2 and the 3 indicate the input number that is the problem in each case.

8.3.14.3. Perform Calculations

Once all of the input values have been retrieved, the heater performance calculations and control logic can be performed. Each line in the following code is commented to show its purpose.

```
!     PERFORM CALCULATIONS
      IF (FLOW .GT. 0.) GO TO 10 !if the inlet flow is greater than 0, skip to
                                 !line 10

!     NO FLOW CONDITION
      OUT(1) = TIN        !set the outlet temperature to the inlet temperature
      OUT(2) = 0.         !set the outlet flow rate to 0.
      OUT(3) = 0.         !set auxiliary energy use to 0.
      OUT(4) = 0.         !set heater losses to 0.
      OUT(5) = 0.         !set the energy imparted to the liquid to 0.
      RETURN 1            !return control to the kernel

!     FLOW CONDITION
!     CHECK INLET TEMPERATURE AND CONTROL FUNCTION
10    IF ((TIN.LT.TSET).AND.(IGAM.EQ.1)) GO TO 20 !if the inlet temperature is
                                                  !below the set point temperature
                                                  !otherwise, the heater and the
                                                  !control signal is set to ON,
                                                  !skip to line 20. is OFF.

!     HEATER “OFF” CONDITION
      TOUT   = TIN        !set the outlet temperature to the inlet temperature
      QAUX   = 0.         !set the auxiliary energy use to 0.
      QLOSS  = 0.         !set the heater losses to 0.
      QFLUID = 0.         !set the energy imparted to the liquid to 0.
      GO TO 50            !skip to the output setting section.

!     HEATER “ON” CONDITION
!     calculate the outlet temperature assuming that the heter is on and
!     running at capacity
20    TON  =(QMAX*HTREFF+FLOW*CP*TIN+UA*TAMB-UA*TIN/2.d0)/(FLOW*CP+UA/2.d0)
      !the outlet temperature is the lesser of TON and TSET – the heater is
      !able to modulate its energy output to reach the setpoint temperature.
      TOUT = MIN(TSET,TON)
      !calculate the average heater temperature – losses are based on the
      !average
      TBAR = (TIN+TOUT)/2.d0
      !calculate the auxiliary energy required, accounting for heater
      !efficiency.
      QAUX = (FLOW*CP*(TOUT-TIN)+UA*(TBAR-TAMB))/HTREFF
      !calcualte the energy lost from the heater, including heater inefficiency
      QLOSS = UA*(TBAR-TAMB) + (1.d0-HTREFF)*QAUX
      !calculate the energy imparted to the liquid.
      QFLUID = FLOW*CP*(TOUT-TIN)
```

8.3.14.4. Set Storage Values

Before exiting the Type at each iteration, it is good practice to update all appropriate storage variables. Obviously, if one or more of the storage variables is an initial value to be used for all iterative calculations at a given time step until convergence has been reached then it is not appropriate to update this value until the INFO(13) = 1 call (see sections 8.3.10 and 8.4.3.9 for additional information.

8.3.14.5. Set Outputs

In much the same way that we took the TRNSYS kernel provided parameter and input arrays and set them to local variable names, we now have to set the calculated local variables to the appropriate TRNSYS output array. The code to do so for Type151 is as follows:

```
...
!      SET OUTPUTS
50     OUT(1) = TOUT
       OUT(2) = FLOW
       OUT(3) = QAUX
       OUT(4) = QLOSS
       OUT(5) = QFLUID
...
```

8.3.14.6. Return Control

The very last requirement is that a Type return control to the TRNSYS kernel after its computations are complete. The “1” following the RETURN statement is referred to as an “Alternate Return.” In essence, a call to a FORTRAN subroutine can be set up so that under one set of circumstances, control is returned to the line following the call line. Under other circumstances, an “alternate return” to return control to some other point in the calling routine. For additional information on the alternate return statement, please refer to section 8.4.6.

```
...
RETURN 1
END SUBROUTINE Type151
...
```

8.3.15. Complete Code

The entire code of Type151 is reproduced here below as a complete reference.

```
SUBROUTINE TYPE151(TIME,)
!-------------------------------------------------------------------------------

!DEC$ATTRIBUTES DLLEXPORT :: TYPE151

! USE STATEMENTS
      USE TrnsysFunctions

      IMPLICIT NONE !FORCE DECLARATION OF ALL VARIABLES

!     TRNSYS DECLARATIONS
      DOUBLE PRECISION XIN,OUT,TIME,PAR,T,DTDT,TIME0,TFINAL,DELT,STORED
      INTEGER*4 INFO(15),NP,NI,NOUT,ND,IUNIT,ITYPE,ICNTRL
      CHARACTER*3 OCHECK,YCHECK

!     SET THE MAXIMUM NUMBER OF PARAMETERS(NP),INPUTS(NI),OUTPUTS(NO),
!     DERIVATIVES(ND), AND STORAGE SPOTS(NS) THAT MAY BE SUPPLIED FOR THIS TYPE
      PARAMETER (NP=a,NI=b,NO=c,ND=d,NS=e)

!     REQUIRED TRNSYS DIMENSIONS
      DIMENSION XIN(NI),OUT(NO),PAR(NP),YCHECK(NI),OCHECK(NO),STORED(NS)

!     LOCAL VARIABLE DECLARATIONS
      INTEGER IGAM
      DOUBLE PRECISION TIN, & !temperature of fluid at heater inlet [C]
      TOUT, &   !temperature of fluid at heater outlet [C]
      TBAR, &   !average temperature of fluid in heater [C]
      TAMB, &   !ambient temperature of heater surroundings [C]
      TSET, &   !heater setpoint temperature [C]
      TON, &    !set temporarily to TOUT before check on TOUT>TSET [C]
      QMAX, &   !heater capacity [kJ/hr]
      QAUX, &   !required heating rate [kJ/hr]
      QLOSS, &  !rate of thermal losses to surroundings [kJ/hr]
      FLOW, &   !fluid flow rate through heater [kg/hr]
      CP, &     !fluid specific heat [kJ/kg.K]
      HTREFF, & !heater efficiency [-]
      UA, &     !overall loss coeff. for heater during operation [kJ/hr.K]
      QFLUID    !rate of energy delivered to fluid [kJ/hr]

!     DATA STATEMENTS
      DATA YCHECK/'TE1','MF1','CF1','TE1','TE1'/
      DATA OCHECK/'TE1','MF1','PW1','PW1','PW1'/

!------------------------------------------------------------------------------
!     GET GLOBAL TRNSYS SIMULATION VARIABLES
      TIME0   = getSimulationStartTime()
      DELT    = getSimulationTimeStep()

!-------------------------------------------------------------------------------
!     SET THE VERSION INFORMATION FOR TRNSYS
      IF(INFO(7).EQ.-2) THEN
        INFO(12)=16
        RETURN 1
      ENDIF

!-------------------------------------------------------------------------------
```

```
!      PERFORM LAST CALL MANIPULATIONS
       IF (INFO(8).EQ.-1) THEN
         RETURN 1
       ENDIF

!-----------------------------------------------------------------------------
!      PERFORM POST CONVERGENCE MANIPULATIONS
       IF(INFO(13).GT.0) THEN
         RETURN 1
       ENDIF

!-----------------------------------------------------------------------------
!    PERFORM INITIALIZATION MANIPULATIONS
       IF (INFO(7).EQ.-1) THEN
         !retrieve unit and type number for this component from the INFO array
         IUNIT=INFO(1)
         ITYPE=INFO(2)
         !reserve space in the global OUT array
         INFO(6) = NO
         ! this TYPE should be called at each iteration whether or not its
         ! inputs change
         INFO(9) = 1
         ! reserve space in the single precision storage structure
         INFO(10) = 0
         ! call the Type check subroutine to compare what this Type requires to
         ! what has been supplied in the input file.
         CALL TYPECK(1,INFO,NI,NP,ND)
         ! call the input-output check subroutine to set the correct input and
         !output variable units.
         CALL RCHECK(INFO,YCHECK,OCHECK)
         ! return to the calling program.
         RETURN 1
      ENDIF

!-----------------------------------------------------------------------------
!      RE-READ THE PARAMETERS IF ANOTHER UNIT OF THIS TYPE HAS BEEN CALLED SINCE
!      THE LAST TIME THEY WERE READ IN
       IF(INFO(1).NE.IUNIT) THEN
         !reset the unit number
         IUNIT  = INFO(1)
         ITYPE  = INFO(2)
         !read the parameter values
         QMAX   = PAR(1)
         CP     = PAR(2)
         UA     = PAR(3)
         HTREFF = PAR(4)
       ENDIF

!-----------------------------------------------------------------------------
!      RETRIEVE THE CURRENT INPUT VALUES FROM THE XIN ARRAY
       TIN  = XIN(1)
       FLOW = XIN(2)
       IGAM = JFIX(XIN(3)+0.1)
       TSET = XIN(4)
       TAMB = XIN(5)
!      CHECK THE INPUTS FOR VALIDITY AND RETURN IF PROBLEMS ARE FOUND
       IF (FLOW.LT.0.)                         CALL TYPECK(-3,INFO,2,0,0)
       IF ((IGAM.GT.1.).OR.( IGAM.LT.0.))      CALL TYPECK(-3,INFO,3,0,0)
       IF (ErrorFound() ) RETURN 1

!      PERFORM CALCULATIONS
       IF (FLOW .GT. 0.) GO TO 10
```

```
!      NO FLOW CONDITION
       OUT(1) = TIN        !set the outlet temperature to the inlet temperature
       OUT(2) = 0.         !set the outlet flow rate to 0.
       OUT(3) = 0.         !set auxiliary energy use to 0.
       OUT(4) = 0.         !set heater losses to 0.
       OUT(5) = 0.         !set the energy imparted to the liquid to 0.
       RETURN 1            !return control to the kernel

!      FLOW CONDITION
!      CHECK INLET TEMPERATURE AND CONTROL FUNCTION
10     IF ((TIN.LT.TSET).AND.(IGAM.EQ.1)) GO TO 20

!      HEATER “OFF” CONDITION
       TOUT   = TIN        !set the outlet temperature to the inlet temperature
       QAUX   = 0.         !set the auxiliary energy use to 0.
       QLOSS  = 0.         !set the heater losses to 0.
       QFLUID = 0.         !set the energy imparted to the liquid to 0.
       GO TO 50            !skip to the output setting section.

!      HEATER “ON” CONDITION
20     TON  =(QMAX*HTREFF+FLOW*CP*TIN+UA*TAMB-UA*TIN/2.d0)/(FLOW*CP+UA/2.d0)
       TOUT = MIN(TSET,TON)
       TBAR = (TIN+TOUT)/2.d0
       QAUX = (FLOW*CP*(TOUT-TIN)+UA*(TBAR-TAMB))/HTREFF
       QLOSS = UA*(TBAR-TAMB) + (1.d0-HTREFF)*QAUX
       QFLUID = FLOW*CP*(TOUT-TIN)

!      SET OUTPUTS
50     OUT(1) = TOUT
       OUT(2) = FLOW
       OUT(3) = QAUX
       OUT(4) = QLOSS
       OUT(5) = QFLUID

       RETURN 1
       END SUBROUTINE Type151
```

8.3.16. Including your new component in an external DLL

This step is covered by section 8.5.2, which explains how to create and configure a new project that will generate a DLL in the UserLib folder.

Note that if you start from the TRNSYS Studio and generate a skeleton for your Type using the "Export as Fortran" command, the Studio will automatically create a Project with the correct settings for you.

8.4. Reference

8.4.1. Global constants

8.4.1.1. Constants settings limits on TRNSYS simulations

Constant	Default	Comment
nMaxUnits	1000	Maximum number of units
nMaxEquations	500	Maximum number of equations
nMaxDerivatives	100	Maximum number of derivatives
nMaxOutputs	3000	Maximum number of Outputs
nMaxParameters	2000	Maximum number of Parameters
nMaxStorageSpots	10000	Number of Storage places in the S array
nMaxFiles	300	Maximum number of files that can be opened by TRNSYS
nMaxDescriptions	750	Maximum number of descriptions in an input file
nMaxVariableUnits	750	Maximum number of input variable types in an input file
nMaxLabels	100	Maximum number of labels in an input file
nMaxFormats	100	Maximum number of format statements in an input file
nMaxChecks	20	Maximum number of chek statements in an input file
nMaxCheckCodes	30	Maximum number of outputs per CHECK statement
nMaxUnitConversions	250	Max. number of unit conversions allowed in units.lab file
nMaxUnitTypes	250	Maximum number of unit types allowed in units.lab file
nMaxCardValues	250	Max. nb. of values allowed in one TRNSYS input file card E.g. a component with 500 parameters would require nMaxCardValues to be greater or equal to 500
minTimeStep	$^{0.1}/_{3600}$	Minimum Simulation Time Step [in hours] (0.1 second)
nMaxTimeSteps	109	Maximum number of time steps in a simulation (1 billion)
nMaxPlottedTimeSteps	525601	Max. nb. of time steps plotted by the online plotter Note: This constant is set in TRNExe. Do not edit
nMaxErrorMessages	1000	Maximum number of standard error messages in TrnsysErrors Note: If you increase this constant, additional messages will not be initialized. Add lines in TrnsysErrors.f90

8.4.1.2. Constants used for string length

Constant	**Default**	**Comment**
maxPathLength	300	Maximum length of variables containing path- and filenames Note: DFWIN defines MAX_PATH as 260 but Windows XP can actually use much longer pathnames. ATTENTION: TRNExe must be adapted if this constant is modified. Users should not change maxPathLength
maxFileWidth	1000	Maximum file width, i.e. maximum length of any line in a text file that must be read from / written to by TRNSYS maxFileWidth should be >= maxPathLength. This constant is also used for strings, e.g. error messages
maxDescripLength	25	Maximum length of a variable description, e.g. descriptions for printers and plotters
maxVarUnitLength	20	Maximum length of units associated with variables
maxEqnNameLength	20	Maximum length of variable names in equations
maxLabelLength	300	Maximum length of labels. Some labels are file- and pathnames so a suggested value is maxLabelLength = maxPathLength
maxMessageLength	800	Maximum length of notices, warnings and error messages Error messages and any text printed on the same line must fit within the maximum file width (maxFileWidth). It is recommended to set maxMessageLength to maxFileWidth-200

8.4.2. Access functions

This section lists all functions declared in the "TrnsysFunctions" module. Note that the subroutines used to handle data storage are mentioned here for the record, but detailed explanations on their use can be found in section 8.4.4.16.

Please note that the first group of access functions (getMaxDescripLength(), etc.) provide access to global constants. Those constants are declared in the TrnsysConstants module, so Fortran-written Types can access them more easily through an "use" statement: use TrnsysConstants. Those access functions are provided for non-Fortran Types.

8.4.2.1. function getMaxDescripLength( )

[maxDescripLength is also available through "`use TrnsysConstants`"]

An integer function that returns the maximum allowable length of a variable description, e.g. input descriptions for printers and plotters. The value of the maximum allowable length of a variable description can be set in the TrnsysConstants file. If the value is modified, the trnlib.dll must then be recompiled in order for the change to take effect.

Example Usage

```
INTEGER DescripLen
...
DescripLen = getMaxDescripLength ()
```

8.4.2.2. function getmaxFileWidth( )

[maxFileWidth is also available through "`use TrnsysConstants`"]

An integer function that returns the maximum allowable width (in characters or columns) of line in any text file that must be read by TRNSYS. This includes both the input file and external data files read by the DynamicData utility routine. The value of the maximum allowable line length can be set in the TrnsysConstants file. If the value is modified, the trnlib.dll must then be recompiled in order for the change to take effect.

Example Usage

```
INTEGER FileWidth
...
FileWidth = getmaxFileWidth()
```

8.4.2.3. function getMaxLabelLength( )

[maxLabelLength is also available through "`use TrnsysConstants`"]

An integer function that returns the maximum allowable length (in characters) of variables that contain pathnames and filenames. The value of the maximum allowable label length can be set in the TrnsysConstants file. If the value is modified, the trnlib.dll must then be recompiled in order for the change to take effect.

Example Usage

```
INTEGER LabLen
...
LabLen = getMaxLabelLength()
```

8.4.2.4. *function getMaxPathLength( )*

[maxPathLength is also available through "`use TrnsysConstants`"]

An integer function that returns the maximum allowable length (in characters) of pathnames. The value of the maximum allowable path length can be set in the TrnsysConstants file. If the value is modified, the trnlib.dll must then be recompiled in order for the change to take effect.

Example Usage

```
INTEGER PathLen
...
PathLen = getMaxPathLength()
```

8.4.2.5. *function getnMaxStorageSpots( )*

[nMaxStorageSpots is also available through "`use TrnsysConstants`"]

An integer function that returns the maximum total number of storage spots that may be requested in a given simulation. Note that this function is used by the setStorageSize subroutine and that therefore it is not necessary for the Type programmer to check whether the required number of storage spots for his/her Type will exceed the maximum allowable number of storage spots. The value of the maximum allowable number of storage spots can be set in the TrnsysConstants file. If the value is modified, the trnlib.dll must then be recompiled in order for the change to take effect.

Example Usage

```
INTEGER StorSpots
...
StorSpots = getnMaxStorageSpots()
```

8.4.2.6. *function CheckStability( )*

This function is only used with Solver 1 (Powell's method). You can find more information on TRNSYS Solvers in Volume 07, TRNEdit (check the section on the Solver Statement).

An integer function that returns a 1 if the last time step converged and a 0 if the last time step did not converge. The CheckStability function is used in SOLVER 1, which tries more than one control strategy and then backs up a time step in order to try something else if it did not find a stable solution. TRNSYS data reading components need to know that they should not continue reading the data file but should back up as well.

Example Usage

```
if (CheckStability() < 1) then
    backspace(LU_DATA)
endif
```

8.4.2.7. *function ErrorFound( )*

Logical function that returns a value of “TRUE” if TRNSYS errors have been found. This function is most useful in Types that call one or more of the TRNSYS Utility subroutines (such as TYPECK, PSYCHROMETRICS, or STEAM) which may call find and flag errors. For example, say a user written Type calls TYPECK to indicate that one of the INPUTS had an inappropriate value. TYPECK would call the TRNSYS utility subroutine MESSAGES, which would print out the error, would log that an error occurred and would return control to TYPECK. TYPECK would in turn return control to the Type, which can then avoid the remainder of its calculations by accessing the ErrorFound function. An example follows in which calculations cease if an illegal value of the input variables FLOW_CHW, FLOW_CW, or FLOW_HW is found.

Example Usage

```
IF(FLOW_CHW.LT.0.) CALL TYPECK(-3,INFO,2,0,0)
IF(FLOW_CW.LT.0.) CALL TYPECK(-3,INFO,4,0,0)
IF(FLOW_HW.LT.0.) CALL TYPECK(-3,INFO,6,0,0)
IF( ErrorFound() ) RETURN 1
```

8.4.2.8. *function getConvergenceTolerance( )*

Double precision function that returns the user specified value of a simulation’s convergence tolerance (the second argument in the TOLERANCES keyword statement).

Example Usage

```
DOUBLE PRECISION ErrTol
...
ErrTol = getConvergenceTolerance()
```

8.4.2.9. *function getDeckFileName( )*

A character function that returns the name (NOT including the path) of the input file being run. This function is useful in printer and other output devices in which results that pertain to a particular input file are to be stored. The length of the returned string is maxPathLength.

Example Usage

```
USE TrnsysConstants, ONLY: maxPathLength

CHARACTER (len=maxPathLength) DeckName
...
DeckName = getDeckFileName()
```

8.4.2.10. *function getFormat(i,j)*

A character function that returns the jth format statement associated with unit i. The length of the returned string is maxFileWidth. For example, in certain modes, Type9 takes a single Format statement.. To access the format of the data file line that is to be read, Type9 contains the following code:

Example Usage

```
use TrnsysConstants, only: maxFileWidth
...
character (len=maxFileWidth) :: myFormat
...
```

```
myFormat = getFormat(INFO(1),1)
...
! Print or read using myFormat
write(luPrint,myFormat) ... variables to be printed
```

8.4.2.11. function getLabel(i,j)

A character function that returns the jth label of unit i. The length of the returned string is maxLabelLength.. For example, Type66 takes two LABELs; the first is the location of the EES executable, the second is the name and location of the EES file that is to be solved. To access these labels, Type66 contains the following code:

Example Usage:

```
USE TrnsysConstants, ONLY: maxLabelLength

INTEGER ThisInstance
CHARACTER (len=maxLabelLength) EESLocation,EESFile(10)
...
EESLocation = getLabel(INFO(1),1)
EESFile(ThisInstance) = getLabel(INFO(1),2)
```

8.4.2.12. function getListingFileLogicalUnit( )

An integer function that returns the logical unit of the TRNSYS list file.

Example Usage

```
INTEGER LUW
...
LUW = getListingFileLogicalUnit()
```

8.4.2.13. function getLUfileName (i)

A character function that returns the name of the file corresponding to the logical unit i. If the file logical unit number has not been assigned, it returns an error message.

Example Usage

```
INTEGER LU
CHARACTER (len=maxLabelLength) fileName
...
fileName = getLUfileName(LU)
```

8.4.2.14. function getMinimumTimestep( )

Double precision function that returns the minimum allowable TRNSYS time step. The minimum allowable TRNSYS time step can be set in the TrnsysConstants file. If the value is modified, the trnlib.dll must then be recompiled in order for the change to take effect.

Example Usage

```
DOUBLE PRECISION minStep
...
minStep = getMinimumTimestep()
```

8.4.2.15. function getNextAvailableLogicalUnit( )

Integer function that returns an available logical unit number and logs the logical unit number in a global list of assigned logical unit numbers. The getNextAvailableLogicalUnit function can be used to find an available logical unit number for temporary files, scratch files, or for data files accessed by components that do not require a logical unit number among their list of parameters.

Example Usage

```
LU = getNextAvailableLogicalUnit()
! Write some data to a temporary file
open(UNIT=LU,FILE='TmpFileForMyType.tmp',status='new')
write(LU,*) ... Data to be written ...
close(LU,status='keep')
```

8.4.2.16. function getnMaxIterations( )

An integer function that returns the maximum allowable number of iterations in one time step before the simulation continues with the current values and a warning is issued. The maximum allowable number of iterations can be set in a Limit statement (or in the Studio's Control Cards).

Example Usage

```
INTEGER maxIter
...
maxIter = getnMaxIterations()
```

8.4.2.17. function getnMaxWarnings( )

An integer function that returns the maximum allowable number of warnings written to the simulation list and log files before the simulation stops in error. The maximum allowable number of warnings can be set in a Limit statement (or in the Studio's Control Cards).

Example Usage

```
INTEGER maxWarns
...
maxWarns = getnMaxWarnings()
```

8.4.2.18. function getnTimeSteps( )

An integer function that returns the total number of time steps in the simulation being run.

Example Usage

```
INTEGER nSteps
...
nSteps = getnTimeSteps()
```

8.4.2.19. function getNumberOfErrors( )

An integer function that returns the total number of errors that have occurred thus far in the simulation.

Example Usage

```
INTEGER NumErrs
...
NumErrs = getNumberOfErrors()
```

8.4.2.20. function getNumericalSolver( )

An integer function that returns the Solver used by TRNSYS. More information on TRNSYS solvers can be found in Volume 07, TRNEdit (check the section on the Solver statement). Handling of discrete control signals is different with Solver 0 and Solver 1. See Type 2 for a n example.

Example Usage

```
integer TRNSolver
...
if (TRNSolver == 0) then
    ... Do things required for successive substitution
else
    ... Do things required for Powell's solver
endif
```

8.4.2.21. function getSimulationStartTime( )

Double precision function that returns the hour of the year at which the simulation began.

Example Usage

```
DOUBLE PRECISION TIME0
...
TIME0 = getSimulationStartTime()
```

8.4.2.22. function getSimulationStartTimeV15( )

Double precision function that returns the hour of the year at which the simulation began. The returned value is the original start time in a Version 15 deck file (Start time is changed by TRNSYS 16 to adapt it to the new start time definition)

Example Usage

```
DOUBLE PRECISION TIME0V15
...
TIME0V15 = getSimulationStartTimeV15()
```

8.4.2.23. function getSimulationStopTime( )

Double precision function that returns the hour of the year at which the simulation is set to end.

Example Usage

```
DOUBLE PRECISION TFINAL
...
TFINAL = getSimulationStopTime()
```

8.4.2.24. *function getSimulationTimeStep( )*

Double precision function that returns the simulation time step (in hours).

Example Usage

```
DOUBLE PRECISION DELT
...
DELT = getSimulationTimeStep()
```

8.4.2.25. *function getTrnsysExeDir( )*

A character function that returns the path in which the TRNSYS executable (TrnExe.exe) is located (default = "C:\Program Files\Trnsys16\Exe"). The length of the returned string is maxPathLength.

Example Usage

```
USE TrnsysConstants, ONLY: maxPathLength

CHARACTER (len= maxPathLength) ExeLoc
...
ExeLoc = getTrnsysExeDir()
```

8.4.2.26. *function getTrnsysInputFileDir( )*

A character function that returns the path in which the input file being run is contained. This function is useful in printer and other output devices in which results that pertain to a particular input file are to be stored. The length of the returned string is maxPathLength.

Example Usage

```
USE TrnsysConstants, ONLY: maxPathLength

CHARACTER (len=maxPathLength) DeckLoc
...
DeckLoc = getTrnsysInputFileDir()
```

8.4.2.27. *function getTrnsysRootDir( )*

A character function that returns the path to the root TRNSYS directory (default = "C:\Program Files\Trnsys16 "). The length of the returned string is maxPathLength.

Example Usage

```
USE TrnsysConstants, ONLY: maxPathLength

CHARACTER (len=maxPathLength) RootDir
…
RootDir = getTrnsysRootDir()
```

8.4.2.28. *function getTrnsysUserLibDir( )*

A character function that returns the path and directory in which all of the loadable user dlls are located (default = "C:\Program Files\Trnsys16\UserLib"). The length of the returned string is maxPathLength.

Example Usage

```
USE TrnsysConstants, ONLY: maxPathLength

CHARACTER (len=maxPathLength) UsrLibDir
...
UsrLibDir = getTrnsysUserLibDir()
```

8.4.2.29. *function getVariableDescription(i,j)*

String function that returns the variable descriptor for the jth variable of Unit i. Variable descriptors are strings that give the "name" of the variable in printers and other output devices. The length of the returned string is maxDescripLength.

Example usage

```
character (len=maxDescripLength) :: columnHeader
...
do j = 1, nVariables
    columnHeader = getVariableDescription(INFO(1),j)
    ... write column header to a file, etc...
enddo
```

8.4.2.30. *function getVariableUnit(i,j)*

String function that returns the variable unit string for the jth variable of Unit i. Variable descriptors are strings that give the "name" of the variable in printers and other output devices. The length of the returned string is maxVarUnitLength.

Example usage

```
character (len=maxVarUnitLength) :: varUnitString
...
do j = 1, nVariables
    varUnitString = getVariableDescription(INFO(1),j)
    ... write units to a file, etc...
enddo
```

8.4.2.31. *function getVersionNumber( )*

Integer function that returns the TRNSYS major version number (i.e, if the version number is XX.Y, it will return XX). This function is primarily useful in allowing for backward compatability between versions of a component. Say, for example that a Type required 14 parameters in an input file written for TRNSYS version 15 but requires 15 parameters in an input file written for TRNSYS version 16. The Type programmer could access the getVersionNumber function and know whether the input file being run was written for TRNSYS version 15 or 16, and could read the corresponding parameter list.

Example Usage

```
INTEGER VERSNUM
...
VERSNUM = getVersionNumber()
```

8.4.2.32. function getMinorVersionNumber( )

Integer function that returns the TRNSYS minor version number (if the version number is XX.Y, it will return Y). This function is primarily useful in allowing for backward compatability between versions of a component, when changes have occurred between minor release versions. For example, a Type required 14 parameters in an input file written for TRNSYS version 16.0 but requires 15 parameters in an input file written for TRNSYS version 16.1. The Type programmer could access the getMinorVersionNumber function and know whether the input file being run was written for TRNSYS version 16.0 or 16.1, and could read the corresponding parameter list.

Example Usage

```
INTEGER MINORVERSNUM
...
MINORVERSNUM = getMinorVersionNumber()
```

8.4.2.33. function LogicalUnitIsOpen(LU)

Logical function that returns true if the Logical unit has been assigned in the deck file and opened by TRNSYS.

Example usage

```
logical :: luOk
integer :: lu
...
lu = nint(par(1))
luOk = LogicalUnitIsOpen(lu)
if (luOk) then
    ... it's OK to print to / read from the file
else
    ... Probably an error in parameters
endif
```

8.4.2.34. function getTimeReport()

Logical function that returns the value of the global variable TIME_REPORT. This global variable is set to false by default, but may be set to true in the input file.

Example usage

```
logical :: timereport
...
timereport = getTimeReport()
```

8.4.2.35. function getConnectedOutputNumber(i,j)

Integer function that returns the output number connected to the jth input of Unit i. When using Solver 1, you should use the similar function getConnectedOutputNumberS1(i,j).

Example usage

```
Integer:: getConnectedOutputNumber
Integer:: i,j

...
getConnectedOutputNumber = getConnectedOutputNumber (i,j)
```

8.4.2.36. *function getConnectedUnitNumber(i,j)*

Integer function that returns the unit connected to j^th input of Unit i. When using Solver 1, you should use the similar function getConnectedUnitNumberS1(i,j).

Example usage

```
Integer:: getConnectedUnitNumber
Integer:: i,j

...
getConnectedOutputNumber = getConnectedUnitNumber (i,j)
```

8.4.2.37. *function getConnectedTypeNumber(i,j)*

Integer function that returns the type number of the unit connected to j^th input of Unit i. When using Solver 1, you should use the similar function getConnectedTypeNumberS1(i,j).

Example usage

```
Integer:: getConnectedTypeNumber
Integer:: i,j

...
getConnectedTypeNumber = getConnectedTypeNumber (i,j)
```

8.4.2.38. *function getConnectedVariableType(i,j)*

Character function that returns the three-letter description of the type of variable that is connected to j^th input of Unit i. This information is stored the array OCHECK. When using Solver 1, you should use the similar function getConnectedVariableTypeS1(i,j).

Example usage

```
Character*3:: getConnectedVariableType
Integer:: i,j

...
getConnectedVariableType = getConnectedVariableType (i,j)
```

8.4.2.39. *function getConnectedVariableUnit(i,j)*

Character function that returns the corresponding units (e.g., kJ/kg-h) of the variable that is connected to j^th input of Unit i. When using Solver 1, you should use the similar function getConnectedVariableUnitS1(i,j).

Example usage

```
Character(len=maxVarUnitLength):: getConnectedVariableUnit
Integer:: i,j

...
getConnectedVariableUnit = getConnectedVariableUnit(i,j)
```

8.4.3. The INFO array – Typical calling sequence

The INFO array is an integer array of 15 values. It conveys to the component subroutines (Types) and to the TRNSYS kernel information about the current UNIT. The contents of the INFO array for a given unit are given in Table 8.4.2.39-1 (see text for detailed explanations).

Table 8.4.2.39-1: INFO array contents

INFO(n)	Role
1	Unit number
2	Type number
3	Number of Inputs in the input file (.dck) for this Unit
4	Number of Parameters in the input file (.dck) for this Unit
5	Number of Derivatives in the input file (.dck) for this Unit
6	Number of outputs (set by the Type)
7	Number of iterative calls to this Unit in the current time step. Special values: -1, -2
8	Total number of calls to this unit. Special values: -1
9	Indicate whether or not the Type depends on the passage of time and tells TRNSYS where the Type should go in the calling sequence
10	Used in storage management for TRNSYS 15 Types (not used by TRNSYS 16 Types)
11	Indicates number of discrete control variables
12	TRNSYS version for which the Type was written (15 or 16)
13	Indicates when all Units have converged in the current time step
14	Reserved for future use
15	Reserved for future use

8.4.3.1. INFO(1:5)

The fist 5 spots in INFO are set by TRNSYS from information in the input file (.dck). They are used to distinguish the current Unit / Type from the other ones and to detect possible configuration errors with the TYPECK routine. It is possible to check that the number of parameters in the input file is acceptable, etc.

8.4.3.2. *INFO(6)*

INFO(6) indicates the number of outputs used by the Type. A minimum of 20 outputs is reserved for each unit in the simulation, but INFO(6) should always be set to provide an accurate Trace and insure efficient error checking.

8.4.3.3. *INFO(7)*

INFO(7) indicates how many times the Unit has been called in the current time step. This information is very useful for building stability into user written controller routines and for eliminating unneeded calculations at each time step. INFO(7) also carries out information through special values (-1 and -2). Possible values of INFO(7) are listed in Table 8.4.3.3-1.

Table 8.4.3.3-1: INFO(7) values

INFO(7)	Role
-2	Special call at the very beginning of the simulation to allow for Type version signing. Types should only set the value of INFO(13) during this call
-1	Initial call in simulation for this Unit. The Type should only perform initialization operations here: array sizing, memory allocation, file opening, parameter checking, etc. The Type should also set the value of INFO(6), INFO(9) and INFO(10) (if used)
0	First call in the current time step for this Unit.
1	First iterative call (second call) in the current time step for this Unit
2	Second iterative call (third call) in the current time step for this Unit
+n	n^{th} iterative call ($(n+1)^{th}$ call) in the current time step for this Unit

8.4.3.4. *INFO(8)*

INFO(8) indicates how many times the Unit has been called in the simulation. INFO(8) also carries out information through special values (-1). Possible values of INFO(8) are listed in Table 8.4.3.4-1.

Table 8.4.3.4-1: INFO(8) values

INFO(8)	Role
1	Very first call in the simulation (version signing call)
2	Initialization call (INFO(7) = -1)
3	Initial conditions at Time = Start time (no iteration for this time step)
4	First call of the first time step (Time = Start time + Time step)
5	Second call of the first time step (Time = Start time + Time step)
6	Third call of the first time step, or first call of next time step
…	Etc
-1	Very last call of the simulation. This call occurs **after the "OK" or "Continue? Yes"** button has been pressed in TRNExe. Types should close files and perform other post-simulation tasks during this call.

8.4.3.5. *INFO(9) and Types calling sequence*

INFO(9) indicates if the Type depends on the passage of time and tells TRNSYS where it should go in the calling sequence. Different options are available: normal (iterative) component, integrator, printer, etc. INFO(9) is initialized to 1 for all Types by the TRNSYS Kernel. Each Type can change that value during the initialization call (when INFO(7) = -1).

Possible values of INFO(9) are listed in Table 8.4.3.5-1. Please note that the introduction of a post-convergence call (flagged with INFO(13) = 1) makes some of the options offered by INFO(9) redundant. They have been kept for backwards compatibility.

Table 8.4.3.5-1: INFO(9) values

INFO(9)	Role
0	The Type's outputs only depend upon its input values and not explicitly upon time. Note that failure to properly identify such a component (keeping the default value of INFO(9) = 1) will only results in some unnecessary calls to the component
1 (default)	The Type's outputs depend upon the passage of time and the Type must therefore be called at least once every time step even if the values of inputs do not change
3	The Type should be called after all other components have converged and before the integrators and printers. User-written statistic subroutines that send their output to integrators is a situation where this may be appropriate.
4	Standard integrators
5	Standard printers
2	The Type should be called after all other components have converged and after the integrators and printers. User-written output subroutines and non-iterative controllers are situations for which this would be appropriate.

Note that **the order in which components are called does not match the logical progression in INFO(9) values for backwards compatibility reasons**. The calling sequence is as follows:

- Iterations during a time step: Units with INFO(9) = 1 (and Units with INFO(9) = 0 if their inputs have changed) are called iteratively.
- Convergence has been reached. All INFO(9) = 1 and 0 are called again with INFO(13) = 1 so they can perform post-convergence manipulations (e.g. store values)
- INFO(9) = 3 components (user-written routines to be called after convergence but before integrators) are called once
- INFO(9) = 4 components (standard integrators) are called once
- INFO(9) = 5 components (standard printers) are called once
- INFO(9) = 2 components (user-written components to be called after integrators and printers) are called once

This calling sequence only applies to normal time steps. All components are called once, in an order according to their Unit numbers, during the following calls:

- Version signing call (INFO(7) = -2)

- Initialization call (INFO(7) = -1)
- Initial time step (INFO(7) = 0, TIME = Start time). No iterations are performed, Types should just output initial conditions. and the very
- Very last call (INFO(8) = -1)

8.4.3.6. INFO(10)

INFO(10) was used for storage management in TRNSYS 15. It was used to send the number of required storage spots to TYPECK and then receive the index of the first reserved spot in the global storage array. That way of handling storage is obsolete in TRNSYS 16 and should not be used anymore. It is still present to allow TRNSYS 15 Types to run in legacy mode. Please note that the storage array accessed by those Types is single-precision, unlike the storage array manipulated through the new access functions (SetStorageSize, SetStorageVars, GetStorageVars). The standard way of handling variable storage in TRNSYS 16 is described in section 8.4.4.16.

8.4.3.7. INFO(11)

INFO(11) sets the number of discrete control variables in the Type. It should only be set to a non-zero value if Solver 1 must be used with the Type. It is very important to make sure that the Solver used in a simulation is handled properly by all Types. For more details about Solver 1, please refer to the Solver Statement description in Volume 07, TRNEdit manual.

8.4.3.8. INFO(12)

INFO(12) indicates the TRNSYS version for which the Type was written (15 or 16). It must be set at the very first call during the simulation (INFO(7) = -2).

8.4.3.9. INFO(13)

INFO(13) is a flag that indicates that all the Units have converged during the current time step. It is 0 during iterative calls and 1 during the post-convergence call. Special units for which the Type has set INFO(9) to any value greater than or equal to 2 will only be called once per time step, with INFO(13) = 1.

8.4.3.10. Typical calling sequence for an iterative Type

Table 8.4.3.10-1 shows the time and INFO array values for an iterative Type in a typical simulation. Please note the following:

- INFO(1:5) are set by TRNSYS before calling the Type from information in the input (.dck) file and should not be modified by the Type. For that reason they are omitted in the table.
- It is assumed that convergence is reached after 2 iterations (after the 3rd call in a time steps) for all time steps.
- Start, Stop and Step are respectively the simulation start time, stop time and time step. nO is the number of outputs set by the Type.
- S is the value set by the Type for INFO(9).
- Values that change are in **bold** and especially meaningful values have a dark **background**.

- It is assumed that the total number of iterative calls to the Type in the simulation (which is reported in the "Iterative call breakdown" in the listing and log files) is nCalls.
- The Type is assumed to follow TRNSYS 16 Standard (i.e. it is not signed as a "15" Type and it does not use the old storage array)

Table 8.4.3.10-1: Typical INFO calling sequence for an iterative Type

Call description	**Time**	**INFO element**							
		6	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Version signing call	**Start**	**0**	**-2**	**1**	**1**	**-10**	**0**	**0**	**0**
Initialization call	Start	0	**-1**	**2**	1	-10	0	**16**	0
Start (initial conditions)	Start	**nO**	**0**	**3**	**S**	**0**	0	16	0
1st step, 1st call	**Start+Step**	nO	**0**	**4**	S	0	0	16	0
1st step, 2nd call (1st iter.)	Start+Step	nO	**1**	**5**	S	0	0	16	0
1st step, 3rd call (2nd iter.)	Start+Step	nO	**2**	**6**	S	0	0	16	0
1st step, post-convergence	Start+Step	nO	**3**	**7**	S	0	0	16	**1**
2nd step, 1st call	**Start+2*Step**	nO	**0**	**8**	S	0	0	16	**0**
2nd step, 2nd call (1st iter.)	Start+2*Step	nO	**1**	**9**	S	0	0	16	0
2nd step, 3rd call (2nd iter.)	Start+2*Step	nO	**2**	**10**	S	0	0	16	0
2nd step, post-convergence	Start+2*Step	nO	**3**	**11**	S	0	0	16	**1**
etc.	…	…	**…**	**…**	S	0	0	16	…
Last step, 1st call	**Stop**	nO	**0**	**nCalls-3**	S	0	0	16	**0**
Last step, 2nd call (1st iter.)	Stop	nO	**1**	**nCalls-2**	S	0	0	16	0
Last step, 3rd call (2nd iter.)	Stop	nO	**2**	**nCalls-1**	S	0	0	16	0
Last step, post-converg.	Stop	nO	**3**	**nCalls**	S	0	0	16	**1**
Very last call (after "OK")	Stop	nO	**0**	**-1**	S	0	0	16	1

8.4.4. Utility subroutines

8.4.4.1. Calling External Programs

In order to allow calls to external programs, a utility subroutine named "CALLPROGRAM" has been added to TRNSYS with the release of TRNSYS 15. CALLPROGRAM uses several Win32 Application Programming Interface (API) commands to start the external program. API commands directly control the Windows operating system. Depending on the mode specified in the call statement, CALLPROGRAM will either wait for the second program to finish its task before proceeding or will merely start the second program and then proceed with other TRNSYS calculations. The call statement for CALLPROGRAM is:

```
...
CALL CALLPROGRAM(CMDLINE,bwait,prochand, thrdhand)
...
```

Where:

CMDLINE	is the command line text string containing the path and name for the executable program
BWAIT	is a logical variable that determines whether TRNSYS will wait for the second program to finish running or not.
	-If BWAIT = .TRUE., then TRNSYS does not proceed until the second program is finished and closed.
	-If BWAIT = .FALSE., then TRNSYS proceeds once the second program is done initializing but still running.
PROCHAND	is the Windows process handle for the new program. This is a value returned by the CALLPROGRAM routine.
THRDHAND	is the Windows thread handle for the new program. This is a value returned by the CALLPROGRAM routine.

When CALLPROGRAM is used to start another program and leave it running (BWAIT = .FALSE.), then the program needs to be terminated at the end of the simulation before leaving TRNSYS. This can be done by using the following commands:

```
...
res = TerminateProcess(PROCHAND,1)
IF (res.EQ.0) THEN
  WRITE(CHCKWHY,*) GETLASTERROR()
ENDIF
...
```

See the source code of Type 66 (CALL EES MODELS) for an example of the use of CALLPROGRAM.

8.4.4.2. DFIT (FIT)

GENERAL DESCRIPTION

Subroutines FIT and DFIT use linear least-squares regression to determine coefficients of a user defined correlation. FIT requires single precision arguments, while DFIT uses double precision arguments. Since a majority of the kernel routine and standard components use double precision variables throughout, it is recommended that user written routines employ the double precison version DFIT. A call to DFIT is of the form:

```
INTEGER :: NROW,NCOL,NCOEF,NDATA,IFLAG
DOUBLE PRECISION :: X,Y,PHI
...
CALL FIT(NROW,NCOL,NCOEF,NDATA,X,Y,PHI,IFLAG,*N)
```

where,

NROW	(integer) Row dimension of X in routine calling DFIT.
NCOL	(integer) Column dimension of X in routine calling DFIT.
NCOEF	(integer) Number of coefficients in model.
NDATA	(integer) Number of data points for the regression.
X	Double precision two dimensional array containing the values of the independent variables to the model. X(I,J) is the value of the independent variable associated with the ith coefficient of the model and the jth data point.
Y	(double precision) vector containing values of the dependent variable corresponding to independent variable data points.
PHI	(double precision) vector containing the coefficients determined from the regression.
IFLAG	Integer flag returned indicating the error condition (0 - no error, 1 - dimension error, 2 - coefficients cannot be determined).
N	the corresponding line number to jump to if the subprogram is present (see Section 3.3.5)

Least-squares regression is used to minimize a function J, where:

$$J = \sum_{j=1}^{N_{data}} \left(Y_j - Y_{model,j}\right)^2 \qquad \text{Eq. 8.4.4-1}$$

$$Y_{model,j} = \sum_{i=1}^{N_{coef}} \phi_i X_{i,j} \qquad \text{Eq. 8.4.4-2}$$

where,

Yj	jth data point for dependent variable
Ymodel,j	dependent variable of the user's model for the jth data point
ϕi	ith coefficient to be determined with FIT

Xi,j	user defined functions of the independent variable or variables.

Example Consider the equation used in the Type53 Chiller component. The dimensionless power consumption is correlated with performance data using the following bi-quadratic equation:

$$G = a_0 + a_1 E + a_2 E^2 + a_3 F + a_4 F^2 + a_5 EF \qquad \text{Eq. 8.4.4-3}$$

This is the same form as Eq. 8.4.4-2 with:

ϕ_1-ϕ_5 = a_0-a_5

$X_{1,j}$ = 1.0

$X_{2,j}$ = E

$X_{3,j}$ = E^2

$X_{4,j}$ = F

$X_{5,j}$ = F^2

$X_{6,j}$ = EF

If the following data existed for E, F, and G:

E	F	G
3.0	0.2	10.0
5.0	0.4	12.5
7.0	0.6	13.1

Then DFIT would be called with

NROW =	6 or more (minimum is the # of coefficients,i's)
NCOL =	3 or more (minimum is the # of data points,j's)
NCOEF=	6 (number of coefficients, i's)
NDATA =	3 (number of data points, j's)

$$X = \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ E_1 & E_2 & E_3 \\ E_1{}^2 & E_2{}^2 & E_3{}^2 \\ F_1 & F_2 & F_3 \\ F_1{}^2 & F_2{}^2 & F_3{}^2 \\ EF_1 & EF_2 & EF_3 \end{bmatrix} = \begin{bmatrix} 1.0 & 1.0 & 1.0 \\ 3.0 & 5.0 & 7.0 \\ 9.0 & 25.0 & 49.0 \\ 0.2 & 0.4 & 0.6 \\ .04 & .16 & .36 \\ 0.6 & 2.0 & 4.2 \end{bmatrix} \qquad \text{Eq. 8.4.4-4}$$

$$Y = \begin{bmatrix} 10.0 \\ 12.5 \\ 13.1 \end{bmatrix} \qquad \text{Eq. 8.4.4-5}$$

The DFIT subroutine would be called with the above input data and return values for PHI and IFLAG. PHI would be a vector containing the values for a_0-a_5.

8.4.4.3. DIFFERENTIAL_EQN (DIFFEQ)

The double precision utility routine Differential_Eqn and its single precision companion DIFFEQ provide analytical solutions to first-order linear differential equations that can be written as

$$\frac{dT}{dt} = aT + b \qquad \text{Eq. 8.4.4-6}$$

The designation "double precision" means that the arguments and results of the utility routine Differential_Eqn must be declared in the calling Type as DOUBLE PRECISION variables as opposed to REAL or INTEGER variables. In TRNSYS version 15.x and below, only a single precision version of this utility was available. That single precision version (called DIFFEQ) was maintained for completeness and backward compatibility but it is highly recommended that user employ the double precision version (called Differential_Eqn) whenever possible.

Users intending to utilize the Powell's method TRNSYS solver (SOLVER 1) should use the Differential_Eqn subroutine discussed here instead of the optional DTDT array discussed in section 8.4.5. The form of the call to Differential_Eqn is:

```
CALL DIFFERENTIAL_EQN(TIME, AA, BB, TI, TF, TBAR)
```

where

TIME	(double precision) current value of time
AA	(double precision) is the a coefficient of the differential equation
BB	(double precision) is the b coefficient of the differential equation
TI	(double precision) is the value of the dependent variable (T) at the beginning time step
TF	(double precision) is the value of the dependent variable (T) at the end of the time step
TBAR	(double precision) is the average value of the dependent variable over the time step.

The Differential_Eqn routine does not require link checking since it is considered to be a kernel routine; those routines which must be present to successfully run TRNSYS.

Differential_Eqn solves differential equations at each call based on parameters and the current set of inputs. If the inputs have not converged, the iteration proceeds by direct substitution provided that the Successive Substitution solver (SOLVER 0) is being used. Additional notes on the analytical solution of differential equations are as follows:

1) Differential_Eqn requires the arguments AA, BB, and TI, while returning TF and TBAR. For calls to Differential_Eqn at the start time of the simulation (TIME= TIME0), the initial condition TI is returned for both TF and TBAR.

2) It is necessary to save the final values of dependent variables each time step as initial conditions for the following interval. This can be done using calls to the getStorageVars and setStorageVars routine discussed in sections 8.4.4.16.

3) Inputs will often be a function of the dependent variable of the differential equation. For instance, collector outlet temperature, which might be an input to a storage component, might also be a function of the storage temperature. In order to get a good estimate of the average inputs over each interval, TBAR should be set as an output if required as an input of other components. To be consistent with this formulation and the rectangular integration provided by Type24, etc., any outputs that are rates and are expressed as a function of the dependent variable T should be evaluated at TBAR.

As an example of a model formulation, suppose that one wishes to study the temperature response of a one-node house to ambient conditions. The instantaneous energy balance for this situation is given as

$$C\frac{dT_R}{dt} = -(UA)_L(T_R - T_a) \qquad \text{Eq. 8.4.4-7}$$

This can be written in the form of Eq. 8.4.4-6 if

$$a = \frac{-(UA)_L}{C} \quad \text{and} \quad b = \frac{(UA)_L T_a}{C} \qquad \text{Eq. 8.4.4-8}$$

The following code would establish initial conditions, solve the differential equation for the final and average temperatures, and determine the average energy loss rate from the house for each time interval.

```
DOUBLE PRECISION :: STORED(2),TI,AA,BB,UAL,TA,C,QLOSS,TF,TBAR
INTEGER :: NS,INFO(15)
...
!this section is executed at the end of each timestep. In this case it is
!used to update the storage structures for the following time step.
IF(INFO(7).EQ.13) THEN
  CALL getStorageVars(NS,STORED,INFO)
  STORED(1) = STORED(2)
  CALL setStorageVars(NS,STORED,INFO)
 ENDIF
...
CALL getStorageVars(NS,STORED,INFO)
TI = STORED(1)
AA = -UAL/C
BB = UAL*TA/C
CALL DIFFERENTIAL_EQN(TIME,AA,BB,TI,TF,TBAR)
STORED(2) = TF
QLOSS = UAL*(TBAR - TA)
CALL setStorageVars(NS,STORED,INFO)
...
```

For this particular example, the component would not be called by the kernel during any time step after its inputs have converged within the algebraic error tolerance. There is no need for a differential equation convergence check, since the solution will always be consistent with the current set of inputs. A DERIVATIVES control statement should not be used for the component in the simulation input file; the initial value of the dependent variable at the beginning of the simulation should be supplied to the subroutines as a PARAMETER. In this respect the example is similar to Types 12, 21 and 23.

8.4.4.4. DynamicData (DYNDATA and DATA)

The DynamicData utility routine is available to read user supplied data from external text based files that have been assigned a FORTRAN logical unit number in the TRNSYS input file. The DynamicData routine is able to interpolate the data found in the file in up to four dimensions during the course of the simulation. For each data set, up to five dependent (Y) functions may be specified in terms of up to four independent (X) variables. At each call to the DynamicData routine, the calling Type sends the values of each independent variable. DynamicData performs the multi dimensional interpolation and returns the interpolated values of as many dependent variables as have been provided in the data file and requested by the calling Type. DynamicData is NOT able to extrapolate beyond the range of data given in the external file. If the calling Type sends a value of an independent variable that falls above or below the range given in the data file, DynamicData will return the values of the dependent variables that correspond to the maximum or minimum of the range.

DynamicData is a double precision routine, meaning that the independent and dependent variables found in the external file will be treated as FORTRAN double precision variables. Similarly, the X and Y arrays in the calling Type must be declared as double precision. In TRNSYS versions 15.x and below, DynamicData was known as either “DATA” or “DYNDATA,” both of which were single precision routines. DATA and DYNDATA were retained for backwards compatibility but every effort should be made to use the double precision DynamicData routine.

Examples of standard components that use DynamicData are the Type1 Solar Collector and the Type44 Conditioning Equipment model. The form of a call to DynamicData is:

```
INTEGER :: LU,NIND,NX(:),NY(:),INFO(15)
DOUBLE PRECISION :: X(:),Y(:)
...
CALL DynamicData (LU, NIND, NX, NY, X, Y, INFO,*N)
```

where

LU	(integer) is the FORTRAN logical unit number through which the performance data is accessed.
NIND	(integer) is the number of independent variables ($1 \le$ NIND ≤ 4)
NX	is an integer array dimensioned to NIND containing the number of values of each independent variable to be read from LU. NX(1), NX(2), NX(3), and NX(4) are the number of values of X_1, X_2, X_3 and X_4, respectively.
NY	(integer) is the number of dependent (Y) values associated with each set of independent (X) values ($1 \le$ NY ≤ 5)
X	is a double precision array dimensioned to NIND containing the values of the independent variables for which interpolated Y values are desired
Y	is a double precision array of dimension NY containing interpolated values of the dependent variables
INFO	is the calling TYPE's INFO array
N	(integer) is the corresponding line number to jump to if the subprogram is present.

The data is read from logical unit LU at the start of the simulation. Thereafter, DynamicData linearly interpolates for Y values given X values. If DynamicData is called with an X out of range of the supplied data, then the closest specified value is used. DynamicData does not extrapolate beyond the user supplied data.

The data supplied on the logical unit is read in free format. If NIND equals 4 and NX(4) is greater than 1, then NX(4) values of the fourth independent variable X_4 are read first. Similarly, if NIND $\geq$ 3 and NX(3) > 1, then a set of NX(3) values of the third independent variable, X_3 are read. Next, if NIND $\geq$ 2 and NX(2) > 1, then a set of NX(2) values of the second independent variable X_2 are read. Lastly, NX(1) values of the primary independent variable, X_1, are read. The values of the independent variables must be in ascending order, but need not be at regular intervals. The Y values are read beginning with values corresponding to the smallest values of X_1, X_2, X_3, and X_4. NX(1) sets of NY values of Y are read for each value of X_2. NX(2) sets of NX(1) * NY values of Y are read for each value of X_3. In all, if 4 independent variables are employed, then NX(4) * NX(3) * NX(2) * NX(1) * NY values of Y must be specified.

Example: A user wishes to write a model for a water-to-water heat pump whose performance depends upon the inlet flow stream temperatures to both the evaporator (T_{evap}) and condenser (T_{cond}). DynamicData is to be used to evaluate both the capacity and COP. The experimental data is shown in Table 8.4.4.4-1.

Table 8.4.4.4-1

T_{cond} = 20°C			T_{cond} = 50°C		
T_{evap} (°C)	Capacity (kJ/hr)	COP	T_{evap} (°C)	Capacity (kJ/hr)	COP
10	35000	2.47	10	21000	1.73
20	41000	2.80	20	25000	1.96
30	49500	2.99	30	29000	2.09

The data is to be read from logical unit 10. Three values of T_e and two values of T_c will be supplied. The call to DynamicData within the calling Type routine might appear as

```
      USE TrnsysFunctions
      ...
      INGEGER :: NX(2),INFO(15),DUM
      DOUBLE PRECIsion :: X(2),Y(2),CAP,COP,T_EVAP,T_COND
      ...
      NX(1) = 3     !There are 3 values of the second variable in the file.
      NX(2) = 2     !There are 2 values of the first variable in the file.
      X(1)    = T_Evap    !sets x(1) to the local variable t_evap
      X(2)    = T_Cond    !sets x(2) to the local variable t_cond

      CALL DynamicData (10, 2, NX, 2, X, Y, INFO,*101)
      CALL LINKCK('TYPE74', 'DynamicData',3,DUM)
101   IF (ErrorFound()) RETURN 1

      CAP    = Y(1)!sets the call result y(1) to local variable CAP.
      COP    = Y(2)! sets the call result y(1) to local variable COP.
      ...
```

The syntax of the data file accessed through logical unit 10 would be as follows:

20.0 50.0	!T_{cond} values
10.0 20.0 30.0	!T_{evap} values

35000. 2.47 !Capacity and COP for T_{cond}=20°C, and T_{evap}=10°C

41000. 2.80 !Capacity and COP for T_{cond} =20°C, and T_{evap}=20°C

49500. 2.99 !Capacity and COP for T_{cond} =20°C, and T_{evap}=30°C

21000. 1.73 !Capacity and COP for T_{cond} =50°C, and T_{evap}=10°C

25000. 1.96 !Capacity and COP for T_{cond} =50°C, and T_{evap}=20°C

29000. 2.09 !Capacity and COP for T_{cond} =50°C, and T_{evap}=30°C

8.4.4.5. *ENCLOSURE (ENCL)*

General Description

The utility routine ENCLOSURE (and its single precision companion ENCL) calculate view factors between all surfaces of a rectangular parallelpiped. Up to 9 windows or doors may be located on any of the 6 wall surfaces. ENCLOSURE is a double precision routine, meaning that its arguments must be declared in the calling Type as double precision variables (as opposed to REAL variables). The values returned by ENCLUSRE are also double precision. In TRNSYS versions 15.x and below, ENCLOSURE was known simply as “ENCL,” which was a single precision routine. ENCL was retained for backwards compatibility and every effort should be made to use the double precision ENCLOSURE routine.

This routine is used by the TYPE 19 zone model. A call to ENCLOSURE is of the form

```
DOUBLE PRECISION :: SPAR(10),WPAR(10),FV
INTEGER :: INFO(15)
...
CALL ENCLOSURE (SPAR, WPAR, FV, INFO,*N)
```

where

SPAR	a double precision array of 10 values containing the 10 zone geometry parameters described in the TYPE 19 documentation (parameters 2-11 of geometry Mode 1).
WPAR	a double precision array dimensioned to 54, containing the complete list of window or door geometry parameters described in the TYPE 19 section for geometry Mode 1.
FV	a double precision array containing the view factors for all surfaces as calculated by the subroutine.
INFO	the calling TYPE's INFO array.
N	is the corresponding line number to jump to if the subprogram is present.

Subroutine ENCLOSURE outputs a table of the view factors between surfaces. The surface numbers are as specified in the SPAR array.

Mathematical Description

Subroutine ENCL utilizes function subroutine VIEW to obtain view factors between continuous rectangular surfaces. Reciprocity is also used to reduce the number of computations. In order to determine view factors between surfaces that contain windows or doors, it is necessary to perform some view factor algebra.

Consider the general case of m windows located on a surface i and n windows on a surface j as shown in Figure 8.4.4.5-1.

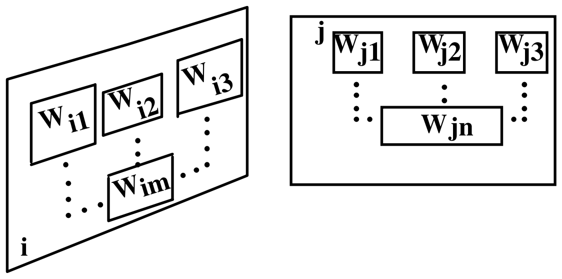


Figure 8.4.4.5-1

In general, the view factor between i and j is

$$F_{i-j} = F_i - \left(j + W_{ji} + ... + W_{jn}\right) - \sum_{k=1}^{n} F_i - (W_{jk}) \qquad \text{Eq. 8.4.4-9}$$

where

$$F_{i-\left(j+W_{j1}+...+W_{jn}\right)} = \frac{\left(A_j + A_{W_{j1}} + ... + A_{W_{jn}}\right)F_{(j+W_{j1}+...+W_{jn})-i}}{A_i} \qquad \text{Eq. 8.4.4-10}$$

$$F_{(j+W_{j1}+...+W_{jn})-i} = F_{(j+W_{j1}+...+W_{jn})-(i+W_{i1}+...+W_{im})} - \sum_{k=1}^{m} F_{(j+W_{j1}+...+W_{jn})-W_{ik}} \qquad \text{Eq. 8.4.4-11}$$

$$F_{i-W_{jk}} = \frac{A_{W_{jk}} F_{(W_{jk})-i}}{A_i} \qquad \text{Eq. 8.4.4-12}$$

$$F_{W_{jk}-i} = F_{W_{jk}-(i+W_{i1}+...+W_{im})} - \sum_{l=1}^{m} F_{W_{jk}-W_{il}} \qquad \text{Eq. 8.4.4-13}$$

The variables W_{i1} through W_{im} are the surface numbers associated with windows on wall i. Likewise, W_{j1} through W_{in} refer to windows on wall j. Thus, (i+W_{i1}+...W_{in}) and (j+W_{j1}+... W_{jm}), each denote surfaces that are collections of individual surfaces. The variable A refers to the area of a surface whose number is used as a subscription.

8.4.4.6. FLUID_PROPS (FLUIDS)

The double precision utility routine called Fluid_Props and its single precision companion FLUIDS are used to calculate the thermodynamic properties of various refrigerants based on correlations. Arguments to the Fluid_Props routine should be declared in the calling Type as DOUBLE PRECISION variables. Arguments to the FLUIDS routine should be declared in the calling Type as REAL. With TRNSYS version 15.x and below, only the single precision version was available. Since most kernel and standard Type variables were made DOUBLE PRECISION with the release of TRNSYS version 16.0, every effort should be made to employ the double precision version of this routine (Fluid_Props). Given two state properties, the Fluid_Props routine will return the complete state of the refrigerant. A call to FLUIDS is of the form

```
INTEGER :: NREF,ITYPE,IFLAG
DOUBLE PRECISION :: PROP(9)
CHARACTER(len=2) :: UNITS
...
CALL FLUIDS (UNITS,PROP,NREF,ITYPE,IFLAGR,*N)
```

where

UNITS	(character) two character variable denoting unit system to be employed by the fluids routine; 'SI' for metric units and 'EN' for english units
PROP	double precision array holding the two known thermodynamic properties of the refrigerant in question upon entry and the complete state of the refrigerant upon return from the Fluid_Props routine. This array must be dimensioned to 9 in the routine that calls Fluid_Props. Each location in this array is described below:
PROP(1)	temperature of refrigerant (F, C)
PROP(2)	pressure of refrigerant (psi, MPa)
PROP(3)	enthalpy of refrigerant (Btu/lbm, kJ/kg)
PROP(4)	entropy of refrigerant (Btu/lbm R, kJ/kg K)
PROP(5)	quality (0 < x < 1)
PROP(6)	specific volume (ft3/lbm, m3/kg)
PROP(7)	internal energy (Btu/lbm, kJ/kg)
PROP(8)	dynamic viscosity (lbm/ft hr, N s/m2)
PROP(9)	thermal conductivity (Btu/hr ft R, kJ/hr m K)
NREF	an integer variable denoting the refrigerant to be analyzed.

Available refrigerants are listed along with the required NREF code number:

R-11	(11)
R-12	(12)
R-13	(13)
R-14	(14)
R-22	(22)
R-114	(114)

	R-134A	(134)
	R-500	(500)
	R-502	(502)
	AMMONIA	(717)

ITYPE	integer variable denoting which two properties are supplied as known properties. ITYPE = 10* PROP indicator 1 + PROP indicator 2 (12 to 98)
IFLAGR	an integer variable returning the error messages from the FLUIDS call: 0 - no error found, calculations completed 1 - error found, calculations could not be completed
N	the corresponding line number to jump to if the subprogram is present.

An example illustrating the use of the FLUID_PROPS routine is shown below. In the example, two inputs (temperature and quality) are read and are placed in the PROP array. The Fluid_Props routine is called. If the Fluid_Props routine is not found for some reason, the error handling routine called LINKCK is called (refer to section 8.4.4.7 for more information). If Fluid_Props returns, the example checks to make sure that no errors were reported by Fluid_Props by calling an Access Function called ErrorFound( ) (refer to section 8.4.2.6 for more information). If errors were detected, execution of the Type that called Fluid_Props stops immediately by returning control to the calling program. If errors were not found then the PROP array has been filled. The Second and third array spots are set to local variables and Type execution continues normally.

```
      USE TrnsysFunctions
      ...
      CHARACTER(len=2) :: UNITS
      DOUBLE PRECISION :: PROP(9),XIN(2),TEMP,QUALITY,PRESSURE,ENTHALPY
      INTEGER :: NREF
      ...
      UNITS = ‘SI’  !SI units are desired.
      NREF = 12     !Refrigerant R12
      ...
      TEMP = XIN(1)
      QUALITY = XIN(2)
      ...
      PROP(1) = TEMP
      PROP(5) = QUALITY
      CALL FLUID_PROPS(UNITS,PROP,12,15,IFLAG,*100)
      CALL LINKCK('TYPE58','FLUID_PROPS',1,58)
100   IF (ErrorFound()) RETURN 1
      ...
      PRESSURE=PROP(2)
      ENTHALPY=PROP(3)
```

MATHEMATICAL DESCRIPTION

The correlations required to solve for the refrigerant properties are from the Engineering Equation Solver program (F-chart software, 1994), a numerical solver employing thermodynamic correlations from many different sources. Interested users should contact the reference for more details on the correlations.

Enthalpies for ammonia and R134a are based on the reference state: enthalpy equal to zero at -40°C.

Enthalpies for all other refrigerants are based on the reference state: enthalpy equal to zero at 0°C

This subroutine checks for many improper inputs, such as qualities less than 0. or greater than 1., and the input of two properties that cannot be correct for one state. For these and other improper inputs, the subroutine either prints a warning, corrects one of the inputs, and continues; or prints an error and halts the simulation.

Subcooled properties are not available for refrigerants. If a subcooled state is specified, the saturated liquid results will instead be provided.

8.4.4.7. LINKCK

The LINKCK subroutine is a utility subroutine used for the detection and subsequent error message printing of unlinked subroutines. With earlier version of TRNSYS (up to version 15.0) the LINKCK routine was of vital importance because as a memory saving step TRNSYS was often compiled to include only those kernel routines that were critical to a given simulation. With advances in computing power and speed, this step became less and less important. LINKCK remains as a safeguard against inadvertently unlinked routines but almost always, unlinked routines will be caught during the compiling and linking processes. A call to LINKCK is of the form:

```
CHARACTER(len=12) :: ENAME1,ENAME2
INTEGER :: ILINK,LNKTYP
...
CALL LINKCK(ENAME1,ENAME2,ILINK,LNKTYP,*N)
```

where

ENAME1	a 12-character variable identifying the subroutine from which the call originated.
ENAME2	a 12-character variable identifying the subroutine that was called
ILINK	an integer indicating the steps to be taken by the LINKCK program
	= 1 an unlinked subroutine has been found, generate an error message and stop the program
	= 2 an unlinked subroutine has been found, generate a warning but keep running
	= 3 an unlinked TYPE subroutine has been found, generate an error message and stop the program
	= 4 warn the user that a subroutine requires use of an external function which can not be link checked
LNKTYP	integer variable corresponding to the TYPE subroutine which was not found in the TRNSYS executable

The LINKCK subroutine is provided to the users as a means to standardize the TRNSYS error and warning messages associated with unlinked subroutines. The user should only call LINKCK when an unlinked subroutine is detected or an external function is required.

The following example should clarify the use of the LINKCK subroutine. Refer to section 3.3.5 for more details.

```
SUBROUTINE TYPE75(TIME,XIN,OUT,T,DTDT,PAR,INFO,ICNTRL,*)
...
CHARACTER(len=12) ENAME1,ENAME2
DOUBLE PRECISION :: SPAR(10),WPAR(10),FV
 INTEGER :: ILINK,LNKTYP,INFO(15)
...
!CALL THE ENCLOSURE SUBROUTINE AND INFORM LINKCK IF NOT PRESENT
CALL ENCLOSURE(SPAR,WPAR,FV,INFO,*101)
!STOP THE PROGRAM IF ENCL IS NOT PRESENT
ILINK = 1
IDUM = 75
ENAME1 = 'TYPE75      '
ENAME2 = 'ENCLOSURE   '
```

```
      CALL LINKCK(ENAME1,ENAME2,ILINK,IDUM)

      !ENCL IS PRESENT - CONTINUE ON
101   CONTINUE
      ...
```

8.4.4.8. MATRIX_INVERT (INVERT and DINVRT)

Subroutine Matrix_Invert is a double precision routine available to invert a matrix of up to 50 by 50 entries. In TRNSYS version 15.x and below, the subroutine had two forms; INVERT was a single precision version and DINVRT was a double precision. With the release of TRNSYS 16.0, the double precison routine was renamed and is recommended for use by all standard and user written Types. INVERT and DINVRT were maintained for backwards compatibility. A call to Matrix_Invert is of the form

```
INTEGER :: NRC,N,IFLAG
DOUBLE PRECISION :: A
...
CALL MATRIX_INVERT (NRC, N, A, IFLAG,*N)
```

where

NRC	(integer) is the column and row dimensions of the two-dimensional A array as defined in the program that calls Matrix_Invert.
N	(integer) the number of equations and unknowns associated with the problem.
A	a double precision two-dimensional array (A(NRC, NRC)). Upon calling Matrix_Invert A should contain the matrix to be inverted. Matrix_Invert also returns the inverted matrix in the A array.
IFLAG	an integer flag that is set to 0 if the inversion proceeds correctly, 1 if the number equations and unknowns, N, exceeds 50, and 2 if the matrix is singular.
N	the corresponding line number to jump to if the subprogram is present.

The inverted matrix is returned in the A array. The method used to invert the matrix is the Gauss-Jordan reduction with maximum pivoting.

8.4.4.9. MESSAGES

The Messages utility routine provides the Type programmer with a convenient method of reporting error messages to the end user. When a Type catches a particular condition, it can send a text string and information as to the severity of the condition to the Messages utility subroutine. Messages will take care of logging the event, reporting it to both the simulation list and log files, and stopping the simulation if appropriate.

INTERFACE

```
subroutine Messages(errorCode,message,severity,unitNo,typeNo)
    integer :: errorCode
    character (len=*) :: message, severity
    integer :: unitNo, typeNo
```

(Note: Although the length of message is not explicitly declared, it should never exceed the maxMessageLength global constant).

USAGE

ERRORCODE

errorCode is a standard TRNSYS error message number. Please refer to the initializeErrorMessages() subroutine in Messages.f90 for existing messages.

If you wish to generate a custom error message (as will most often be the case for user-written Types), simply set ErrorCode to a value of –1, indicating the messages that the second argument (message) contains the information to be printed.

MESSAGE

If you are generating a custom error message, this string variable will be printed in a standard format by the Messages subroutine. It should consist of one line of text, shorter than maxMessageLength. The example here below illustrates the use of Write to add run-time information to the message.

If you are using a standard error message, the string text will be printed in addition to the standard error message under the heading “Reported Information.” It is a method for providing additional information about the error.

SEVERITY

severity is a string that indicates the severity of the message. Messages understands 4 levels of severity:

- "**Notice**": A notice is simply information that you would like the user to know
- "**Warning**": The messages subroutine keeps track of how many "Warning" messages have been generated. If the maximum allowable number of warnings (set by the LIMITS statement in the input file) are exceeded, then the messages subroutine automatically generates a "Fatal" error.
- "**Fatal**": Designates a fatal error, i.e. an error that should stop the simulation. A call to Messages with such an error should be followed by a "return 1" statement in the calling Type. Note that TRNSYS will stop after going up the chain of "return" statements and giving

the chance to all subroutines to perform end-of-simulation operations. Further errors might be generated during that process.

- "**Stop**": The error will abort TRNSYS immediately without going up the chain of "return" statements, by generating an exception in the DLL and returning control to TRNExe.exe. This error type should only be used as a last resort, e.g. when returning from that error would result in memory access violations or other exceptions. Exiting from a DLL by generating an exception is generally considered as bad programming and users should only use this technique if the cost of modifying the Type to handle the error properly is too high.

Note: severity is a Fortran literal string and can be enclosed in single or double quotes. Messages will understand some case variants of the codes here above. e.g. notice, Notice and NOTICE are accepted (other variants are not)

EXAMPLE

The following example illustrates calls to Messages with different severity levels by a Type for which par(1)>10 is not acceptable (values <10.1 are just rounded to 10 and the simulation goes on).

```
subroutine type200(time,xin,out,t,dtdt,par,info,iCntrl,*)
…
use TrnsysConstants
…
character (len=maxMessageLength) :: myMessage
integer info(15)
…
if ( par(1) < 10.0 ) then
    write(myMessage,'("Par(1) = ",g," , which is OK")') par(1)
    call Messages(-1,trim(myMessage),'Notice',info(1),info(2))
else if ( par(1) <= 10.1) then
    call Messages(-1,'Par(1) has been rounded to 10','Warning',info(1),info(2))
else
    call Messages(-1,'Par(1) is outside the acceptable range.',   &
                     'Fatal',info(1),info(2))
    return 1
endif
…
```

Please note the use of RETURN 1 following the call to Messages with a severity of ‘Fatal.’ When the severity of the message is low (notice, message, or warning), the simulation should proceed; thus no RETURN statement is required. In the case of a ‘fatal’ message, however, the simulation should be immediately stopped and the Type should return control to the TRNSYS kernel through use of the RETURN 1 statement.

8.4.4.10. ParRead

This is an undocumented subroutine used internally by some standard Types. See the Source code in ParRead.f90 for additional information.

8.4.4.11. PSYCHROMETRICS (PSYCH)

This utility routine is available to calculate various moist air properties based on ASHRAE correlations and the Ideal Gas Law. A double precision version called PSYCHROMETRICS and a legacy single precision version called PSYCH are available. It is recommended that all user written routines employ the double precision version and that all non integer arguments to the Psychrometrics routine be declared as DOUBLE PRECISION variables in the calling Type. A call to Psychrometrics is of the form

```
DOUBLE PRECISION :: TIME,PSYDAT
INTEGER :: INFO(15),IUNITS,MODE,WBMODE,EMODE,STATUS
...
CALL PSYCHROMETRICS(TIME,INFO,IUNITS,MODE,WBMODE,PSYDAT,EMODE,STATUS)
```

where

TIME	(double precision) This is the standard TIME used in TRNSYS. It is used in the PSYCHROMETRICS subroutine only to identify the time at which warnings or errors occur.
INFO	This is the standard integer INFO array used in TRNSYS (see Section 8.4.3). It is used in the PSYCHROMETRICS subroutine to identify the unit and type number of the component which calls the subroutine, which will be printed during a simulation if warnings or errors occur.
IUNITS	an integer variable equal to 1 or 2 identifying the units desired for the properties. IUNITS = 1 results in properties in SI units. IUNITS = 2 results in properties in English units.
MODE	an integer variable from 1 through 6. Three properties are needed to specify the moist air state. The MODE identifies which two properties besides pressure are need to be input, leaving the other properties to be calculated. The modes are as follows:

1:	input dry bulb and wet bulb temperatures (PSYDAT(2) and PSYDAT(3))
2:	input dry bulb temperature and relative humidity (PSYDAT(2) and PSYDAT(4))
3:	input dry bulb and dew point temperatures (PSYDAT(2) and PSYDAT(5))
4:	input dry bulb temperature and humidity ratio (PSYDAT(2) and PYSDAT(6))
5:	input dry bulb temperature and enthalpy (PSYDAT(2) and PSYDAT(7))
6:	input humidity ratio and enthalpy (PSYDAT(6) and PSYDAT(7)) . If saturation conditions occur, enthalpy is reset to saturation enthalpy at the given value of humidity ratio.
7:	input humidity ratio and enthalpy (PSYDAT(6) and PSYDAT(7)) . If saturation conditions occur, humidity ratio is reset to saturation humidity ratio at the given

value of enthalpy. 8: input relative humidity and enthalpy (PSYDAT(6) and PSYDAT(4)).

WBMODE	an integer variable equal to 0 or 1. If WBMODE = 0, the wet bulb temperature will not be calculated in MODEs 2 through 6. If WBMODE = 1, the wet bulb temperature is calculated. At PATM = 1.0 and temperatures common to HVAC applications, a correlation (1) is used to find the wet bulb temperature given enthalpy. For other pressures and temperatures, a Newton's iterative method is used with the equations given by ASHRAE (2) to calculate the wet bulb temperature. If the wet bulb temperature is not needed, WBMODE should be set to 0 to decrease the computational effort. If WBMODE = 0, the wet bulb temperature is set to the dry bulb temperature. PSYDATan array containing the properties of the moist air. This array must be dimensioned to 9 in the routine that calls PSYCHROMETRICS. Each location in this array is described below:
PSYDAT(1)	(double precision) The total system pressure in atmospheres. The subroutine prints a warning if PATM is over 5 atmospheres, which is considered here to be an upper limit for using ideal gas relations for air and water mixtures.
PSYDAT(2)	(double precision) dry bulb temperature (°C).
PSYDAT(3)	(double precision) wet bulb temperature (°C).
PSYDAT(4)	(double precision) relative humidity (fraction).
PSYDAT(5)	(double precision) dew point temperature (°C).
PSYDAT(6)	(double precision) humidity ratio (kg water/kg dry air).
PSYDAT(7)	(double precision) enthalpy (kJ/kg dry air).
PSYDAT(8)	(double precision) density of the air water mixture (kg/m3)
PSYDAT(9)	(double precision) density of the air portion of the mixture (kg dry air/m3)
EMODE	an integer variable equal to 0, 1, or 2 specifying how warnings are handled. If EMODE=0, no warnings will be printed. If EMODE=1, only one warning for each condition will be printed throughout the simulation. If EMODE=2, warnings will be printed every time step. (EMODE=2 is useful for program and simulation development, but it can produce an excessive amount of output).
STATUS	an integer variable equal to 0 through 13 identifying the warning condition that occurred within subroutine PSYCH. 0 indicates no warnings. For a complete listing of warnings 1 through 13 the subroutine source code should be reviewed.
N	the corresponding line number to jump to if the subprogram is present.

MATHEMATICAL DESCRIPTION

In determining a number of moist air properties the water vapor saturation pressure (PWS) is required. The correlations (ASHRAE, 2001) used for PWS are accurate over the temperature range of 100°C to 200°C. A warning is printed if moist air states occur outside this temperature range.

The correlations for the dew point temperature (Ashrae, 2001) are accurate over the temperature range of -60°C to 70°C. A warning is printed if moist air states occur outside this dew point range.

Enthalpies are based on the following reference states:

air enthalpy is zero at 0°C

liquid water enthalpy is zero at 0°C

This subroutine checks for many improper inputs, such as relative humidities less than 0. or greater than 1., dew point temperatures greater than the dry bulb temperature, and the input of two properties that cannot be correct for one state (such as a humidity ratio greater than saturation humidity ratio for dry air at a given dry bulb temperature). For these and other improper inputs, the subroutine prints a warning, corrects one of the inputs and continues, or prints an error and halts the simulation.

8.4.4.12. RCHECK

The RCHECK subroutine was added to TRNSYS 14 to provide input-output mismatch checking for all components, especially those that do not have the benefit of INPUT/OUTPUT unit connection checking as part of an interface program such as IISiBat or the TRNSYS Simulation Studio. A call to RCHECK is of the form:

```
INTEGER ::INFO(15)
CHARACTER(len=3) :: YCHECK(:),OCHECK(:)
...
CALL RCHECK(INFO,YCHECK,OCHECK)
```

where

INFO	This is the standard INFO array used in TRNSYS (see Section 8.4.3). It is used in the PSYCH subroutine to identify the unit and type number of the component which calls the subroutine, which will be printed during a simulation if warnings or errors occur.
YCHECK	A 3-character array containing the expected input types for the component.
OCHECK	A 3-character array containing the output types for the component

The RCHECK array is passed the input and output types for each component in the simulation and stores these variables in arrays with the same structure as the XIN and OUT arrays. When the TRNSYS processor checks for input-output mismatches, the arrays are inspected to ensure that the variable types for an input-output connection are consistent.

A user formulating a new component should utilize the input/output checking feature of TRNSYS for two reasons: input/output checking ensures that TRNSYS connections are correctly defined, and output information such as output type and output units are carried with the output and are able to be processed by the TYPE 25 printer and the TYPE 57 unit conversion routine.

The first step in input/output checking is to characterize the inputs and outputs using Table 3.4.12.1. Users wishing to create additional input-output types or add a unit conversion to an existing input-output type should modify the file “UNITS.LAB”, keeping the same format and style as the original. Refer to the TYPE 57 unit conversion routine for more details on modifying this file. The input-output types should be stored in the YCHECK and OCHECK arrays respectively.

The final step in using the input/output checking is to call the RCHECK array at the proper time, with the proper arrays. The RCHECK routine should be called during the first iteration in most cases (INFO(7)=-1 see Section 8.4.3) and after the call to the TYPECK subroutine (see Section 3.4.1).

The following example should help clarify the use of the RCHECK subroutine:

```
SUBROUTINE TYPE75(TIME,XIN,OUT,T,DTDT,PAR,INFO,ICNTRL,*)
!3 INPUTS AND 4 OUTPUTS FOR THIS TYPE
!INPUTS: TEMPERATURE IN  (F)
!        MASS FLOWRATE IN  (LBM/HR)
!        CONTROL SIGNAL  (0 OR 1)
!OUTPUTS: TEMPERATURE OUT  (C)
!          MASS FLOWRATE OUT (KG/HR)
!          HEAT TRANSFER RATE (KJ/HR)
!          HEAT TRANSFER RATE (W)
...
CHARACTER(len=3) :: YCHECK(3),OCHECK(4)
INTEGER :: INFO(15)
...
DATA YCHECK/'TE2','MF3','CF1'/
DATA OCHECK/'TE1','MF1','PW1','PW2'/
...
IF (INFO(7).EQ.-1) THEN
  ...
  CALL RCHECK(INFO,YCHECK,OCHECK)
  ...
ENDIF
```

Table 8.4.4.12-1: TEMPERATURE

VAR. TYPE #	VAR. UNITS	VAR.TYPE	MULT. FACTOR	ADD. FACTOR
1	°C	TE1	1.0	0
2	°F	TE2	1.8	32
3	K	TE3	1.0	273.15
4	R	TE4	1.8	492

Table 8.4.4.12-2: LENGTH

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	m	LE1	1	0
2	cm	LE2	100	0
3	km	LE3	1000	0
4	in	LE4	39.3701	0
5	ft	LE5	3.28084	0
6	miles	LE6	6.21371 E-04	0

Table 8.4.4.12-3: AREA

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	m2	AR1	1	0
2	cm2	AR2	1 E+04	0
3	km2	AR3	1 E-06	0
4	in2	AR4	1550	0
5	ft2	AR5	10.7639	0
6	mi2	AR6	3.86102 E-07	0

Table 8.4.4.12-4: VOLUME

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	m3	VL1	1	0
2	l	VL2	1000	0
3	ml	VL3	1 E+06	0
4	in3	VL4	6.10237 E+04	0
5	ft3	VL5	35.3147	0
6	gal	VL6	264.172	0

Table 8.4.4.12-5: SPECIFIC VOLUME

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	m3/kg	SV1	1	0
2	l/kg	SV2	1000	0
3	ft3/lbm	SV3	16.0185	0
4	in3/lbm	SV4	2.76799 E+04	0

Table 8.4.4.12-6: VELOCITY

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	m/s	VE1	1.0	0
2	km/hr	VE2	3.6	0
3	ft/s	VE3	3.28084	0
4	ft/min	VE4	196.85	0
5	mph	VE5	2.23694	0

Table 8.4.4.12-7: MASS

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	kg	MA1	1	0
2	g	MA2	1000	0
3	lbm	MA3	2.20462	0
4	ounces	MA4	35.274	0
5	ton	MA5	1.10231 E-03	0

Table 8.4.4.12-8: DENSITY

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	kg/m3	DN1	1.0	0
2	kg/l	DN2	0.001	0
3	lbm/ft3	DN3	6.2428 E-02	0
4	lbm/gal	DN4	8.3454 E-03	0

Table 8.4.4.12-9: FORCE

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	N	FR1	1.0	0
2	lbf	FR2	0.224809	0
3	ounce	FR3	3.59694	0

Table 8.4.4.12-10: PRESSURE

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	BAR	PR1	1	0
2	kPa	PR2	100	0
3	Pa	PR3	1 E+05	0
4	ATM	PR4	0.986923	0
5	psi	PR5	14.5038	0
6	lbf/ft2	PR6	2.08854 E+03	0
7	in. H2O	PR7	401.463	0
8	in. Hg	PR8	29.53	0

Table 8.4.4.12-11: ENERGY

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	kJ	EN1	1	0
2	kWh	EN2	2.77778 E-04	0
3	Cal	EN3	238.846	0
4	ft-lbf	EN4	737.562	0

5	hp-hr	EN5	3.72506 E-04	0
6	BTU	EN6	0.947817	0

Table 8.4.4.12-12: POWER

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	kJ/hr	PW1	1	0
2	W	PW2	0.277778	0
3	kW	PW3	2.77778 E-04	0
4	hp	PW4	3.72505 E-04	0
5	BTU/hr	PW5	0.947817	0
6	BTU/min	PW6	1.57969 E-02	0
7	Tons	PW7	7.89847 E-05	0

Table 8.4.4.12-13: SPECIFIC ENERGY

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	kJ/kg	SE1	1	0
2	BTU/lbm	SE2	0.429923	0
3	ft-lbf/lbm	SE3	334.553	0

Table 8.4.4.12-14: SPECIFIC HEAT

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	kJ/kg-K	CP1	1	0
2	W-hr/kg-K	CP2	0.277778	0
3	BTU/lbm-R	CP3	0.238846	0

Table 8.4.4.12-15: FLOW RATE

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	kg/hr	MF1	1	0
2	kg/s	MF2	2.77778 E-04	0
3	lbm/hr	MF3	2.20462	0
4	lbm/s	MF4	6.12395 E-04	0

Table 8.4.4.12-16: VOLUMETRIC FLOW RATE

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	m3/hr	VF1	1	0
2	m3/s	VF2	2.77778 E-04	0
3	l/hr	VF3	1000	0
4	l/s	VF4	0.277778	0
5	ft3/s	VF5	9.80958 E-03	0
6	ft3/hr	VF6	35.3144	0
7	gpm	VF7	4.40286	0

Table 8.4.4.12-17: FLUX

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	kJ/hr-m2	IR1	1	0
2	W/m2	IR2	0.277778	0
3	BTU/hr-ft2	IR3	8.8055 E-02	0

Table 8.4.4.12-18: THERMAL CONDUCTIVITY

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	kJ/hr-m-K	KT1	1	0
2	W/m-K	KT2	0.277778	0
3	BTU/hr-ft-R	KT3	0.160497	0

Table 8.4.4.12-19: HEAT TRANSFER COEFFICIENTS

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	kJ/hr-m2-K	HT1	1	0
2	W/m2-K	HT2	0.277778	0
3	BTU/hr-ft2-R	HT3	4.89194 E-02	0

Table 8.4.4.12-20: DYNAMIC VISCOSITY

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	N-s/m2	VS1	1	0
2	kg/m-s	VS2	1	0
3	poise	VS3	10	0
4	lbf-s/ft2	VS4	2.08854 E-02	0
5	lbf-hr/ft2	VS5	5.80151 E-06	0
6	lbm/ft-hr	VS6	2419.08	0

Table 8.4.4.12-21: KINEMATIC VISCOSITY

VAR. TYPE #	VAR. UNITS	VAR. TYPE	MULT. FACTOR	ADD. FACTOR
1	m2/s	KV1	1	0
2	m2/hr	KV2	3600	0
3	ft2/s	KV3	10.7639	0
4	ft2/hr	KV4	3.87501 E+04	0

Table 8.4.4.12-22: MISCELLANEOUS

	VAR. TYPE
Dimensionless	DM1
Degrees	DG1
Percentage	PC1
Month	MN1
Day	DY1
Hour	TD1
Control function	CF1
Volts	VO1
Ampere	CU1
Ohm	RE1
Ampere-hour	AH1

In user-written routines that have output variable types dependent on input variable types, such as a statistics or integrator routine, the call to RCHECK must come on the second iteration of the first time step, after a call to the EQUATER subroutine. The EQUATER subroutine will determine the output connected to the unknown input and provide the appropriate input type. A call to the EQUATER subroutine has the following form:

```
CHARACTER(len=3) :: TRUTYP,INDES,ODES
DOUBLE PRECISION :: X1,A1,X2,A2
INTEGER :: IU,INNO, OUTU,OUTNO,ICK1,ICK2

CALL EQUATER(IU,INNO,INDES,TRUTYP,OUTU,OUTNO,ODES,X1,A1,X2,A2,ICK1,ICK2)
```

where

IU	the unit number of the user-written component routine calling the EQUATER subroutine (integer variable)
INNO	the input number with the undetermined input type (integer variable)
INDES	the desired input variable type - not used in this capacity, set to ‘NAV’ for not available (3-character variable)
TRUTYP	a 3 character array returning the output variable type connected to this input (this is the desired result for this application) (3 character variable)
OUTU	the unit number of the output connected to the unknown input (not used in this capacity) (integer variable)
OUTNO	the output number of the OUTU unit connected to the unknown input (not used in this capacity) (integer variable)
ODES	the desired output variable type for this unknown input - not used in this capacity (3-character variable)
X1	the multiplication factor for the INDES variable - not used in this capacity (double precision value)
A1	the addition factor for the INDES variable - not used in this capacity (double precision value)
X2	the multiplication factor for the OUTDES variable - not used in this capacity (double precision value)
A2	the addition factor for the OUTDES variable - not used in this capacity (double precision value)
ICK1	error flag for calculation (1 = error found) (integer variable)
ICK2	error flag for calculation (1 = error found) (integer variable)

Users attempting to use the input/output checking algorithm in this capacity should refer to the TYPE 55 Periodic Integrator subroutine and the EQUATER subroutine source code for more details.

8.4.4.13. REWIND

The Rewind Utility is designed to rewind a data file whose end has been reached through multiple read statements. It is able to then skip a specified number of steps from the beginning of the file and leave a pointer ready to read the first non skipped line. A call to Rewind should have the following form:

```
INTEGER :: LU,SKIP,IU,IT,IE
...
CALL REWIND(LU,SKIP,IU,IT,IE)
```

where

LU	(integer) the Fortran logical unit number ASSIGNed to the data file that is to be rewound.
SKIP	(integer) the number of lines that should be skipped in the data file once it is rewound.
IU	(integer) the unit number of the Type calling the Rewind routine.
IT	(integer) the type number of the Type calling the Rewind routine.
IE	(integer) an error code that is returned by the Rewind routine and is set to zero if no errors were encountered while rewinding the data file or skipping lines and which is set to a vale of 1 if errors were encountered.

EXAMPLE

The following example shows the usage of the Rewind routine and a possible method for checking whether errors occurred during the rewind process.

```
INTEGER :: LU,SKIP,IU,IT,IE,INFO(15)
...
LU = JFIX(PAR(1)+0.1)
SKIP = 10
IU = INFO(1)
IT = INFO(2)
CALL REWIND(LU,SKIP,IU,IT,IE)
IF (IE.GT.1) THEN
  RETURN 1
ENDIF
...
```

8.4.4.14. SOLCEL

SOLCEL Models silicon cell I-V characteristics

Contributor:	Professor Don L. Evans, Arizona State University. Tempe, AZ 85281
	(provided as-is. Check the source code for additional information)

The SOLCEL subroutine is called by Types 49 and 50 and uses a set of empirical values to model the cells. Data for Spectrolab type cells, provided by Professor Evans, is "built-in" and is used by default if no other data are supplied by the user. Users may supply their own data by adding an extra parameter to the list of TYPE 49 or TYPE 50 in modes 5-8. This extra parameter is interpreted as the logical unit number of the file containing the cell characteristics. The data are read using 9F10.0 format and should be ordered as follows; (default values are listed after the definition of the parameter):

First record:

ICELL	Cell type indicator, 0 for Spectrolab and Solarix Types, 1 for RCA types (peak power tracking only for RCA); (0).
CELLPF	Ratio of cell area to absorber area; if ICELL = 1, the numerical value is ignored, but a number must be present; (0.8).

Second record:

TL	A low temperature reference (˚ C); (43)
TH	A high temperature reference (˚ C); (175)
CL	A low reference concentration; (6)
CH	A high reference concentration; (20)
VOCLL	Open circuit voltage of a cell at temperature TL and concentration CL (volts); (0.625)
ISCLL	Short circuit current of the cell at temperature TL and concentration CL (amps); (0.649)
ISCLH	Short circuit current of the cell at temperature TL and concentration CH (amps); (0.625)
ISCHL	Short circuit current of the cell at temperature TH and concentration CL(amps); (2.16)
ISCHH	Short circuit current of the cell at temperature TH and concentration CH(amps); (2.16)

Third record (Not used if ICELL = 1):

ACELL	Gross cell area (m2); (0.0004)
EG	Cell band gap at 0 Kelvin; (14000)
AOMIN	Value of Ao. at concentration of 0; (1)
DAODC	Slope of the Ao. vs concentration curve; (0.018)
RSH	Shunt resistance (ohms); (10000)
RSMIN	Value of series resistance when it becomes essentially independent of concentration (ohms); (0.01)

CONTR	Minimum concentration for which the series resistance can be considered to be equal to RSMIN; (10)
DRSDC	Slope of the series resistance vs concentration curve below a concentration CONTR; (-0.005)
NSER	Number of cells to be wired in series in the array (10)

8.4.4.15. STEAM_PROPS (STEAM)

The double precision STEAM_PROPS routine and its single precision counterpart STEAM are used to calculate the thermodynamic properties of steam based on correlations from the National Bureau of Standards. Given two state properties, the STEAM_PROPS routine will return the complete state of the fluid. A call to STEAM_PROPS is of the form

```
CHARACTER(len=2) :: UNITS
DOUBLE PRECISION :: PROP(7)
INTEGER :: ITYPE,IERR
...
CALL STEAM_PROPS(UNITS,PROP,ITYPE,IERR,*N)
```

where

UNITS	two character variable denoting unit system to be employed by the fluids routine; ‘SI’ for metric units and ‘EN’ for english units
PROP	double precision array holding the two known thermodynamic properties of steam upon entry and the complete state upon return from the STEAM_PROPS routine. This array must be dimensioned to 7 in the routine that calls STEAM_PROPS. Each location in this array is described below:
PROP(1)	(double precision) temperature of the steam (F, C)
PROP(2)	(double precision) pressure of the steam (PSI, MPa)
PROP(3)	(double precision) enthalpy of the steam (BTU/lbm, kJ/kg)
PROP(4)	(double precision) entropy of the steam (BTU/lbm R, kJ/kg K)
PROP(5)	(double precision) quality (0 < x < 1)
PROP(6)	(double precision) specific volume (ft3/lbm, m3/kg)
PROP(7)	(double precision) internal energy (BTU/lbm, kJ/kg)
ITYPE	integer variable denoting which two properties are supplied as known properties. ITYPE = 10* PROP indicator 1 + PROP indicator 2 (12 to 76)
IERR	an integer variable returning the error messages from the STEAM_PROPS call:
	= 0 - no error found, calculations completed
	> 0 - error found, calculations could not be completed
N	the corresponding line number to jump to if the subprogram is present

An example illustrating the use of the STEAM_PROPS routine is shown with the following considerations: SI units are desired, and Temperature (PROP(1)) and Quality (PROP(5)) are known.

```
USE TrnsysFunctions
...
CHARACTER(len=2) :: UNITS
DOUBLE PRECISION :: PROP(7),XIN(2),TEMP,QUALITY,PRESSURE,ENTHALPY
...
```

```
       UNITS = ‘SI’  !SI units are desired.
       ...
       TEMP = XIN(1)
       QUALITY = XIN(2)
       ...
       PROP(1) = TEMP
       PROP(5) = QUALITY
       CALL STEAM_PROPS(UNITS,PROP,INFO(1),INFO(2),IFLAG,*100)
       CALL LINKCK('TYPE58','STEAM_PROPS',1,58)
100    IF (ErrorFound()) RETURN 1
       ...
       PRESSURE=PROP(2)
       ENTHALPY=PROP(3)
```

MATHEMATICAL DESCRIPTION

The correlations required to solve for the steam properties are from the EES program (8), a numerical solver employing thermodynamic correlations from many different sources. Interested users should contact the reference for more details on the correlations.

Enthalpies for steam are based on the following reference state: enthalpy equal to zero at 0°C

This subroutine checks for much improper input, such as qualities less than 0. or greater than 1., and the input of two properties that cannot be correct for one state. For these and other improper inputs, the subroutine prints a warning, corrects one of the inputs, and continues; or prints an error and halts the simulation.

Subcooled properties are not available for steam. If a subcooled state is specified, the saturated liquid results will instead be provided.

8.4.4.16. *Storage of Data Between Time Steps*

Since a single Type subroutine may be used for several UNITS in one simulation, care must be taken when storing values that are to be used from one time step to the next. If a tank model, for instance, requires the initial tank temperature throughout the simulation, simply setting a variable TZERO to the tank temperature when INF0(7) = -1 will not work. When two tanks are used, the local variable TZERO will be set twice. Throughout the simulation, it will contain the initial temperature of the second tank. As a result, the first tank will not be modeled correctly.

One solution to this problem is to use the OUT array. Since each unit is allocated a minimum of 20 outputs, there is often enough space to store a few extra values which are not actual outputs. The number of outputs can be increased beyond 20 by using INFO(6) (see section 8.4.3.2). However, for large storage requirements (greater than 10 or so), it is recommended that a series of calls to utility routine be used instead. With the release of TRNSYS 16, these utility routines replace the “S array” and the STORE common block as the recommended solution to data storage by components.

The first of the three utility routines reserves a requested number of spots in the global data structure. The second routine retrieves the values from the global storage structure that correspond to the current Type. The third sets new values in the global structure for the current Type. The following section provides general information about the manner in which the three storage related utility routines are called. An example of using storage in a Type follows.

SETSTORAGESIZE

INTERFACE

```
subroutine setStorageSize(n,INFO)
    integer :: n, INFO(15)
```

DESCRIPTION

The setStorageSize subroutine should be called when INFO(7) = -1. This will tell the storage data structure how many spots it needs to reserve for your component. The subroutine has two arguments:

- n: The integer number of storage spots required by this component.
- INFO: The standard INFO array is sent so that the storage data structures know what unit and type number are requesting storage.

GETSTORAGEVARS

INTERFACE

```
subroutine getStorageVars(storageArray,m,INFO)
    integer :: m, INFO(15)
    double precision :: storageArray(*)
```

DESCRIPTION

When the user wants to obtain a previously stored value (usually at the first call of a new time step, i.e. when INFO(7) = 0), a call is made to the getStorageVars subroutine. The subroutine takes the following arguments:

- StorageArray: a locally defined double precision array dimensioned to n (see setStorageSize above) that will contain the stored variables that are being retrieved from storage.
- m: The integer number of variables that are to be retrieved. This argument allows the user to retrieve only a subset of the stored variables. For example, if only the 5th stored variable is desired, setting m to 5 will retrieve stored variables 1 through 5 and place them into spots 1 through 5 of storageArray
- INFO: the standard INFO array is sent so that the storage data structures know what unit and type number are requesting storage

SETSTORAGEVARS

INTERFACE

```
subroutine setStorageVars(storageArray,m,INFO)
    integer :: m, INFO(15)
    double precision :: storageArray(*)
```

DESCRIPTION

When the user wants to set storage at the end of a time step, a call is made to the setStorageVars subroutine when INFO(13) > 0. Much like the getStorageVars subroutine, the setStorageVars subroutine takes the following arguments:

- storageArray: a locally defined double precision array dimensioned to n (see setStorageSize above) that will contain the stored variables that are being retrieved from storage.
- m: the integer number of variables that are to be stored. This argument allows the user to store only a subset of an array that is used in the Type.
- INFO: the standard INFO array is sent so that the storage data structures know what unit and type number are requesting storage.

STORAGE EXAMPLE

A tank model which uses global storage to store the initial tank temperature might contain the following lines:

```
...
C PERFORM FIRST CALL MANIPULATIONS
  IF (INFO(7).EQ.-1) THEN
    ...
    !reserve space in the double precision storage structure
    StorageSize = 1
    CALL setStorageSize(StorageSize,INFO)
    ...
    !return to the calling program
    RETURN 1
  ENDIF

C PERFORM INITIAL TIME STEP MANIPULATIONS
  IF (TIME.LT.(TIME0+DELT/2.)) THEN
    ...
    !set initial values of variables in the double precision storage structure
    Stored(1) = T
    CALL setStorageVars(Stored, StorageSize,INFO)
    ...
    !return to the calling program
    RETURN 1
  ENDIF
...
CALL getStorageVars(Stored,StorageSize,INFO)
TZERO = Stored(1)
...
```

The above example is exceedingly simplistic. The result of it is that no matter what is done to the local variable TZERO later in the Type, it will always be reset to the initial value of T each time the Type is called anew. A much more common situation arises when not only do we wish to use an initial value in calculations but at the end of the time step, we wish to set the final calculated value as the initial value for the next time step. In TRNSYS version 15.x and before, there was no way for a Type to know when it was being called for the last time in a given time step. With the release of TRNSYS version 16, a "post convergence" call to all Types was added, simplifying this process. In the following modification to the above example, TZERO is set to an initial value. TFINAL is calculated based on TZERO. At the end of each time step, the converged value of TFINAL is set as the value of TZERO for the next time step.

```
...
C PERFORM POST CONVERGENCE MANIPULATIONS
  IF (INFO(13).GT. 0) THEN
    CALL getStorageVars(Stored,StorageSize,INFO) !recall the storage values
    Stored(1)=Stored(2)          !update the TZERO spot with the value stored
                                 !in the TFINAL spot.
    CALL setStorageVars(Stored,StorageSize,INFO) !set the new storage values
  ENDIF

C PERFORM FIRST CALL MANIPULATIONS
  IF (INFO(7).EQ.-1) THEN
    ...
    !reserve space in the double precision storage structure
    StorageSize = 2
    CALL setStorageSize(StorageSize,INFO)
    ...
    !return to the calling program
    RETURN 1
  ENDIF

C PERFORM INITIAL TIME STEP MANIPULATIONS
  IF (TIME.LT.(TIME0+DELT/2.)) THEN
    ...
    !set initial values of variables in the double precision storage structure
    Stored(1) = T
    Stored(2) = T
    CALL setStorageVars(Stored, StorageSize,INFO)
    ...
    !return to the calling program
    RETURN 1
  ENDIF
...
!retrieve the stored value of the initial temperature for this time step.
CALL getStorageVars(Stored,StorageSize,INFO)
TZERO = Stored(1)
...
!calculate TFINAL based on the initial temperature.
TFINAL = TZERO + ...
...
!update the second storage spot with the calculated value of TFINAL
Stored(2) = TFINAL
CALL setStorageVars(Stored,StorageSize,INFO)
...
```

8.4.4.17. TABLE_COEFS (TABLE)

The double precision subroutine TABLE_COEFS and its single precision companion TABLE return transfer function coefficients corresponding to the standard walls, partitions, floors, and ceilings given in the ASHRAE Handbook of Fundamentals. “Double Precision” indicates that the arguments to the TABLE_COEFS routine must be declared in the calling Type as “DOUBLE PRECISON” instead of as REAL variables. The single precision version (called TABLE) was the only version available in TRNSYS version 15.x and below. It was retained for backwards compatibility but every effort should be made to use the double precision version: TABLE_COEFS. The TABLE_COEFS routine is used by the TYPE 19 Zone Model. A call to TABLE_COEFS is of the form

```
INTEGER :: ITYPE,NTABLE, IU,NBS,NCS,NDS
DOUBLE PRECISION :: B,C,D
...
CALL TABLE_COEFS(ITYPE,NTABLE,IU,NBS,NCS,NDS,B,C,D,*N)
```

where

ITYPE	an integer variable equal to 1, 2, or 3 denoting which type of surface is desired. ITYPE = 1 corresponds to roof coefficients, ITYPE = 2 is for exterior walls, and ITYPE = 3 denotes interior partitions, floors and ceilings. The tables containing the surface descriptions can be found in the documentation for TYPE 19; both in this manual and in the Component Description manual.
NTABLE	(integer) number of wall, roof, etc., from the table specified by ITYPE.
NBS	(integer) the number of b coefficients for the desired wall as returned by the subroutine.
NCS	(integer) the number of c coefficients for the desired wall as returned by the subroutine.
NDS	(integer) the number of d coefficients for the desired wall as returned by the subroutine.
B	a double precision array dimensioned to 10 that contains the b coefficients. These are the coefficients for the current and previous sol-air temperatures (i.e. bo to bNBS-1).
C	a double precision array dimensioned to 10 that contains the c coefficients. These are the coefficients for the current and previous room temperatures (i.e. co to cNCS-1).
D	a double precision array dimensioned to 10 that contains the d coefficients. These are the coefficients for the previous heat fluxes (d1 to dNDS).
N	the corresponding line number to jump to if the subprogram is present.

Subroutine Table_Coefs reads the transfer function coefficient from the file ASHRAE.COF accessed through FORTRAN logical unit 8. This logical unit can be changed by modifying a data statement within the subroutine. After the coefficients have been read for a particular wall, the coefficients are stored internally within the subroutine for use on subsequent calls to Table_Coefs.

Note: Transfer function coefficients in the file ASHRAE.COF are different from those actually found in the ASHRAE handbooks. Because the TYPE 19 single-zone model handles the inside radiative heat transfer separately, the inside radiative resistance is not included in the ASHRAE.COF transfer coefficients. Thus while the file ASHRAE.COF is well-suited for use with TYPE 19 and other models which calculate radiative heat transfer separately, it must not be assumed that this file matches the ASHRAE handbook values.

8.4.4.18. TAU_ALPHA (TALF)

GENERAL DESCRIPTION

The double precision function Tau_Alpha and its single precision counterpart TALF calculate the transmittance-absorptance product ($\tau\alpha$) of a solar collector as a function of angle of incidence of radiation and collector construction. The transmittance τ of one or more sheets of glass or plastic can be obtained by specifying an absorptance of 1. Tau_Alpha is a double precision routine, meaning that all but its first argument must be declared in the calling Type as double precision variables (as opposed to REAL variables). The value returned by Tau_Alpha is also double precision. In TRNSYS versions 15.x and below, Tau_Alpha was known simply as “TALF,” which was a single precision routine. TALF was retained for backwards compatibility and every effort should be made to use the double precision Tau_Alpha routine.

A call to function TAU_ALPHA is of the form

```
DOUBLE PRECISION :: TAUALF,TAU_ALPHA,THETA,XKL,REFRIN,ALPHA,RHOD
INTEGER :: N
TAUALF = TALF(N,THETA,XKL,REFRIN,ALPHA,RHOD)
```

The function arguments are interpreted as follows:

N	(integer) number of covers
THETA	(double precision) angle of incidence (degrees)
XKL	(double precision) product of cover thickness and extinction coefficient
REFRIN	(double precision) index of refraction
ALPHA	(double precision) absorber surface absorptance
RHOD	(double precision) reflectance of the N covers to diffuse radiation

If the value of RHOD is not known, then TALF can calculate it: When the sixth argument in the function reference is less than zero, the reflectance of the N covers to diffuse radiation is determined. The variable used as the sixth argument in the call to TALF is then set to this value so that on subsequent calls to TALF the value of RHOD may be supplied.

Due to FORTRAN considerations, functions can not be checked for unlinked subroutines. Use the LINKCK subroutine to warn the user that this function must be linked to the TRNSYS executable.

MATHEMATICAL DESCRIPTION

The function subprogram Tau_Alpha calculates the transmittance-absorptance product ($\tau\alpha$) of a solar collector for a given angle of incident radiation. The collector is described by the number of glazings, N, each with refractive index n_g and extinction length kL, and the absorptance of the collector plate α. The model uses Fresnel's equation for the specular reflectance at a planar interface (assuming unpolarized incident radiation) and accounts for multiple reflections. The transmittance for diffuse radiation is approximated as the transmittance for specular radiation at an incident angle of 60°. Fresnel's equations for reflectance at a planar interface for perpendicular and parallel polarized radiation are the following:

<u>perpendicular</u> <u>parallel</u>

$$\rho_1 = \frac{\sin^2(\theta_1 - \theta_2)}{\sin^2(\theta_1 + \theta_2)} \quad \text{Eq. 8.4.4-14}$$

$$\rho_2 = \frac{\tan^2(\theta_1 - \theta_2)}{\tan^2(\theta_1 + \theta_2)} \quad \text{Eq. 8.4.4-15}$$

The angles θ_1 and θ_2 are related by Snell's Law with the index of refraction for air of unity.

$$\sin\theta_2 = \frac{\sin\theta_1}{n_g} \quad \text{Eq. 8.4.4-16}$$

The transmittance and reflectance of a single glazing for each component of polarization, including absorption and multiple reflections within the glazings, can be derived to give the following [1,2]:

$$T_{i,1} = \frac{\tau_a(1-\rho_i)^2}{1-\tau_a{}^2\rho_i{}^2} \quad \text{Eq. 8.4.4-17}$$

$$R_{i,1} = \rho_i\left[1 + \frac{\tau_a{}^2(1-\rho_i)^2}{1-\tau_a{}^2\rho_i{}^2}\right] \quad \text{Eq. 8.4.4-18}$$

The internal transmittance, τ_a, is a function of the extinction length and the transmission angle.

$$\tau_a = \exp\left(\frac{-kL}{\cos\theta_2}\right) \quad \text{Eq. 8.4.4-19}$$

The transmittance and reflectance of N parallel plates, of the same material and thickness, can be determined by the following recursive formulae.

$$T_{i,N} = \frac{T_{i,1}T_{i,N-1}}{1-R_{i,1}R_{i,N-1}} \quad \text{Eq. 8.4.4-20}$$

$$R_{i,N} = R_{i,N-1} + \frac{R_{i,1}T_{i,N}}{1-R_{i,1}R_{i,N-1}} \quad \text{Eq. 8.4.4-21}$$

The transmittance-absorptance product of the collector plate and glazing assembly can then be calculated as

$$(\tau\alpha) = \frac{(T_{1,N} + T_{2,N})\alpha}{2(1-(1-\alpha)R_D)} \quad \text{Eq. 8.4.4-22}$$

where R_D is the reflectance of the N covers to diffuse radiation. R_D is calculated as the reflectance for specular radiation at 60° incident angle.

8.4.4.19. TYPECK

The purpose of TYPECK is to perform some simple checks on the TRNSYS input file to ensure the proper number of INPUTS, PARAMETERS, and DERIVATIVES have been specified for the given component, and to terminate the simulation after compilation of the component input file with appropriate error messages if errors are found. The form of a TYPECK call is:

```
INTEGER :: IOPT,INFO(15),NI,NP,ND
...
CALL TYPECK(IOPT,INFO,NI,NP,ND)
```

where

IOPT-	indicates to TYPECK what kind of check to perform:	
	<0	the compilation will be terminated immediately after printing an explanatory error message.
	>0	the simulation will be terminate after all UNITS have been called once or 20 error messages have been issued.
	1 or -1	check the number of user furnished INPUTS, PARAMETERS, and DERIVATIVES from the INFO array against the values of NI, NP, and ND. An error message will be printed only if TYPECK finds an error.
	2 or -2	TYPE routine has found unspecified error and wants TYPECK to flag it.
	3 or -3	TYPE routine has found a bad INPUT or number of INPUTS and wants TYPECK to flag it.
	4 or -4	TYPE routine has found a bad PARAMETER or number of PARAMETERS and wants TYPECK to flag it.
	5 or -5	TYPE routine has found a bad DERIVATIVE or number of DERIVATIVES and wants TYPECK to flag it.
	6	TYPE routine has determined that the maximum number of UNITS of that TYPE has been exceeded and wants TYPECK to flag it and stop the simulation.
INFO-	TYPE's INFO array containing pertinent bookkeeping data on the UNIT (see Section 8.4.3).	
NI-	correct number of INPUTS	
NP-	correct number of PARAMETERS	
ND-	correct number of DERIVATIVES	

8.4.4.20. VIEW_FACTORS (VIEW)

GENERAL DESCRIPTION

The double precision function View_Factors (and its single precision companion VIEW) calculate the view factor between two planer rectangles of any orientation. For common orientations, analytical expressions are used. With A call to View_Factors is of the form:

```
F21 = VIEW(ITYPE,A,B,C,D,E,F,G,H,X3,Y3,Z3,NA1,NA2)
```

The integer variable ITYPE defines the type of orientation between the two surfaces. There are five possibilities. The arguments A, B, C, D, E, F, G, H, X3, Y3, and Z3 (all double precision) are dimensions or coordinates that depend upon the orientation chosen. Not all of these variables are used for each orientation. The possible orientations and required dimensions are shown in Figures 8.6.7.16-1 to 8.6.7.16-5. The VIEW factor from surface 2 to 1 is determined by the routine. The integer arguments NA1 and NA2 are the number of differential areas for each surface, when numerical integration is used to determine VIEW factors. NA1 is only used for ITYPE equal to 3, 4, or 5. NA2 is only for ITYPE equal to 5.

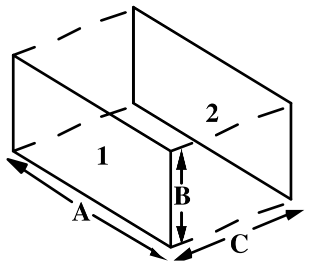


Figure 8.4.4.20-1:ITYPE=1; parallel identical rectangles.

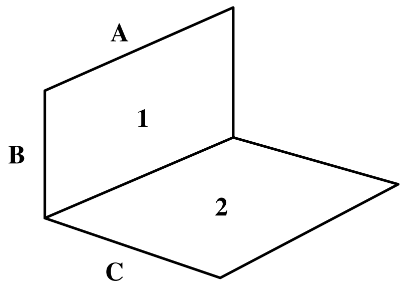


Figure 8.4.4.20-2: ITYPE=2; perpendicular rectangles with a common edge.

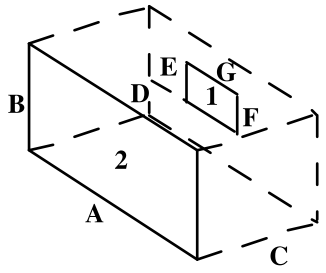


Figure 8.4.4.20-3: ITYPE=3; nonidentical parallel rectangles.

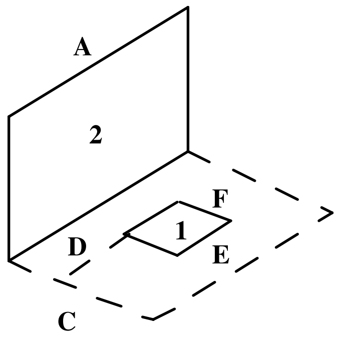


Figure 8.4.4.20-4: ITYPE=4; perpendicular rectangles without a common edge.

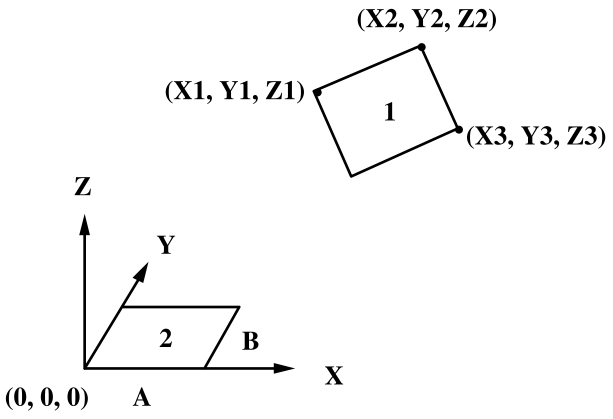


Figure 8.4.4.20-5: ITYPE=5; two rectangular surfaces of any orientation (X1=C, Y1=D, Z1=E, X2=F, Y2=G, Z2=H from subroutine call statement).

MATHEMATICAL DESCRIPTION:

ITYPE = 1

The view factor from surface two to one of Figure 8.4.4.20-1 is determined with the following expression taken from Reference 1.

$$F_{21} = \frac{2}{\pi XY}\left\{ \ln\left[\frac{(1+X^2)(1+Y^2)}{1+X^2+Y^2}\right]^{1/2} + \sqrt{1+Y^2}\tan^{-1}\frac{X}{\sqrt{1+Y^2}} \right.$$

Eq. 8.4.4-23

$$\left. + Y\sqrt{1+X^2}\tan^{-1}\frac{Y}{\sqrt{1+X^2}} - X\tan^{-1}X - Y\tan^{-1}Y \right\}$$

where,

X = A/C
and
Y = B/C

ITYPE = 2

From Reference 1, the view factor from surface two to one of Figure 8.4.4.20-2 is

$$F_{21} = \frac{1}{\pi Y}\left\{ Y \tan^{-1}\frac{1}{Y} + X \tan^{-1}\frac{1}{X} - \sqrt{X^2+Y^2}\tan^{-1}\frac{1}{\sqrt{X^2+Y^2}} \right.$$
$$\left. + \frac{1}{4}\ln\left[\frac{(1+Y^2)(1+X^2)}{1+X^2+Y^2}\left(\frac{Y^2(1+X^2+Y^2)}{(1+Y^2)(X^2+Y^2)}\right)^{Y^2}\left(\frac{X^2(1+X^2+Y^2)}{(1+X^2)(X^2+Y^2)}\right)^{X^2}\right]\right\} \quad \text{Eq. 8.4.4-24}$$

where,

X = B/A
and
Y = C/A

ITYPE = 3

The VIEW factor from surface two to one of Figure 8.6.7.16-3 is found by numerical integration using the following expression:

$$F_{21} = \frac{\int_{A_1} F_{d1-2}\, dA_1}{A_2} \quad \text{Eq. 8.4.4-25}$$

where F_{d1-2} is the view factor from an elemental area dA_1 on surface 1 to surface 2 and A_1 and A_2 are the areas of surfaces 1 and 2, respectively. The number of differential areas used in the integration, NA1, is specified as an argument in the call to VIEW.

The view factor F_{d1-2} is determined by breaking surface 2 into 4 sections such that the normal to the center of the element passes through a common corner of each section of surface 2.

From Reference 1,

$$F_{d1-2} = \sum_{i=1}^{4}\frac{1}{2\pi}\left(\frac{X_i}{\sqrt{1+X_i^2}}\tan^{-1}\frac{Y_i}{\sqrt{1+X_i^2}} + \frac{Y_i}{\sqrt{1+Y_i^2}}\tan^{-1}\frac{X_i}{\sqrt{1+Y_i^2}}\right) \quad \text{Eq. 8.4.4-26}$$

where

$$X_i = \frac{A_i}{C} \text{ and } Y_i = \frac{B_i}{C}$$

and A_i and B_i are dimensions of each section on surface 2, measured in the same directions as A and B.

ITYPE = 4

The view factor from surface 2 to 1 of Figure 8.4.4.20-4 is also determined using Equation 8.6.7.16-3 The view factor F_{d1-2} is found by breaking surface 4 into 2 sections such that the normal to a common corner of the sections splits the element in half. From Reference 1,

$$F_{d1-2} = \sum_{i=1}^{2} \frac{1}{2\pi}\left(\tan^{-1}\frac{1}{Y_i} - \frac{Y_i}{\sqrt{X_i^{\,2} + Y_i^2}} \tan^{-1}\frac{1}{\sqrt{X_i^{\,2} + Y_i^2}} \right) \qquad \text{Eq. 8.4.4-27}$$

where

$$X_i = \frac{A}{B_i} \text{ and } Y_i = \frac{C'}{B_i}$$

B_i is the dimension of each section measured in the direction of B. C' is the distance from the edge of surface 2 to the center of the element on surface one.

ITYPE = 5

For two rectangular surfaces of arbitrary orientation as shown in Figure 8.4.4.20-5, the view factor from 2 to 1 is:

$$F_{21} = \frac{1}{A_2}\int_{A_1}\int_{A_2} \frac{\cos\theta_1 \cos\theta_2}{\pi S^2} dA_2 dA_1 \qquad \text{Eq. 8.4.4-28}$$

where S is the distance between the centers of elemental areas on each surface, θ_1 and θ_2 are angles between the normals to each element and the line connecting their centers, and A_1 and A_2 are the areas of surfaces 1 and 2. Equation (8.6.7.16-6) is solved by numerical integration. The number of elemental areas used in the integration procedure, NA1 and NA2, are specified in the argument list in the call to VIEW.

The distance S is

$$S = \sqrt{(X_{d2} - X_{d1})^2 + (Y_{d1} - Y_{d2})^2 + (Z_{d2} - Z_{d1})^2} \qquad \text{Eq. 8.4.4-29}$$

where (X_{d1},Y_{d1},Z_{d1}) and (X_{d2},Y_{d2},Z_{d2}) are the coordinates of the elemental areas on surfaces 1 and 2, respectively. For Surface 2,

$$\cos\theta_2 = \frac{Z_{d1} - Z_{d2}}{S} \qquad \text{Eq. 8.4.4-30}$$

For Surface 1,

$$\cos\theta_1 = \frac{\vec{S}\cdot\vec{N}}{|N|\,|S|} \qquad \text{Eq. 8.4.4-31}$$

where $\vec{S}$ is the vector originating at the center of the elemental area on surface 1 and ending at the center of the second element on surface 2 and $\vec{N}$ is a normal vector to the elemental area on surface 1.

In general,

$$\vec{S} = s_x\vec{i} + s_y\vec{j} + s_z\vec{k} \qquad \text{Eq. 8.4.4-32}$$

$$\vec{N} = n_x\vec{i} + n_y\vec{j} + n_z\vec{k} \qquad \text{Eq. 8.4.4-33}$$

where $\vec{i}$, $\vec{j}$, and $\vec{k}$ are unit vectors in the x, y, and z directions, respectively. Substitution of Equations (8.6.7.16-10) and (8.6.7.16-11) into Equation (8.6.7.16-9) gives

$$\cos\theta = \frac{S_x n_x + S_y n_y + S_z n_z}{\sqrt{S_x{}^2 + S_y{}^2 + S_z{}^2}\sqrt{n_x{}^2 + n_y{}^2 + n_z{}^2}} \qquad \text{Eq. 8.4.4-34}$$

where,

$$S_x = X_{d2} - X_{d1} \qquad \text{Eq. 8.4.4-35}$$

$$S_y = Y_{d2} - Y_{d1} \qquad \text{Eq. 8.4.4-36}$$

$$S_z = Z_{d2} - Z_{d1} \qquad \text{Eq. 8.4.4-37}$$

If n_Z is arbitrarily chosen to be 1, then

$$n_x = \frac{K - n_y Y1 - Z1}{X1} \qquad \text{Eq. 8.4.4-38}$$

$$n_y = \frac{K\left(1 - \frac{X2}{X1}\right) + \frac{X2}{X1}Z1 - Z2}{Y2 - \frac{X2}{X1}Y1} \qquad \text{Eq. 8.4.4-39}$$

where,

$$K = \frac{\left(\frac{X3}{X1}Z1 - Z3\right)\left(Y2 - \frac{X2}{X1}Y1\right) - \left(\frac{X2}{X1}Z1 - Z2\right)\left(Y3 - \frac{X3}{X1}Y1\right)}{\left(1 - \frac{X2}{X1}\right)\left(Y3 - \frac{X3}{X1}Y1\right) - \left(1 - \frac{X3}{X1}\right)\left(Y2 - \frac{X2}{X1}Y1\right)} \qquad \text{Eq. 8.4.4-40}$$

The coordinates X1, X2, X3, Y1, Y2, Y3, Z1, Z2, Z3 are as defined in Figure 8.4.4.20-5.

8.4.5. Solving Differential Equations

TRNSYS versions 11.1 and higher provide two methods for solving differential equations which may occur in component models: an approximate analytical solution using the subroutine Differential_Eqn (see Section 8.4.4.3), and a numerical solution using one of three numerical methods (see Manual 07-TRNEdit). The analytical method is recommended whenever practical it is possible to write the differential equation in the following general form:

$$\frac{dT}{dt} = aT + b \qquad \text{Eq. 8.4.5-1}$$

The numerical method generally requires shorter time steps and more computation for comparable accuracy and numerical stability. Users intending to utilize the Powell's method TRNSYS solver (SOLVER 1) should use the Differential_Eqn subroutine whenever possible.

Numerical solutions to differential equations, when required can be obtained through the DTDT and T arrays in the subroutine calling sequence. To use a very simple example, if the equation

$$\dot{Q}_i = C\frac{dT_1}{dt} \qquad \text{Eq. 8.4.5-2}$$

were part of the mathematical model and the DERIVATIVE of the temperature T_1 with respect to time t were to be integrated for a known heat flow rate $\dot{Q}$ and a constant value C, it would appear in the subroutine in a manner similar to that shown below.

```
.
.
DTDT(1) = QDOT / C
.
.
OUT(2) = T(1)
.
.
```

DTDT(1) is the differential dT_1/dt, and T(1) is the result of integrating DTDT(1). In this example T_1 is also desired to be the second OUTPUT of the sequentially ordered OUTPUT list. Note that the subscripts of OUT and T do not have to be the same, but that T(n) will always be the result of integrating DTDT(n).

A DERIVATIVES control statement followed by an initial values line must be included among the component control statements for any model which uses the DTDT array. The presence of a DERIVATIVES command signals TRNSYS to integrate derivatives placed in the DTDT array and to return results in the T array, repeating the procedure at each timestep until either the appropriate error TOLERANCE or the iteration LIMIT is reached. The initial values of the dependent variables (i.e., T(n)) are available to the component subroutine, if needed, through the T array at the first call of the simulation (i.e., when INF0(7) = - 1; see Section 8.4.3).

8.4.6. *Use of the Alternate Return (RETURN 1)*

TRNSYS checks to make sure that all routines required by the TRNSYS input file are included in the TRNSYS executable file by utilizing the FORTRAN alternate return. In previous TRNSYS versions run with compilers allowing unresolved externals, unlinked subroutines were not checked and were often the source of strange errors. The TRNSYS routines and all user-written routines that call other subroutines should include an alternate return. The following example should help the user understand the workings of the alternate return.

```
C* CALLING PROGRAM **
      CALL TYPE2(TIME,XIN,OUT,T,DTDT,PAR,INFO,ICNTRL,*101)
100           WRITE(*,*) ‘ERROR - TYPE 2 IS NOT LINKED !’
              CALL MYSTOP(1001)
              RETURN 1
101   CONTINUE
      .
      .
C* CALLED SUBROUTINE **
      SUBROUTINE TYPE2(TIME,XIN,OUT,T,DTDT,PAR,INFO,ICNTRL,*)
      .
      RETURN 1
```

The calling program will call the TYPE 2 subroutine and return to line 101 when the subroutine calculations are completed due to the RETURN 1 line in the TYPE 2 subroutine. If the RETURN 1 is missing from the TYPE 2 subroutine or the subroutine is not linked, the program will return to line 100 of the calling program and the error message will be flagged.

All user written routines should contain an asterisk (*) at the end of the subroutine designation and ‘RETURN 1’s every place a RETURN is desired. Failure to do so will cause the unlinked subroutine error message to be displayed and the program to terminate.

All user-written routines that call other subprograms should also include the alternate return feature to check for unlinked subprograms. To activate the alternate return:

1) Call the subprogram with an asterisk-line number combination at the end of the subprogram designation.

2) Specify the line number given in the step above as a CONTINUE line.

3) Between the subprogram call and the CONTINUE statement, place an error message and a:

   ```
   CALL MESSAGES(-1,’Error Message Text’,’fatal’,INFO(1),INFO(2))
   ```

   command - a call may instead be placed to the LINKCK subroutine (see Section 8.4.4.7).

4) Make sure all RETURN’s in the subprogram are replaced with a RETURN 1 statement. Refer to the example given or a FORTRAN manual for more details.

8.4.7. *Use of the ICNTRL Array*

The ICNTRL array, which was new to version 14, conveys to the component subroutines (TYPE1, TYPE2, etc.) and to the TRNSYS equation processor, the states of the controller variables. With the release of TRNSYS 14, a new control strategy was designed to eliminate the problems caused by previous TRNSYS controllers "sticking" in order to solve the system of equations (see Manual 07-TRNEdit). With TRNSYS 14, the TRNSYS executive program, rather than the component models, directly controlled discrete variables. At the start of each time step, the values of all discrete variables are known. TRNSYS 14 (and subsequent versions) force these values to remain at their current state until a converged solution to the equations is obtained or an iteration limit is encountered. During these calculations, the component models calculate the 'desired' value of the discrete variables but they do not actually change the settings. After a converged solution is obtained, or the maximum number of iterations is exceeded, TRNSYS compares the current discrete variable values with the 'desired' values. If they do not differ and a converged solution has been obtained, the calculations are completed for the time step. Otherwise, TRNSYS changes the values of the discrete variables to the 'desired' setting and repeats this process, taking care to not repeat the calculations with the same set of discrete variables used previously.

To incorporate the new control strategy into user-written component routines, the ICNTRL and INFO arrays must be set up correctly. INFO(11) should specify the number of discrete control variables in the component routine. ICNTRL(2*(i-1)+1) and ICNTRL(2i) will contain the control state at the previous iteration for controller i in the component model. (controlled by the TRNSYS processor) and the desired control state at this iteration respectively.

The ICNTRL array is an integer array and must therefore pass and receive only integer values. The assumption in TRNSYS is that an ICNTRL value of 1 implies the control signal was enabled while an ICNTRL value of 0 denotes a disabled control signal.

For example, a user wishes to describe an on/off controller with hysteresis effects (control signal depends on previous control signal). The component formulation would be of the following format:

```
UDB=PAR(1)    ! UPPER DEAD BAND TEMP. DIFFERENCE
LDB=PAR(1)    ! LOWER DEAD BAND TEMP. DIFFERENCE
.
INFO(11)=1    ! 1 CONTROL VARIABLE)
.
.
TH=XIN(1)     ! UPPER INPUT TEMPERATURE
TL=XIN(2)     ! LOWER INPUT TEMPERATURE
.
.
STATEOLD=ICNTRL(1)  ! CONTROL STATE AT PREVIOUS SOLUTION
STATENEW=STATEOLD

IF(STATEOLD.EQ.1) THEN
       IF((TH-TL).LT.LDB) STATENEW=0
ELSE
       IF(TH-TL).GT.UDB)) STATENEW=1
ENDIF
ICNTRL(2)=STATENEW
.
```

At every iteration, the value of ICNTRL(1) holds the controller state at the previous solution while ICNTRL(2) holds the desired controller state at this iteration. The TRNSYS processor will check to see if ICNTRL(1) is equal to ICNTRL(2) at convergence. If the values are not equal at convergence, ICNTRL(1) will be set to the desired controller state at convergence (ICNTRL(2)) and the process will be repeated.

IMPORTANT!! If a new component makes use of the ICNTRL array, then SOLVER 1 must be used in any TRNSYS input file that contains that component.

8.4.8. Converting Types from earlier TRNSYS Versions

As programming standards have evolved, so too have the conventions required by TRNSYS Types. The following sections provide the user with information on the steps required to convert Types written for earlier versions of TRNSYS to later versions.

8.4.8.1. Converting components from version 13.x to version 14.1

TRNSYS routines written for version 13.1 or earlier have to be modified for use with TRNSYS version 14.1 and higher. The steps are summarized below for convenience - refer to the sections listed for full details.

1) Change the TYPE specification from:

```
CALL TYPEn(TIME,XIN,OUT,T,DTDT,PAR,INFO)
```

   to:

```
CALL TYPEn(TIME,XIN,OUT,T,DTDT,PAR,INFO,ICNTRL,*)
```

2) Replace all RETURN statements in the subroutine with RETURN 1 Failure to replace all returns will result in an unlinked subroutine error. (see Section 8.4.6)

3) Change control algorithms to accommodate the Powell's Method control strategy (Optional) (see Section 8.4.7)

4) Change the XIN and OUT arrays to DOUBLE PRECISION variables. (see Section 8.3.2)

5) Change the INFO array dimension from 10 places to 15. (see Section 8.4.3)

6) Define the input and output variable types in the YCHECK and OCHECK arrays and call the RCHECK subroutine to perform input/output checking. (see Section 8.4.4.12)

7) All calls to the following routines should be modified to include the link checking option: DATA,ENCL,TABLE,INVERT,DINVRT,FIT,DFIT and PSYCH. (see Section 8.4.4.7)

8.4.8.2. Converting components from version 14.1 to version 14.2

TRNSYS routines written for version 14.1 have to be modified for use with TRNSYS 14.2 for Windows. The Windows operating system does not deal well with two common features of TRNSYS version 14.1 subroutines. The steps are summarized below for convenience.

1) Change any statements that contain a STOP statement from:

```
STOP
```

to the following if the Stop occurred in the primary Type subroutine:

```
CALL MYSTOP(1001)
RETURN 1
```

or to the following if the Stop occurred in a secondary subroutine:

```
CALL MYSTOP(1001)
RETURN
```

The number 1001 can be changed to another error number if so desired.

2) Windows does not allow any write-to-screen statements. Any WRITE or PRINT statements that print to the screen such as:

```
WRITE (*,*) ‘ There is an error in this type.’
```

need to be replaced with a statement that writes the text to the listing file.

```
WRITE(LUW,*) ‘There is an error in this type’
```

Note: In most cases, writing to the list file requires the inclusion of the common block that contains the value for the logical unit. Include the following line:

```
COMMON /LUNITS/ LUR, LUW, IFORM, LUK
```

8.4.8.3. Converting components from version 14.2 to version 15.x

There are no modifications necessary to a TRNSYS 14.2 Type in order for it to be used with TRNSYS version 15.x.

8.4.8.4. Converting components from version 15.x to version 16.x

The steps involved with converting TRNSYS components written for TRNSYS version 15.x to the TRNSYS 16.x coding standard are included in section 8.2.1.3 of this manual.

8.5. Instructions to rebuild TRNDll.dll and other dynamic link libraries

The following sections give detailed information on the compiler settings used to rebuild TRNDll.dll. They are for reference only. It is possible to recompile the main dynamic link library, TRNDll.dll, with two different Fortran compilers:

- Compaq Visual Fortran 6.6, and

- Intel Visual Fortran 9.1 with MS Visual Studio 2003 or 2005.

If you want to use one of these compilers, go to the directory %TRNSYS16%\Compilers and select the corresponding folder. These directories contain the workspace (for Compaq Visual Fortran (CVF)) and solution files (for Intel Visual Fortran (IVF)) with the appropriate settings. You do not have to modify any setting to recompile and debug TRNSYS if you use the project included in the distribution.

You can use the workspace (for CVF) or solution file (for IVF) to make modifications to the standard components and kernel subroutines, or to add you own components to the main library TRNDll.dll.

You may also use the workspace (for CVF) or solution file (for IVF) to build your own components into an individual dll. In this case, use the project ‘MyDll’ and add the source code for your own components.

8.5.1. Instructions to rebuild TRNDll.dll with Compaq Visual Fortran 6.6

To use the default workspace, launch CVF 6.6, go to "File/Open workspace" and select %TRNSYS16%\Compilers\Cvf66\Cvf66.dsw. You do not have to modify any setting to recompile and debug TRNSYS if you use the project included in the distribution. The instructions hereunder are for reference only. They explain how this workspace was created and give all modifications to default settings. You will also need to use the instructions here below if you want to use the **open-source version of Type 56** (see section 8.5.1.4).

Supported Compiler version: 6.6b (Note: there has been a 6.6c update but it appears to be unstable and it introduces some "features" that are incompatible with many existing programs. It was pulled back by HP-Compaq, then posted again. We advise against using it)

8.5.1.1. Cvf66 Workspace

In CVF, a Project is a set of files and settings that are used to build an application (.exe or .dll). A Workspace is a container for one or more projects. We will first create an empty Workspace:

- Go to File/New, select the "Workspace" Tab
- Select "Blank Workspace"

- Place in %TRNSYS16%\Compilers
- Workspace name: Cvf66

8.5.1.2. TRNDll project

The TRNSYS kernel and the standard components are compiled and linked into one DLL, TRNDll.dll. To create the project:

- Go to File/New, select the "Project" Tab
- Project type: select "Win32 Dynamic Link Library"
- Select "Add to current workspace"
- Project name: TRNDll
- The folder name should read %TRNSYS16%\Compilers\Cvf66\TRNDll
- Click OK
- Select "Empty DLL"
- Click Finish, then OK

8.5.1.3. Project settings

- Go to Projects/Settings
- In the left-hand window, select the "TRNDll" project
- In the "Settings for" drop-down list box showing "Win32 Debug", select "All configurations". We will now choose the settings that are common to the Debug and Release configurations.

There are numerous settings here controlled by a series of tabs at the top. Below are the important settings for each of these tabs. Each tab can have several "Categories" controlled by a drop-down list, so there are many settings involved. Only the settings that must be changed from their default values are mentioned.

Table 8.5.1.3-1: TRNDll Settings – CVF 6.6, All Configurations

Tab	Drop-down	Parameter	Value
Fortran	General	Predefined Preprocessor Symbols	TRNSYS_MULTI_DLL, TRNSYS_WIN32
	Compatibility	Powerstation 4.0 compatibility options	Check "Libraries" and "Other Run-time Behavior"
	External Procedures	Arguments Passing Convention	C, by Reference
		String Length Argument Passing	After All Args
	Floating Point	Floating point exception handling	0
		Extend precision of Single precision constants	Checked

In the "Settings for" drop-down list box, select "Win32 Debug". We will now choose the settings that only apply to the Debug configuration. Here again, only the settings that must be changed from their default values are mentioned.

Table 8.5.1.3-2: TRNDll Settings – CVF 6.6, Win32 Debug

Tab	Drop-down	Parameter	Value
Debug	General	Executable for debug session	..\..\..\Exe\TRNExe.exe
		Program arguments	You can optionally enter the name of the deck file that you wish to debug here, to bypass the file/open dialog in TRNExe.exe
Fortran	General	Predefined Preprocessor Symbols	Add: TRNSYS_DEBUG
		Generate Source Browse Information	
	Runtime	Generate traceback information	Checked
		Runtime Error checking	Check all boxes
Link	Input	Ignore Libraries	libcd.lib, msvcrtd.lib
Post-build step		Post-build description	Post-build: Copying Debug TRNDll.dll and TRNDll.lib to the Exe folder
		Post-build command(s)	copy "Debug\TRNDll.dll" "..\..\..\Exe\TRNDll.dll" copy "Debug\TRNDll.lib" "..\..\..\Exe\TRNDll.lib"

In the "Settings for" drop-down list box, select "Win32 Release". We will now choose the settings that only apply to the Release configuration. Here again, only the settings that must be changed from their default values are mentioned.

Table 8.5.1.3-3: TRNDll Settings – CVF 6.6, Win32 Release

Tab	Drop-down	Parameter	Value
Link	Input	Ignore Libraries	libcd.lib, msvcrtd.lib
Post-build step		Post-build description	Post-build: Copying Release TRNDll.dll and TRNDll.lib to the Exe folder
		Post-build command(s)	copy "Release\TRNDll.dll" "..\..\..\Exe\TRNDll.dll" copy "Release\TRNDll.lib" "..\..\..\Exe\TRNDll.lib"

Notes:

- The ignored libraries must change depending on the libraries used in the precompiled files. Currently Type56 is distributed in Release mode, i.e. linked against the release version of the standard libraries. Consequently the "debug" version must be ignored (libc***d***.lib and msvcrt***d***.lib). If a Debug version of Type56 was linked to the DLL, the "non-debug" version of default libraries should be ignored (libc.lib and msvcrt.lib).
- The post-build step makes sure the required files are copied to the Exe folder without adding unnecessary compiler files to that folder and making sure a copy of those files stays in the Debug or Release directory

8.5.1.4. Project files

On the left-hand window, go to "TRNDll files". You can delete the "Resource Files" and "Header Files" folders, which will not be used. Go to the "Source file" folder. Right-click on it and choose "New folder". Create the "Kernel" folder. Right click on this folder and choose "Add files to folder".

Browse to %TRNSYS16%\SourceCode\Kernel and select all .for and .f90 files. Click OK to add the files to the project.

Go back to the "Source file" folder. Right-click on it and choose "New folder". Create the "Types" folder. Right click on this folder and choose "Add files to folder". Browse to %TRNSYS16%\SourceCode\Types and select all .for and .f90 files (if any). Click OK to add the files to the project.

TYPE 56 MULTIZONE BUILDING

You can either use the Open source version (which does not include all Type56 features) OR use the precompiled .OBJ files. To use the precompiled files:

- Go to the "TRNDll Files" root folder and create the "Precompiled" folder.
- Select this new folder, right-click and choose "Add files to folder".
- Browse to %TRNSYS16%\SourceCode\Type56\ and choose a subdirectory according to your compiler (i.e. Cvf66 here). Select all .obj and .lib files (if any) and add them to the project.

To use the Open Source version:

- Go to "TRNDll Files" , "Source Files" and then "Types". Right-click, select "Add files to folder".
- Browse to Browse to %TRNSYS16%\SourceCode\Type56\OpenSource and select all .for and .f90 (if any) files.

8.5.1.5. Building the TRNSYS DLL (TRNDll.dll)

- Make sure the right project (TRNDll) is active in Project/Set active project, in case you have several projects.
- Select the appropriate configuration (Debug or Release) in Build/Set Active Configuration
- Go to Build/Rebuild all

Note: You need to do a Rebuild all before compiling any file that uses TRNSYS data modules. If you try to compile a routine using the data modules before performing a "Rebuild" you will get an message saying "error in opening module file" and the compilation will fail. The "Rebuild" operation compiles the modules and creates the module files (.mod) before compiling the files that require them.

8.5.1.6. Debugging TRNSYS

Debugging TRNSYS is easy using the Visual Studio: Just press F5 to start TRNSYS. If you left "Project Settings/Debug/General/Program Arguments" empty, an "open file" dialog box will show up and you can select the deck file from there.

Before starting the debugger, you can set/remove breakpoints anywhere in the code using the F12 key. You can then step through the code one instruction at a time using the F10 key (F11 to enter called subroutines or functions).

Hint: To force the compiler to stop when an error such as an "out of bounds" array occurs, you have to set it to stop on exceptions. To do this, launch the debugger (F5). While it is stopped in the File/open dialog box, come back to the debugger and go to Debug/Exceptions. Select all exceptions in the lower text field, select "Stop always" for the action and click on "change". CVF has a bug in the display but if you then click OK and come back at the same place you will see that your changes have been made. From now on, CVF will stop when such exceptions are generated and it will indicate the line that caused the problem.

8.5.1.7. Known issues with CVF 6.6

ERRORS ABOUT IFPORT.MOD, IFWIN.MOD, ETC. (FILE NOT FOUND)

This is a bug in CVF but it does not have any effect on TRNSYS. The "if*.mod" files are only used when compiling and linking with the Intel Visual Fortran, but CVF checks for their existence (in the wrong location) before even checking that it is going to use them. So you can safely ignore those messages.

CVF UNNECESSARILY RECOMPILING SOME ROUTINES WHEN REBUILDING TRNDLL.DLL

CVF has a known issue that causes it to recompile some files even though they have not been changed. This usually does not happen immediately but after using the project for some time you may notice that some files are recompiled without a valid reason. To fix that issue, you can try the following:

- In CVF, with the right project and configuration selected, do a Build/Clean
- In Windows explorer, browse to %TRNSYS16%\Compilers\Cvf66\TRNDll\Debug and delete all files that may have been left over. Do the same for the %TRNSYS16%\Compilers\Cvf66\TRNDll\Debug folder
- Then do a Build/Update all dependencies
- Then do a Rebuild all

This should solve the problem for a while. Moving the project from one location to another one or from one computer to another one is known to cause the problem to reappear.

HOW TO "FIX" A PROJECT THAT CRASHES WHEN YOU TRY DEBUGGING TRNSYS

Occasionally, some project files get corrupt and CVF crashes as soon as you try to debug TRNSYS. This is a known compiler issue. The following fix has been found to work in most cases:

- Delete ALL files in the %TRNSYS16%\Compilers\Cvf66 directory except for Cvf66.dsw and the subdirectories (only delete FILES, e.g. Cvf66.opt)
- Delete ALL files in the %TRNSYS16%\Compilers\Cvf66\TRNDll directory except for TRNDll.dsp and the subdirectories (only delete FILES, e.g. TRNDll.plg)
- Delete EVERYTHING in %TRNSYS16%\Compilers\Cvf66\TRNDll\Debug and %TRNSYS16%\Compilers\Cvf66\TRNDll\Release
- Then reopen Cvf66.dsw. Some settings like breakpoints will be lost but the important project settings are still there. The choice of debugging EXE is lost and you must re-enter ..\..\..\Exe\TRNExe.exe in Project settings / Debug / General / Executable for debug session.

If you are still unable to debug TRNSYS, please reinstall a fresh copy of the project from your original installation media or recreate the workspace after deleting all existing files, according to the instructions here above, after deleting all existing files in %TRNSYS16%\Compilers\Cvf66.

8.5.2. Instructions to add a new project to the CVF66 workspace

This section explains how to add a new project to the CVF66 workspace provided with TRNSYS 16 in order to create a library of user components.

> Note: You can use the "Export as Fortran" command in the TRNSYS Studio to generate a code template and a CVF project template with the settings that are listed in this section. The instructions here below are given for reference only and for users who want to control more advanced options. Please refer to the "Getting Started" TRNSYS Manual and to the TRNSYS Studio manual for more information on the "Export as Fortran" command.

8.5.2.1. Cvf66 workspace

The workspace file is %TRNSYS16%\Compilers\Cvf66\Cvf66.dsw. In Compaq Visual Fortran, a workspace is essentially a collection of Projects, each project being a set of files that allow you to build an executable program or a DLL when linked together. When you open the workspace, you will see a list of included projects on the left (by default only the TRNDll project is present).

8.5.2.2. New Project

- Go to File/New, select the "Project" Tab
- Project type: select "Win32 Dynamic Link Library"
- Select "Add to current workspace"
- Choose a Project name, e.g. MyDll.dll
- The folder name should read %TRNSYS16%\Compilers\Cvf66\MyDll
- Click OK
- Select "Empty DLL"
- Click Finish, then OK

This should add the MyDll project to the Workspace explorer (docked window on the left side of the screen by default).

8.5.2.3. Project settings

- Go to Projects/Settings and in the left-hand window, select the "MyDll" project (or right-click on the MyDll Project in the workspace explorer and select "Settings")
- In the "Settings for" drop-down list box showing "Win32 Debug", select "All configurations". We will now choose the settings that are common to the Debug and Release configurations.

There are numerous settings here controlled by a series of tabs at the top. Below are the important settings for each of these tabs. Each tab can have several "Categories" controlled by a drop-down list, so there are many settings involved. Only the settings that must be changed from their default values are mentioned.

Table 8.5.2.3-1: New DLL Settings – CVF 6.6, All Configurations

Tab	Drop-down	Parameter	Value
Fortran	Compatibility	Powerstation 4.0 compatibility options	Check "Libraries" and "Other Run-time Behavior"
	External Procedures	Arguments Passing Convention	C, by Reference
		String Length Argument Passing	After All Args
	Floating Point	Floating point exception handling	0
		Extend precision of Single precision constants	Checked

In the "Settings for" drop-down list box, select "Win32 Debug". We will now choose the settings that only apply to the Debug configuration. Here again, only the settings that must be changed from their default values are mentioned.

Table 8.5.2.3-2: New DLL Settings – CVF 6.6, Win32 Debug

Tab	Drop-down	Parameter	Value
General		Output files	..\..\..\Userlib\DebugDLLs
Debug	General	Executable for debug session	..\..\..\Exe\TRNExe.exe
		Program arguments	You can optionally enter the name of the deck file that you wish to debug here, to bypass the file/open dialog in TRNExe.exe
Fortran	General	Generate Source Browse Information	Checked
	Preprocessor	Include and Use Paths	..\TrnDll\Debug
	Runtime	Generate traceback information	Checked
		Runtime Error checking	Check all boxes
Link	Input	Ignore Libraries	See note hereunder
Post-build step		Post-build description	Post-build: Copying Debug .dll to the UserLib\DebugDLLs folder
		Post-build command(s)	copy "Debug\MyDll.dll" "..\..\..\UserLib\DebugDLLs\MyDll.dll"

In the "Settings for" drop-down list box, select "Win32 Release". We will now choose the settings that only apply to the Release configuration. Here again, only the settings that must be changed from their default values are mentioned.

Table 8.5.2.3-3: New DLL Settings – CVF 6.6, Win32 Release

Tab	Drop-down	Parameter	Value
General		Output files	..\\..\\..\Userlib\ReleaseDLLs
Fortran	Preprocessor	Include and Use Paths	..\TrnDll\Release
	Run time	Run time error checking	- Array and string bounds - Floating point underflow - Integer overflow - Power operations
Link	Input	Ignore Libraries	See note hereunder
Post-build step		Post-build description	Post-build: Copying Release .dll to the Exe folder
		Post-build command(s)	copy "Release\MyDll.dll" "..\\..\\..\UserLib\ReleaseDLLs\MyDll.dll"

Notes:

- The ignored libraries must be adapted to the precompiled object files or libraries that you link to the DLL. The linker will tell you which libraries cause conflicts. The most frequent ones are libc.lib and msvcrt.lib, as well as their debug versions, libcd.lib and msvcrtd.lib.

8.5.2.4. Project files

On the left-hand window, go to "MyDll files". You can delete the "Resource Files" and "Header Files" folders, which will not be used.

Go to the "Source file" folder. Right click on this folder and choose "Add files to folder". Browse to your source files (Types and possibly the utility routines they require) and add them.

Add the "TRNDll.lib" file that you will find in %TRNSYS16%\Exe to the project (e.g. in a "Libraries" folder that you create). This will allow your DLL to use all the symbols exported by TRNSYS.

8.5.2.5. Building the new DLL

- Make sure the right project (MyDll) is active in Project/Set active project, in case you have several projects.
- Select the appropriate configuration (Debug or Release) in Build/Set Active Configuration
- Go to Build/Rebuild all

MyDll.dll should be created in %TRNSYS16%\UserLib. You can debug components in this DLL by making the project active in Project/Set active project and launching the Exe used for Debugging (TRNExe.exe) by pressing F5.

8.5.2.6. *Troubleshooting multiple DLL's*

SHARING LOGICAL UNITS (FILES) BETWEEN DLL'S

Typical symptoms:

You assign a file in the TRNSYS input file and that file seems to be inaccessible to user-Types compiled in an external DLL.

- An "Inquire" statement tells you that the Logical Unit is not associated to a file
- Attempts to write to / read from the file create "fort.nnn" (where nnn is the logical unit)

Solution:

If a TRNSYS Type uses a file that is opened and closed by the kernel (i.e. a file managed through an "ASSIGN" statement in the deck, or an "External File" in a Studio proforma), the logical unit is associated to the actual file in TRNDll. If a Type that accesses the file is compiled into a separate DLL in UserLib:

- **Both DLL's must be linked to the DLL form of the Fortran run-time libraries**. Use "/libs:dll" in the command line, or set "Project Settings\Fortran\Libraries" to "Single-threaded DLL" in the Visual Studio ("Debug Single-threaded DLL" for debug configuration).
- Furthermore, **all DLL's must be compiled in the same mode (Debug or Release)**. Note that by default TRNSYS is shipped with a Release TRNDll.dll, but a Debug DLL is available in "\Compilers\Cvf66\TRNDll\Debug".

8.5.3. Instructions to rebuild TRNDll.dll with Intel Visual Fortran 9.1 and MS Visual Studio 2003

The following sections give detailed information on the compiler settings used to rebuild TRNDll.dll. They are for reference only. To use the "solution" (Visual Studio .Net equivalent of the Visual Studio 6.0 "workpace"), launch Microsoft Visual Studio .Net, go to "File/Open Solution" and select the followinf file: %TRNSYS16%\Compilers\Ivf91-Mvs2003\Ivf91-Mvs2003.sln. You do not have to modify any setting to recompile and debug TRNSYS if you use the project included in the distribution. The instructions hereunder are for reference only. They explain how this solution was created and give all modifications to default settings. You will also need to use the instructions here below if you want to use the **open-source version of Type 56** (see section 8.5.3.4).

Supported Compiler version: 8.1.029 or above. The latest tested version is identified by the following Package ID: w_fc_p_8.1.029. You can find out the Package ID you are using by typing "ifort" at the command prompt (you may have to add the path to ifort.exe to your PATH environmental variable or to type the full path at the command prompt, by default: " C:\Program Files\Intel\Fortran\compiler80\IA32\BIN\ifort.exe".

Note: The solution distributed with TRNSYS 16 is created using Microsoft Visual Studio 2003. Unfortunately, Microsoft changed the solution file format between Visual Studio .Net 2002 and 2003, so the solution is not usable using MVS .Net 2002. However, the distributed projects for Intel Visual Fortran can easily be imported in a newly created solution if you use MVS .Net 2002. All the important settings are in those projects, not in the solution.

8.5.3.1. Ivf91-Mvs2003 Solution

In Intel Visual Fortran, a project is set of files and settings that are used to build an application (.exe or .dll). A Solution is a container for multiple projects. The solution itself can be built, which builds all of the contained projects in a specified order (which can be adjusted by the user).

- Go to File/New, select the "Blank Solution" option
- In "name", type in "Ivf91-Mvs2003" (without the quotes)
- For "Location", browse to %TRNSYS16%\Compilers
- Click OK. This will create the file %TRNSYS16%\Compilers\Ivf91-Mvs2003\Ivf91-Mvs2003.sln

8.5.3.2. TRNDll project

The TRNSYS kernel and the standard components are compiled and linked into one DLL, TRNDll.dll. To create the project:

- Go to File / New / Project
  - Under "Project Types", select "Intel ® Fortran Project"
  - Under "Templates", select "Dynamic-link Library"
  - Type in "TRNDll" for the name
  - Select the radio button ‘Add to Solution’.
  - Click OK.

- This will open the "Fortran Dynamic Link Library Wizard" window.
  - Select "DLL Settings" under "Overview", in the left frame, and choose "Empty Project" as the DLL Type.
  - Click "Finish".
- This will create the following Project file with default settings: %TRNSYS16%\Compilers\Ivf91-Mvs2003\TRNDll\TRNDll.vfproj

8.5.3.3. *Project settings*

In the Solution Explorer (right-hand window by default), right-click on the TRNDll project and select "Properties".

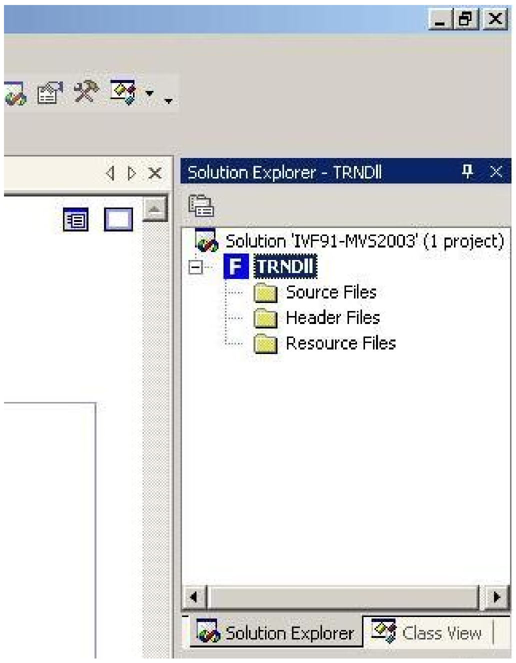
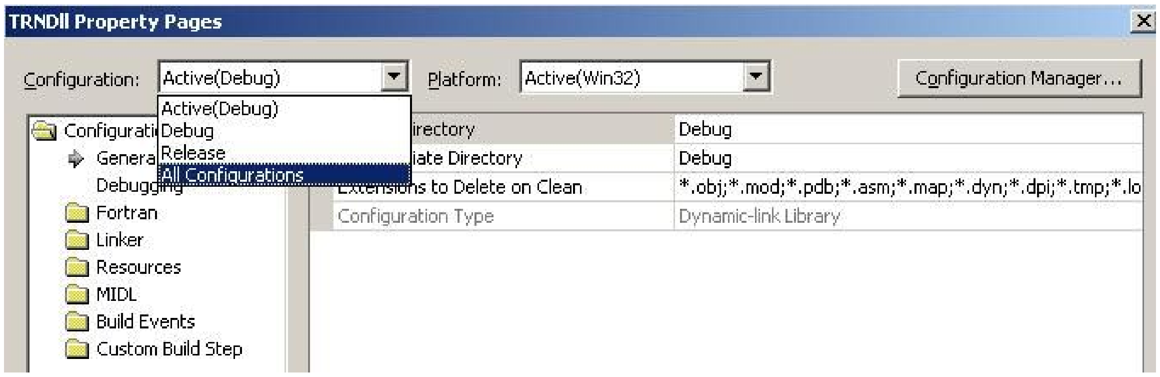


Figure 8.5.3.3-1: Solution explorer, with TRNDll project.

In the window ‘TRNDll Property Pages’, go to the pull down menu ‘Configuration’; select "All configurations". We will now choose the settings that are common to the Debug and Release configurations.



Configuration: Active(Debug)
Platform: Active(Win32)

irectory	Debug
iate Directory	Debug
Extensions to Delete on Clean	*.obj;*.mod;*.pdb;*.asm;*.map;*.dyn;*.dpi;*.tmp;*.lo
Configuration Type	Dynamic-link Library

Figure 8.5.3.3-2: TRNDll properties window. Select 'All configurations'.

There are numerous settings here controlled by a tree structure on the left. Below are the important settings for each of these tabs. Each tree branch can have several "Categories", so there are many settings involved. Only the settings that must be changed from their default values are mentioned.

Table 8.5.3.3-1: TRNDll Settings – IVF 9.1, All Configurations

Branch	**Category**	**Parameter**	**Value**
Fortran	General	Preprocessor Definitions	TRNSYS_MULTI_DLL; TRNSYS_WIN32; __INTEL_COMPILER
	Data	Local Variable Storage	All variables SAVE
		Initialize Local Saved Scalars to Zero	Yes
	Floating Point	Extend precision of Single precision constants	Yes
	External procedures	Calling convention	C,REFERENCE
		Name case interpretation	Upper Case
Linker	Command line	/ignore:4049,4217	

In the "configuration" drop-down list box, select "Debug". We will now choose the settings that only apply to the Debug configuration. Here again, only the settings that must be changed from their default values are mentioned.

Table 8.5.3.3-2: TRNDll Settings – IVF 9.1, Win32 Debug

Branch	**Category**	**Parameter**	**Value**
Debugging		Command	..\..\..\Exe\TRNExe.exe
		Command arguments	You can optionally enter the name of the deck file that you wish to debug here, to bypass the file/open dialog in TRNExe.exe
Fortran	General	Preprocessor Definitions	Add 'TRNSYS_DEBUG'
	Libraries	Runtime library	Debug Single-threaded DLL
	Run-time	Runtime Error Checking	All
Build events	Post-build Event	Command Line	copy "Debug\TRNDll.dll" "..\..\..\Exe\TRNDll.dll" copy "Debug\TRNDll.lib" "..\..\..\Exe\TRNDll.lib"
		Description	Post-build: Copying TRNDll.dll and TRNDll.lib to the Exe folder

In the "configuration" drop-down list box, select "Release". We will now choose the settings that only apply to the Release configuration. Here again, only the settings that must be changed from their default values are mentioned.

Table 8.5.3.3-3: TRNDll Settings – IVF 9.1, Win32 Release

Branch	Category	Parameter	Value
Fortran	Libraries	Runtime library	Single-threaded DLL
Build events	Post-build Event	Command Line	copy "Release\TRNDll.dll" "..\..\..\Exe\TRNDll.dll" copy "Release\TRNDll.lib" "..\..\..\Exe\TRNDll.lib"
		Description	Copying TRNDll.dll and TRNDll.lib to the Exe folder
Linker	Optimization	References	Keep unreferenced data

Notes:

- The post-build step makes sure that the required files are copied to the Exe folder without adding unnecessary compiler files to that folder and making sure a copy of those files stays in the Debug or Release directory

8.5.3.4. Project files

- In the Solution Explorer, you can optionally remove the "Header Files" and "Resource Files" folders under "TRNDll", since they are not used (right-click on the folder and select "Remove").
- Add Kernel routines:
  - Right-Click on "Source Files" under the "TRNDll" project.
  - Select "Add", then "New Folder".
  - Create a folder and rename it to "Kernel".
  - Right-click on that newly-created folder and select "Add", then "Add Existing Item".
  - Browse to %TRNSYS16%\SourceCode\Kernel and select all *.for and *.f90 files.
  - Click OK to add the files to the project.
- Add Standard Types:
  - Go back to the "Source file" folder.
  - Right-click on it and choose "New folder".
  - Create the "Types" folder.
  - Right-click on this folder and choose "Add", then "Add Existing Item".
  - Browse to %TRNSYS16%\SourceCode\Types and select all .for and .f90 files (if any).
  - Click OK to add the files to the project.

TYPE 56 MULTIZONE BUILDING

You can either use the Open source version (which does not include all Type56 features) OR use the precompiled .OBJ files. To use the precompiled files:

- Go to the "TRNDll / Source Files" root folder and create the "Precompiled" folder.
- Select that folder, right-click and choose "Add", then "Add Existing Item".

- Browse to %TRNSYS16%\SourceCode\Type56\ and choose a subdirectory according to your compiler (i.e. Ivf91 here). Select all .obj and .lib files (if any) and add them to the project (Tip: change the filter in "File Type" to "all files") .

To use the Open Source version:

- Go to "TRNDll", "Source Files" and then "Types". Right-click, select "Add", then "Add Existing Item".
- Browse to Browse to %TRNSYS16%\SourceCode\Type56\OpenSource and select all *.for and *.f90 (if any) files.

8.5.3.5. Building TRNDll.dll

- Make sure the right project (TRNDll) is active (its name should be in bold characters in the Solution Explorer). If not, make it active (Select, right-click, choose "Set as StartUp Project").
- Select the appropriate configuration (Debug or Release) in Build/Configuration Manager
- Go to Build/Rebuild Solution

Note: You need to do a Rebuild before compiling any file that uses TRNSYS data modules. If you try to compile a routine using the data modules before performing a "Rebuild" you will get an message saying "error in opening module file" and the compilation will fail. The "Rebuild" operation compiles the modules and creates the module files (.mod) before compiling the files that require them.

8.5.4. Instructions to add a new project to the Ivf91-Mvs2003 Solution

This section explains how to add a new project to the IVF90 Solution provided with TRNSYS 16 in order to create a library of user components.

Note: You can use the "Export as Fortran" command in the TRNSYS Studio to generate a code template and a CVF project template with the settings that are listed in this section. The project can then be opened in MS Visual Studio .Net and automatically updated to IVF 90

The instructions here below are given for reference only and for users who want to control more advanced options. Please refer to the "Getting Started" TRNSYS Manual and to the TRNSYS Studio manual for more information on the "Export as Fortran" command.

8.5.4.1. Ivf91-Mvs2003 Solution

The solution is %TRNSYS16%\Compilers\Ivf91-Mvs2003\Ivf91-Mvs2003.sln

In Intel Visual Fortran, a project is set of files and settings that are used to build an application (.exe or .dll). A Solution is a container for multiple projects. The solution itself can be built, which builds all of the contained projects in a specified order (which can be adjusted by the user).

8.5.4.2. New Project

- Go to File / New / Project
  - Under "Project Types", select "Intel ® Fortran Project"
  - Under "Templates", select "Dynamic-link Library"
  - Type in "MyDll" for the name. You may select a different name for your application, or rename it at the end of these instructions.
  - Check the "Add to solution" radio button. The Location will be adapted automatically and should say: %TRNSYS16%\Compilers\Ivf91-Mvs2003
  - Click OK.
- This will open the "Fortran Dynamic Link Library Wizard" window.
  - Select "DLL Settings" under "Overview", in the left frame, and choose "Empty Project" as the DLL Type.
  - Click "Finish".
- This will create the following Project file with default settings: %TRNSYS16%\Compilers\Ivf90-Mvs2003\MyDll\MyDll.vfproj

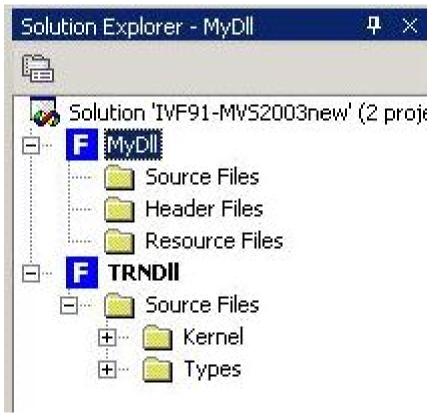


Figure 8.5.4.2-1: Solution explorer, with projects ‘TRNDll’ and ‘MyDll’.

8.5.4.3. Project settings

In the Solution Explorer (right-hand window by default), right-click on the project ‘MyDll’ and select "Properties".

In the "configuration" drop-down list box showing "Active (Debug)", select "All configurations". We will now choose the settings that are common to the Debug and Release configurations. There are numerous settings here controlled by a tree structure on the left. Below are the important settings for each of these tabs. Each tree branch can have several "Categories", so there are many settings involved. Only the settings that must be changed from their default values are mentioned.

Table 8.5.4.3-1: New Dll Settings – IVF 9.1, All Configurations

Branch	Category	Parameter	Value
Fortran	Data	Local Variable Storage	All variables SAVE
		Initialize Local Saved Scalars to Zero	Yes
	Floating Point	Extend precision of Single precision constants	Yes
	External procedures	Calling convention	C, REFERENCE
		Name case interpretation	Upper Case

In the "configuration" drop-down list box, select "Debug". We will now choose the settings that only apply to the Debug configuration. Here again, only the settings that must be changed from their default values are mentioned.

Table 8.5.4.3-2: New Dll Settings – IVF 9.0, Win32 Debug

Branch	Category	Parameter	Value
Debugging		Command	..\..\..\Exe\TRNExe.exe
		Command arguments	You can optionally enter the name of the deck file that you wish to debug here, to bypass the file/open dialog in TRNExe.exe
Fortran	Preprocessor	Additional Include Directories	..\TRNDll\Debug
	Libraries	Runtime library	Debug Single-threaded DLL

	Run-time	Runtime Error Checking	All
Build events	Post-build Event	Command Line	copy "Debug\MyDll.dll" "..\..\..\UserLib\DebugDLLs\MyDll.dll" [♣]
		Description	Post-build: Copying .dll to the UserLib folder

In the "configuration" drop-down list box, select "Release". We will now choose the settings that only apply to the Release configuration. Here again, only the settings that must be changed from their default values are mentioned.

Table 8.5.4.3-3: TRNDll Settings – IVF 9.0, Win32 Release

Branch	**Category**	**Parameter**	**Value**
Fortran	Preprocessor	Additional Include Directories	..\TRNDll\Release
	Libraries	Runtime library	Single-threaded DLL
Build events	Post-build Event	Command Line	copy "Release\MyDll.dll" "..\..\..\UserLib\ReleaseDLLs\MyDll.dll" [♣]
		Description	Copying .dll to the UserLib folder

Notes:

- The post-build step makes sure the required files are copied to the UserLib folder without adding unnecessary compiler files to that folder and making sure a copy of those files stays in the Debug or Release directory

8.5.4.4. Project files

- In the Solution Explorer, you can optionally remove the "Header Files" and "Resource Files" folders under "MyDll", since they are not used (right-click on the folder and select "Remove").
  - Add the source file(s) for your Type(s) and possibly the routines, modules, etc. that they require
  - Create a folder called "Libraries" under MyDll and add the TRNDll.lib file that is found under the folder %TRNSYS16%\Compilers\IVF91-MVS2003\TRDll\Debug. This will allow your DLL to use all the symbols exported by TRNSYS.

8.5.4.5. Building MyDLL

- Make sure the right project (MyDll) is active (its name should be in bold characters in the Solution Explorer). If not, make it active (Select, right-click, choose "Set as StartUp Project").
- Select the appropriate configuration (Debug or Release) in Build/Configuration Manager
- Go to Build/Rebuild Solution

These instructions will create a DLL of your files, and will place a copy of the 'MyDll.dll' into the directory %TRNSYS16%\UserLib\ReleaseDLLs or %TRNSYS16%\UserLib\DebugDLLs. You may want to rename the file 'MyDll.dll' for another meaningful name, such as 'Typexxx.dll'.

[♣] Until TRNSYS16.00.0038 this directory was "..\..\..\UserLib\MyDll.dll"
[♣] Until TRNSYS16.00.0038 this directory was "..\..\..\UserLib\MyDll.dll"

8.5.5. Instructions to rebuild TRNDll.dll with Intel Visual Fortran 9.1 and MS Visual Studio 2005

The instructions for rebuilding the main TRNDLL.dll and to add new libraries with Intel Visual Fortran 9.1 in MS Visual Studio 2005 are the same as those explained in the previous sections.

In the directory %TRNSYS16%Compilers you will find a solution for Intel Visual Fortran 9.1 and MS Visual Studio 2005.

8.6. References

Engineering Equation Solver, F-Chart Software, 4406 Fox Bluff Road, Middleton WI, 1994. www.fchart.com

ASHRAE. 2001. *Handbook of Fundamentals.* American Society of Heating Refrigeration, and Air Conditioning Engineers, Atlanta, Ga.