PANACEA DOCUMENT

Stewart A. Brown

Table of Contents

The PANACEA C API

The PANACEA FORTRAN API

Acknowledgments

As with all such things, PANACEA was not created in a vacuum and owes a debt of thanks to many people who contributed in many different ways to its evolution. There are many more than can be mentioned in this short space, and the ones who are singled out are not mentioned in any particular order.

Bill Eme and Tom Kelleher used the code, looked at the code, and made many excellent suggestions about the functionality. Rex Evans worked on forerunners to PANACEA over the years and many of the ideas here had their origins in his efforts. Lee Taylor similarly has had lots of experience with simulation code systems and has had a large impact on the workings of PANACEA. John Ambrosiano pushed hard for PANACHE and has graciously tested it. David Hardin has helped greatly in clarifying certain user requirements and in verifying their implementation.

Jan Moura and Jeff Long organized a group to look into the requirements for time history data management and their efforts are to be found here. Scott Futral and Al Miller contributed some code and examples.

Thanks also go to Ron Mayle and Howard Scott who have put up with PANACEA from its

conception as often frustrated users.

Disclaimer

Part of this work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Introduction

A numerical simulation code is fundamentally a device to solve an initial value problem. Very often these programs are written starting with the routines that solve some system of equations. Unless care is taken to formulate these routines with regard to a host of issues (e.g., data flow), it can be very awkward to set up a problem or to look at the answers that the code returns. Furthermore, it can be nearly impossible to take advantage of some of the excellent work that various groups have done (e.g., equation solver packages) because the data flow and structuring issues have not been given sufficient attention in the program design stages.

One way to avoid some of these difficulties is to write the data flow controls, the I/O mechanisms, and the database manager first. With these parts in place, there is a great deal of guidance for building the parts of a simulation code that deal with the specific set of equations to be solved; thus initial value data is fed in and the solution is written out clearly and conveniently. Furthermore, this approach provides a basis for defining some conventions to allow the sharing of simulation modules. It also helps isolate the machine dependencies of a code in a very few places, which greatly enhances the portability of the code.

PANACEA is a library of routines built to address these issues. In essence, PANACEA attempts to embody all of the functionality of a numerical simulation code, which is generic to all simulation codes. It accomplishes this by implementing a model of the simulation as a system of services (routines), which are bound together with the simulation specific routines provided by the code developer. One useful result of this approach is that it effectively imposes a discipline for the development of simulation codes that promote shareability, portability, and inter-operability.

Throughout this manual examples will be couched in terms of a sample PANACEA code system called ABC. Like most PANACEA codes, ABC consists of three executable codes: A, the initial value problem generator; B, the main simulation code; and C, the post processor code for ABC. ABC is a simple 2d hydrodynamics code with three packages: a global package for mesh handling and various "global" facilities; a hydrodynamics package;

and a materials package. Many of the code fragments given, although appearing complete, are edited down to draw attention to essential points and leave out details which are not pertinent to the discussion of PANACEA.

The PANACEA Model

This section discusses the motivation for and offers an explanation of the PANACEA model. It summarizes the services that PANACEA provides and examines some related code issues.

Sequence of States

An initial value problem can be viewed as a machine (some system of equations) and an initial state. The initial state is evolved to a final state by the machine. When the system of equations is solved numerically, the evolution of states by the machine is represented as a discrete sequence of states.

In this view, there are three broad phases to a numerical simulation. First, the initial state must be generated. Second, the numerical machine must evolve each state into its successor state until this cycle is terminated according to some criterion. Last, some or all of the information that represents the states of the system must be organized for visualization. The last two phases often occur simultaneously. For example, in a time plot of some physical variable in the simulation, the information for the plot must be gathered up in the course of the entire simulation and viewed at the end.

PANACEA formalizes these three phases by addressing entire sets of routines to supporting the processes involved: problem generation, simulation, and post-processing for visualization. These phases can be accomplished in three separate codes or can be integrated into a single code. PANACEA attempts to make a minimum number of assumptions about the simulation being carried out. The assumptions that are made pertain to the model of a simulation as outlined above.

Simulation Packages

Many of the most complex simulation programs are broken down into parts that solve some subset of the system of equations in the simulation. For example, some codes have a thermal conduction package that is a separately callable module. The reasons are fourfold: ease of control (packages can be switched on and off); simplified design and maintenance; multiple packages solving the same subset of equations (algorithmic testing or problem domain differences); or code recycling.

Ideally, packages built for one code could be loaded into another code with few or no changes. This shareability will require some standardization of interfaces, database access, and so forth. PANACEA provides a framework upon which to build such a standardization. It is especially well suited to this task in light of its attempt to make few assumptions about how a code is organized or what it does.

PANACEA has the facilities to manage a collection of simulation packages. The code developer defines each package to PANACEA (according to the criteria of the PANACEA model of simulations), which amounts to providing a set of functions which carry out simulation specific operations. Figure 1 illustrates how some basic functions provided by the code developer are bound by PANACEA into a simulation code. With the packages defined, PANACEA can manipulate them as abstract entities and perform such generic operations as cycling through the packages to evolve the state of the simulation.

Defining packages for a PANACEA simulation code

```
def_system()
            {PA_run_time_package("global", NULL, NULL, def_global,
                                 cont_global, init_vars, global_entry,
                                 B_build_pseudo_mapping, NULL, NULL);
             PA_run_time_package("hydrodynamics", NULL, NULL,
                                 def_hydro, cont_hydro, init_hydro,
                                 hydro, B_build_pseudo_mapping,
                                 NULL, NULL);
             PA_run_time_package("materials", NULL, NULL,
                                 def_mat, cont_mat, mat_init,
                                 mat, B_build_pseudo_mapping,
                                 NULL, NULL);
             return;}
```

Many (but not all) simulation codes require one or more computational meshes. Every simulation code does require some controlling information (e.g., when to stop) to be managed. PANACEA acknowledges this problem by requiring at least one package be provided. It is given a special status and a special name, the global package. It is recommended (but not required) that the mesh generation and handling routines go in the global package.

To help realize the goal of shareability, the PANACEA package is defined as containing all of the routines needed to use the package in each of the phases: generation, simulation, and post-processing. In that way, the recipient of a PANACEA package should not have to add or change as much coding. For example, an equation-of-state (EOS) package might have some special information, which the generator must process for the main simulation. A PANACEA version of such an EOS package would include the routines to process that information for the generator.

In practice, PANACEA packages can be thought of and realized as libraries of routines that are loaded together with a very simple driver module to produce the entire simulation code system (see Figure 2). If the PANACEA ideal of shareability were perfect, one simulation code could add the invocation of a package from a second code and load the package library to add the new package to the original simulation code system.

A simple driver for the code whose packages are defined in Figure 1.

```
/* B.C - the main simulation code of the ABC system */

       #include "b.h"

       /* MAIN - read the restart dump and run the physics loop */

       main(c, v)
          int c;
          char **v;
          {char *fname;

       /* define the code system by setting up the packages */
           def_system();
           PA_build_domain_hook  = B_build_domain;
           PA_build_mapping_hook = B_build_mapping;

       /* read the restart file */
           if (fname != NULL)
              {PA_rd_restart(fname, NONE);
               name[2] = SC_strsave(fname);};

       /* perform initialization */
           PA_init_system(param[1], (param[3] - param[2])*param[4],
                          swtch[3], name[3], name[4], name[5]);
```

```
/* run the simulation */
    PA_simulate(param[1], swtch[3], N_zones, param[2], param[3],
                param[4], param[5], param[6], param[7],
                name[2], name[3], name[4], name[5]);

/* write a restart dump before ending */
    PA_wr_restart(name[2]);

    exit(0);}
```

Simulation Variables

The complement to the simulation package is the simulation variable. Each package has three categories of variables: input variables, which are provided to the package by other packages; internal variables, which are neither imported into the package nor exported from it; and output variables, which are provided by the package for use by other packages.

PANACEA manages a database of variables to which each package may contribute. Each variable is "owned" by some package and each package can gain access to any variable in the database. Access control is especially useful to help prevent some of the more obscure problems that can arise with unregulated access to simulation variables.

A very important aspect of the variables in the database is to have a generic mechanism for specifying output requests. PANACEA can handle "most" requests in a generic fashion, thus freeing the application developer from having to worry about "most" output requests (plots, edits, dumps, etc.).

Variables in PANACEA are carefully defined, including such information as name, type, array dimensions, units, and various categories of scope. This imposes an important discipline on the developer and permits PANACEA to provide some additional services relating to unit conversion, and subset extraction for I/O (see Figure 3).

The database manager is the main mechanism for regulating data flow through a PANACEA code system. There is a practical problem inherited from the C programming language, in which PANACEA is implemented, in that there are no separate name spaces and hence global variables can (and sometimes do) make variables available where not actually intended or desired. A solution to this problem will come in later versions of PANACEA.

Physical Units

PANACEA maintains three systems of units: external, internal, and cgs. The end user specifies input quantities in external units and output quantities are given in external units.

Internal units are those in which the simulation routines deal. This was done to give code developers the opportunity to debug in units that make sense to them while letting the end users deal in a system of units that is natural to the problems being run. A "complete" set of fundamental constants and many derived physical constants are maintained by PANACEA in cgs units originally. PANACEA handles all data conversions in an efficient and transparent way, though each variable must have its units defined so that conversions can be done. The conversions cannot work if there are any numerical constants that have physical units in a PANACEA code. For example, writing 2.99e10 for the speed of light in an expression is illegal (although PANACEA does nothing to enforce it). The PANACEA provided constant, c, must be used instead.

Defining variable for a PANACEA database. Note that this is one of the functions installed as part of the definition of the hydrodynamics package in Figure 1

```
/* DEF_HYDRO - define the hydrodynamics package data base */


       def_hydro(pck)

          PA_package *pck;

          {int *P_zones;

           P_zones = &swtch[4];


       /* RESTART VARIABLES */

           PA_def_var("n", SC_DOUBLE_S, NULL, NULL,

                      SCOPE, RESTART, CLASS, REQU, CENTER, Z_CENT, ATTRIBUT

                      P_zones, DIMENSION, PER, CC, UNITS);

           PA_def_var("P", SC_DOUBLE_S, NULL, NULL,

                      SCOPE, RESTART, CLASS, REQU, CENTER, Z_CENT, ATTRIBUT

                      P_zones, DIMENSION, ERG, PER, CC, UNITS);

           PA_def_var("rho", SC_DOUBLE_S, NULL, NULL,

                      SCOPE, RESTART, CLASS, REQU, CENTER, Z_CENT, ATTRIBUT

                      P_zones, DIMENSION, G, PER, CC, UNITS);


       /* RUNTIME VARIABLES */

           PA_def_var("Rpdv", SC_DOUBLE_S, NULL, NULL,

                      CENTER, Z_CENT, ATTRIBUTE,
```

```
                P_zones, DIMENSION, EV, PER, SEC, UNITS);
    PA_def_var("Rqdv", SC_DOUBLE_S, NULL, NULL,
                CENTER, Z_CENT, ATTRIBUTE,
                P_zones, DIMENSION, EV, PER, SEC, UNITS);
    PA_def_var("vol", SC_DOUBLE_S, NULL, NULL,
                CENTER, Z_CENT, ATTRIBUTE,
                P_zones, DIMENSION, CC, UNITS);
    PA_def_var("mass-z", SC_DOUBLE_S, NULL, NULL,
                CENTER, Z_CENT, ATTRIBUTE,
                P_zones, DIMENSION, G, UNITS);

/* EDIT VARIABLES */
    PA_def_var("Q", SC_DOUBLE_S, NULL, NULL,
                SCOPE, EDIT, CLASS, REQU, CENTER, Z_CENT, ATTRIBUTE,
                P_zones, DIMENSION, ERG, PER, CC, UNITS);

    return;}
```

This efficiency offers three benefits. First, by requiring constants to be expressed in terms of physical constants and dimensionless numbers, a code system avoids certain arithmetic errors and is more self-documenting. Second, an extra level of consistency is built into simulation codes. I have found bugs in simulation packages by running a problem in two different systems of internal units. Such simulations should produce the same result, but various inconsistencies between packages can easily result in different answers. The third benefit is that without such a mechanism simulation packages in general would not be sharable! By carrying their system of units around in physical terms, packages can be moved around among simulations codes without regard to the way the original developer thought about units.

Data Flow

With the three phase model in mind, the PANACEA model manages the data flow throughout a simulation as follows: the generator reads in ASCII and binary information to define the initial state, which is written as a binary file; the simulation phase code reads state files and source files, communicates with itself by writing state files, and communicates with the post-processor by writing dumps targeted for it; and the post-processor reads only the post-processor dumps and writes out files for target visualization systems.

To maintain portability, PANACEA uses PDBLib to read and write portable binary files. The choice of PDBLib is motivated by its efficiency and its ability to handle structured data including pointers. In fact, PANACEA uses parts of PDBLib directly to manage its database. In this way, the developer can work with relatively arbitrary data structures without having to worry about reading and writing them out to the state files. The fact that the same tool is used for all binary files coupled with the fact that PDB files are self-describing gives the code developer the opportunity to make the greatest use of the data in the various files.

Generation of Initial State

The procedure for using PANACEA to build a program to generate initial states for a simulation is described in this section. For sake of example, the generator program will be referred to as A. As with all PANACEA codes the simulation packages are the focus of attention.

PANACEA uses a very simple mechanism for processing user commands. Each line of input in the user written file that describes the simulation problem is read into a buffer; the first token is stripped off and used in a hash table to dispatch to a routine that the code developer has provided to process the remaining tokens in the input line and to take appropriate action. PANACEA provides several functions to get the tokens and, if necessary, to translate them to numerical data types.

Given this model, it is clear that the developer must write a collection of routines to handle the various commands for A. Very often in non-PANACEA codes this is done by having a routine with a potentially enormous if...elseif...elseif...endif construct. The advantage of PANACEA's hash driven scheme is that each command is processed separately and with equally quick access. This procedure is done on a package-by-package basis. In this way, the routines that generate information for a package are associated with the package.

Since PANACEA manages a database of simulation variables, the variables for each package must be defined to the database. PANACEA provides a routine for this purpose. The code developer must completely define variables for each package. These definitions are put into a single routine, which is one of the defining routines of a PANACEA package. The information required about a variable is shown below:

- A name by which it is installed in the database (a hash table).
- A pointer to the number of elements in the variable.
- A list of pointers to dimensions defining the shape of variable (1-D array, 2-D array, etc.).
- A designation specifying the scope of the variable.
- Another designation for the class of the variable.
- One for the centering of the variable with respect to any coordinate mesh used in the simulation.

- A list of physical units (see Figure 3).

The scope of a variable addresses three issues: (1) If it is necessary to the definition of the simulation state, in which case it is written to the state file when such a dump is done; (2) If the variable can be computed from the state but should otherwise remain present for the entire simulation; or (3) If the variable should only exist long enough to be edited and then have its space reclaimed.

The class of a variable refers to whether it is required to be in the state file, whether it is optional in the state file, or whether it should be read from the state file only when specifically requested. The first two classes are read in whenever a state file is read. The last one only brings in the data when specifically accessed with PA_CONNECT.

Pointers to the size (number of elements and shape) information are necessary because, at the time the variables are defined, there is no problem definition to give values to these numbers. Therefore, PANACEA is given pointers to the locations where that information will be stored when a problem is generated or read in from a state file.

PANACEA provides a routine to install the code developer's generator routines in the command hash table. Thus, in addition to the routines to process the input, a function to install these routines in the command table must be provided. This routine is one of the defining routines for a PANACEA package (see Figure 4).

Install commands for the global package. These form part of the initial value problem generator, A.

```
/* GLOBAL_CMMNDS - install the commands for the global package */


global_cmmnds()

   {N_parts       = 0;


    PA_inst_c("make",  NULL, FALSE, 0, make_mesh, PA_zargs, commands);

    PA_inst_c("clear", NULL, FALSE, 0, clearh,    PA_zargs, commands);

    PA_inst_c("part",  NULL, FALSE, 0, parth,     PA_zargs, commands);

    PA_inst_c("side",  NULL, FALSE, 0, sideh,     PA_zargs, commands);


/* named switches, parameters, and names */

    PA_inst_c("start-time", param, SC_DOUBLE_I, 2, PA_pshand,

              PA_sargs, commands);
```

```
PA_inst_c("stop-time",  param, SC_DOUBLE_I, 3, PA_pshand,
          PA_sargs, commands);

return;}
```

A PANACEA package provides for three arrays: one array of integers called switches; one array of doubles called parameters; and one array of strings called names, which the developer may wish to use to control a package's operation. For example, a number of sub-iterations or a multiplier on some process might be managed by PANACEA on behalf of the code developer (i.e., the database manager overhead can be reduced by managing arrays rather than large numbers of scalars). The code developer can use this facility by providing a routine that specifies how many switches, parameters, and names are required for a given package and specifies their default values. PANACEA itself provides routines to handle input commands to set values for these quantities (collectively referred to as controls). In fact, PANACEA even provides the developer with the option of referring to the controls by names (at least from the generator input see Figure 4). The routine that defines the controls is one of the package defining routines (see Figure 5).

The normal execution sequence for a PANACEA generator follows:

- The command table is built by installing each package's commands.
- The controls for each package are defined.
- The variables for each package are defined.
- The input is read in and processed.
- The variables that are created are interned in the database managed by PANACEA.
- The database is dumped out into a state file.

FIGURE 5 - Define and initialize the controls for a PANACEA package. Note that this is one of the functions installed as part of the definition for the hydrodynamics package in Figure 1.

```
/* CONT_HYDRO - set the hydrodynamics package controls */

cont_hydro(pck)
   PA_package *pck;
   {static int n_names = 2, n_params = 15, n_swtches = 10;
```

```
PA_mk_control(pck, "hydrodynamics", n_names, n_params, n_swtches);

swtch[1]   = TRUE;                              /* hydro switch */
swtch[2]   = 0;
swtch[3]   = 0;
swtch[4]   = global_swtch[12];
swtch[5]   = 0;
swtch[6]   = 1;
swtch[7]   = 0;

param[1]   = 0.2;
param[2]   = 2.0;
param[3]   = 1.0;
param[4]   = 1.5;
param[5]   = 0.2;
param[6]   = 0.2;
param[7]   = 0.2;
param[8]   = 1.0;
param[9]   = 0.0;
param[10]  = 1.0;
param[11]  = 3.0/2.0;
param[12]  = 1.0e-6;
param[13]  = 0.1;
param[14]  = 2.0;
param[15]  = 1.0;

return;}
```

The last step is carried out by PANACEA alone. The step in which the variables are interned in the database requires a routine supplied by the developer to make PANACEA calls that associate local or global variables with entries in the database. This is the final defining routine for a package in the generation phase (see Figure 6).

FIGURE 6 - Intern the variables for the hydrodynamics package. This is a function defined for the initial value problem generator, A, by the hydrodynamics package.

```
/* INTERN_HYDRO - INTERN the variables of the hydrodynamics package
 *              - prior to writing the initial restart dump
 */

intern_hydro()
   {swtch[2]   = global_swtch[1];
    swtch[3]   = global_swtch[2];
    swtch[4]   = global_swtch[12];

    param[12]  *= (global_param[3] - global_param[2]);
    param[13]  *= (global_param[3] - global_param[2]);

    PA_INTERN(n, "n", double);
    PA_INTERN(p, "P", double);
    PA_INTERN(rho, "rho", double);

    return;}
```

Finally, to build a PANACEA generator, the code developer writes a driver module that issues calls to PANACEA to define the packages given the functions discussed above, to process command line arguments, and to hand control of the input reading process to PANACEA. All further action is controlled by PANACEA and the information in the input stream, i.e., reviewing information, writing the initial state file, and exiting. This driver is compiled and loaded with the package libraries, the PANACEA library, and some other lower level libraries such as PDBLib.

In practice, the first place to start is the global package. If the simulation code has a computational mesh, the routines to translate a collection of user specifications into a representation appropriate for the simulation routines should reside here. Controls that specify how many cycles to run, that keep the names of file families, etc. go in the global package. In addition, routines for direct visualization of the initial state before writing a state file might be most logically included in the global package.

This process would be repeated for each package of the simulation until a complete generator code was specified. It is important to remember that PANACEA attempts to make limited assumptions about what the code developer wants to do - a necessity to have some structure specifying a framework on which to hang simulation packages. It is best to keep in mind a picture of fundamental processes of data flow and management and computational flow control, i.e., to think about what has to be happening to get a generation code to work properly.

Simulation

The actual simulation phase of the PANACEA model is the simplest. Since PANACEA simulation codes only read state files that serve what PANACEA itself manages, the code developer only needs to provide the actual simulation routines. Actually, PANACEA allows for some additional routines to be supplied to increase the efficiency of data flow through the code and to handle simulation specific output requests.

Once the database has been read in from the state file, PANACEA alone has access to the variables in the database. In order to gain access to the data, the code developer must issue calls to connect local or global code variables to the information in the database. Even though the database is hash driven and is therefore quite efficient, PANACEA overhead can be reduced by connecting to package variables once and leaving those connections intact for the entire simulation run. This is not required, but it is very convenient and efficient. The code developer can supply a routine in which all of these connections are made on a package-by-package basis, i.e., one routine per package. These routines are defining routines for the packages. PANACEA will execute each of these packages once per simulation run before the main simulation routines are called (see Figure 7).

Most PANACEA packages have a main entry point which is where the real simulation work occurs. The main entry point is probably the routine which is most often the starting point when a code developer sets out to write a new simulation package. Typically, the main entry point it designed to be executed once per major computational cycle. From PANACEA’s point of view it is both an easy point to control and a natural point to monitor the performance and resource usage of a package. Figure 8 illustrates a main entry point routine.

PANACEA packages also include slots for routines to handle special purpose I/O and routines to run after the main simulations routines have been called for the last time in a run. The code developer can provide them if necessary and supply them to the call which defines the package to the simulation code.

PANACEA provides a service to sequentially execute the main simulation routines until a certain "time" has elapsed. Many variations on such a service are possible and the code developer does not have to use the ones PANACEA provides. Given the list of packages, the code developer can manipulate them in whatever fashion is appropriate to the problem at hand (see Figure 2).

FIGURE 7 - Initialize the hydrodynamics package. Note that this is one of the functions installed as part of the definition of the hydrodynamics package in Figure 1.

```
/* INIT_HYDRO - initialize and allocate hydro variables once */

init_hydro(pck)
   PA_package *pck;
   {int i, j;

/* PA_CONNECT global zonal variables */
    PA_CONNECT(rho,   "rho",    double *, TRUE);
    PA_CONNECT(n,     "n",      double *, TRUE);
    PA_CONNECT(p,     "P",      double *, TRUE);
    PA_CONNECT(massz, "mass-z", double *, TRUE);
    PA_CONNECT(t,     "T",      double *, TRUE);
    PA_CONNECT(ab,    "A-bar",  double *, TRUE);
    PA_CONNECT(zb,    "Z-bar",  double *, TRUE);};

/* allocate local zonal variables */
    cs    = MAKE_N(double, N_zones);
    dtvg  = MAKE_N(double, N_zones);
    pdv   = MAKE_N(double, N_zones);
    qdv   = MAKE_N(double, N_zones);
    voln  = MAKE_N(double, N_zones);
    volo  = MAKE_N(double, N_zones);

/* set some scalars */
    csmin  = param[9];
    hgamma = 1.0 + 1.0/(param[11] + SMALL);
```

```
/* initialize some zonal arrays */
    for (j = frz; j <= lrz; j++)
         {volo[j]  = vol[j];
          massz[j] = rho[j]*vol[j];};

    return;}
```

FIGURE 8 - The main entry point for a PANACEA package. Note that this is one of the functions installed as part of the definition of the hydrodynamics package in Figure 1.

```
/* HYDRO - the main hydro entry point */

hydro(pck)
   PA_package *pck;
   {int hyd_z;
    double hyd_dt;
    static int first = TRUE;

/* check that this package was requested */
    if (swtch[1] == FALSE)
       return;

    if (first)
       pck->space = (double) Ssp_alloc;

    PA_MARK_TIME;

    init_cycle();

/* do the real hydro work */
    hydro_wrk(&hyd_dt, &hyd_z);
```

```
/* record the hydro timestep vote */

    pck->dt          = hyd_dt;

    pck->dt_zone = hyd_z;


    PA_ACCM_TIME(pck->time);


    if (first)

       {pck->space = (double) Ssp_alloc - pck->space;

         first = FALSE;};


    return;}
```

To help with the monitoring of performance, each package has slots to record the CPU time spent in the package and the total memory required by the package. The latter is most useful when memory is being managed dynamically. PANACEA will print out statistics for each of the packages at the end of a run. In practice, this tends to be very useful information when algorithmic optimization is being pursued or when obscure behavior is suggesting memory bugs. See Figure 8.

Data Management for Visualization

Ultimately, a simulation code system is limited by its ability to produce information that can be presented in a meaningful way. The visualization problem is one of current intense interest and effort. From the point of view of PANACEA, the important task is to get numerical results out of the simulation code and into a form usable by a visualization system. Because of the variety of visualization systems and their input formats, PANACEA can provide active services for the first part, i.e., it gets data out of the simulation efficiently. It only passively supports the second part by accessing the data it puts out from the simulation. The code developer is left with the task of formatting the information for the visualization system of choice.

There are two basic kinds of data flow out of the simulation code. First, at each cycle, the data that is completed in a cycle must be written out according to user requests. Second, information that is accumulated across more than one cycle must be addressed.

The "snapshot" requests (complete in one cycle) can be put out either directly for the visualization system or into some intermediate form for subsequent processing. In the PANACEA model, output data are not kept in the state of the running code. In fact, PANACEA services are aimed at transferring the data from the code to data files as efficiently as possible. The main advantage of this scheme is that the task of visualization can be done on a different machine while the simulation progresses if data is put out every cycle. A second advantage is that the running code does not grow as a result of the stored output data.

The output whose meaning comes from the results of several computational cycles (generically referred to as time plots) is handled differently. PANACEA dumps the information computed in each cycle out into an intermediate file family at the end of the cycle. After the simulation is complete a separate post-processing code transposes the information and writes it in the desired format. This process naturally involves the code developer to a greater extent than the generation process, given that input parsing is ASCII based whereas visualization systems have a variety of input mechanisms.

The PACT tool, PDBView, has facilities to display the "snapshot" files generated by PANACEA. PANACEA has support for post processor codes to emit files for ULTRA II, the PACT utility for presentation, analysis, and manipulation of 1D data sets. Typically these are time plots since the snapshot mechanism covers most other cases.

FIGURE 9 - Sample PANACEA plot request specifications. All of the range and domain variables are in the database.

A velocity vector plot every 20.0 time units from 0 to 100.0 over the entire mesh.

graph {vx,vy}(t=step(0.0;1.0e2;20.0),rx,ry)

A vector plot with components rho and n every 20.0 time units from 0 to 100.0 over the entire mesh.

graph {rho,n}(t=step(0.0;1.0e2;20.0),rx,ry)

The radiation field at time 50.0 and frequency 1000.0 over the entire mesh. This could be rendered as a contour plot, a surface plot, a wire-frame plot, or an image plot.

graph jnu(t=50.0,rx,ry,nu=1000.0)

The variable, p, at every time over the entire mesh.

graph p(t,rx,ry)

PANACEA has very powerful and general mechanisms for plotting data from the database. These mechanisms translate output requests into data for a visualization system (see Figure 9).

Source Functions and Source Data

Many systems of equations that are simulated numerically involve source functions of one sort or another. Also, initial value and boundary value data must be gotten into the simulation. In general, the amount of information for these purposes is too large to include in an ASCII input file. In any case, it is straightforward enough to handle all of this type of information in complete generality.

PANACEA provides a set of services to manage source or boundary value data. The characterization of the data is specified in the input phase. The characterization may include information as to where or when the source is to be applied, a file from which to read the data, etc. During the simulation phase, the source data is interpolated by PANACEA and handed to simulation routines upon request.

To make this process as efficient as possible, when source data is so voluminous as to require data files to contain it reasonably, it is prepared in advance in the form of PDB files. The self-describing nature of these files and variable database of PANACEA make it easy to code and efficient at run time. PANACEA provides routines to facilitate the writing of a code to gather source information from whatever sources the code developer has and to produce the required PDB source files. The further advantage of this approach is that many simulation runs can be done (and typically are) with one set of source information that does not change. Having the PDB files separate makes this fast and convenient.

The source variable notion is very natural from the data generation and specification point of view. However, from the point of view of the algorithms, additional issues concern initial or boundary value data. Often this data is not directly part of the variable database (perhaps because it is applied to the boundaries of one of the database variables). PANACEA distinguishes between initial or boundary value data and source data. It treats source data as a separate sub-class of initial value data. In particular, source data usually involves information coming in over many computational cycles, and it may require interpolation with respect to "time". It also may involve large quantities of data that are most conveniently kept in files (PDB files actually). PANACEA services make as much of this transparent to the developer as possible.

PANACEA services gather up specifications, doing some processing when appropriate, and hold that information for the simulation on demand. These services also interpolate "temporal" data to the correct "time" in a simulation. Even though PANACEA makes a distinction between sources and initial value data, it gathers them in the same fashion (see Figure 10).

FIGURE 10 - Some examples of source and initial value specifications in PANACEA.

Set a boundary condition tagged as "boundary-pressure". The hydrodynamics package will look for "boundary-pressure" specifications and interpolate appropriate values from the data in file, "press.dat".

specify bc boundary-pressure from press.dat

Set a boundary condition tagged as "constant-vx". The hydrodynamics package will look for "constant-vx" specifications and take the given values as the place to apply it.

specify bc constant-vx 1 1 1 5

Impose a profile on the source variable tagged as "temperature-floor". The relevant package looks for a source variable tagged this way. It will interpret (with PANACEA's help) and interpolate the "time", "value" pairs.

specify src temperature-floor

s 0.0 50.0

s 0.3 100.0

s 0.5 300.0

s 0.8 400.0

s 1.0 500.0

Variables defined in the PANACEA database can be directly controlled by source variables. In this case, the variable rho would be interpolated from the source file rho.dat instead of being computed by a hydrodynamics package.

specify rho from rho.dat

Typically, a PANACEA simulation code system will have a program to build source files for simulation runs. PANACEA has the routines to do the generic work here. It helps to map ASCII (or other) data files that are derived from experiments or other simulations into the database of the simulation code system. The developer provides the routines with specific information about the input data; the PANACEA services package it in a form that is most efficient for the simulation code to use.

Data Structures

As the above discussion indicates, PANACEA attempts to work with many abstract concepts to delineate the boundary between the generic and the specific. The key to working with an abstraction is to create a concrete representation. Briefly, PANACEA uses the C struct mechanism to define structured data types representing packages, variables, source variables, initial value specifications, plot requests, and generator commands. These structures and a collection of routines to create, release, and manipulate them are the key to realizing PANACEA's goals.

One additional set of structures (provided by an associated math library called PML, Portable Mathematics Library) is important for its ability to tie some key aspects of data flow together. In attempting to deal with data that are generated from a specification that is natural for the end user, computed within a form natural to numerical algorithms, stored in intermediate data files or transmitted over networks, and finally visualized in yet a different form, it was important to formulate a description of data that is self-consistent and mathematically precise. In this way, data sets could be assembled, passed among computational routines, stored, and visualized all in a relatively efficient and general way. This description had to be general enough to deal with multidimensional data with a variety of properties.

The key concepts are those of sets and mappings. Mathematically, for the purposes of both computation and visualization, one is interested in working with mappings of domain sets to range sets. A set consists of a collection of data items along with information describing the type of elements, the dimensionality of the set, the dimensionality of the elements, the topology of the set, the metric information, etc. Ideally, the set structure contains sufficient information to allow routines to process them without additional specifications. For example, a set might be a collection of 3-vectors on a 2-sphere or a simple 1-D array.

Sets can be related by mappings that describe how elements of two sets are related. The mapping structure contains a domain set, a range set, some information about extremes of both sets, and other descriptive information (e.g., relative centering). Ideally, a differentiation routine could be handed a mapping whose properties it understands and return a new mapping representing the gradient of the argument mapping. The goal is to objectify a complete collection of information for the purposes of storage and computation.

A third layer of structure combines a mapping with rendering specifications that make it possible to visualize the data set embodied in the mapping. This structure is referred to as a graph, and it provides a means of associating mathematically complete data sets (i.e. mappings) with information describing how they are to be displayed. For example, a mapping with a two-dimensional domain and a one-dimensional range could be rendered as a contour plot, a wire frame mesh plot, a surface plot, or an image plot. These rendering techniques all take the same fundamental data, embodied in the mapping structure (see Figure 9).

Summary and Further Directions

PANACEA provides a collection of services to facilitate the production of numerical simulation codes and to increase the reusability and shareability of simulation packages. By attempting to provide services to do everything that is generic to "all" simulation codes, PANACEA also provides some standards of data exchange, management, and visualization.

Although coded in C, PANACEA is coded in an object-oriented style. The most important ramification of this is that abstract objects (e.g., packages, variables, and mappings) have a relatively faithful concrete representation. This puts PANACEA on a sound conceptual basis and helps to delineate the generic from the specific in simulation code systems.

As an additional benefit, the modularization that follows from this style lends itself to natural coarse-grained parallelization of code systems. In practice, packages can also be organized so as to make fine grained parallelization possible because the controlling structures and the data objects of PANACEA do not really intrude into the detailed workings of the simulation algorithms. Therefore, while PANACEA helps modularize a code system so that packages or large parts of packages might be run in parallel, it does not interfere with parallelizing individual routines which permit it.

The encapsulation of abstract objects in concrete representations facilitates the process of manipulating these objects symbolically. I have used PANACEA with the PACT SCHEME interpreter to give users of one PANACEA code the ability to manipulate the code in very broad and general ways.

This technique allows the manipulation of the packages' execution sequences, the examination of the state of the running code, and the changes in the state of the code. When carried to its logical conclusion, this method will also permit the prototyping of algorithms at the LISP level before investing the effort in writing more efficient code at a lower level.

Finally, PANACEA can bind simulation packages generated by a tool, such as ALPAL, into entire code systems. PANACEA complements ALPAL very neatly by attending to large control and data flow issues, while ALPAL uses the PANACEA services rather than getting loaded down with these issues.

PANACEA Objects

In this section, the objects of PANACEA and their properties are discussed in detail.

PACKAGES

A PANACEA package consists of a collection of controlling information which is defined by the application through function calls to PANACEA routines and a set of functions supplied by the application. These functions perform a well-defined set of operations and PANACEA invokes them in the proper sequence to carry out the desired simulation.

The functions that an application supplies are:

gencmd()	define the generator commands
dfstrc(pdrs)	define data structures used in package variables
intrn()	intern variables into the database at generation
defun(pck)	define new units
defvar(pck)	define the package variables
defcnt(pck)	define the package controls
inizer(pck)	initialize the package
main(pck)	the main entry into the package
ppsor(pr, t)	handle package specific output
finzer(pck)	close out the package

All of these functions except ppsor must return 1 if they complete successfully and 0 otherwise.

VARIABLES

A PANACEA variable consists of a collection of descriptive information and the actual data associated with a computational quantity used in a numerical simulation. PANACEA variables can be scalars or arrays. They correspond to what code developers traditionally think of as variables in systems of equations. In addition they include information which governs their use. This information is collectively referred to as the attributes of the variable.

Each PANACEA variable has a name. Since PANACEA variables are owned by packages, each PANACEA variable knows which package owns it. PANACEA variables also contain dimension information. Since PANACEA variables are defined before the values of their dimensions are known, PANACEA variables keep track of the addresses of the application code variables (typically scalars) which will contain the correct values for the dimensions when the PANACEA variables are accessed. PANACEA variables also contain conversion factors between CGS, internal, and external unit systems. Because the data associated with a PANACEA variable may be stored on disk files, PANACEA variables keep track of information describing any relevant data files.

Three key attributes of a PANACEA variable require special discussion. Typically the data associated with PANACEA variables is in the form of an array which can be extremely large depending on the problems with which the application code deals. To provide flexibility, efficiency, and control, the data arrays associated with PANACEA variables can be allocated and released under the control of these three attributes. The scope attribute, class attribute, and persistence attribute. The scope attribute pertains to the lifetime of a PANACEA variable or more precisely its data. The class attribute defines the behavior of PANACEA variables with regard to PA_CONNECT. The persistence attribute defines the behavior of PANACEA variables with respect to PA_DISCONNECT.

Variable Scopes

The scopes defined in PANACEA are:

DEFN

RESTART

DMND

RUNTIME

EDIT

DEFN

PANACEA variables with the DEFN scope are considered to be a part of the state of the numerical simulation. These variables, in contrast to RESTART variables whose properties they share, must be read from a state file before the RESTART variables. This distinction permits variables which define certain problem parameters (hence the designation DEFiNition variables) to be read in first, before the other RESTART variables which may depend on the information these variables specify.

RESTART

PANACEA variables with the RESTART scope are considered to be a part of the state of the numerical simulation. These variables define the state of the numerical simulation and cannot be derived from other state information. The PANACEA model dictates that with a complete set of RESTART variables in a state file, the simulation can be restarted from the state defined by the contents of the state file. These variables are read into database memory at the time the state file is opened.

DMND

PANACEA variables with the DMND scope are considered to be a part of the state of the numerical simulation. Like RESTART variables they define the state of the simulation. It sometimes happens that for reasons of size or efficiency, some state variables should not be read into memory when the file is opened, but when access is specifically requested via PA_CONNECT. The DeMaND scope provides for this contingency.

RUNTIME

PANACEA variables with RUNTIME scope can be derived from state information defined by the RESTART variables. As such it is not necessary to write them into state files or read them out of state files. This saves space in state files and the time it would take to read or write the variables.

EDIT

There are some variables in a system of equations to be simulated which are only temporary or auxiliary. It is often useful to look at these variables for purposes such as debugging. In every computational cycle, PANACEA variables with EDIT scope are allocated just before the packages which own them are entered, their contents are "edited" as soon as the package returns, and the space associated with them is released.

Variable Classes

Another attribute is class. The class attribute specifies the behavior of PANACEA regarding the handling of the various scopes especially in the PA_CONNECT process. The classes defined in PANACEA are:

REQU

OPTL

PSEUDO

REQU

PANACEA variables in this class are required. That is to say that when access to such a variable is requested via PA_CONNECT, it is a fatal error if at least one of the following sources of data values does not provide data for the PANACEA variable: the state file (DEFN, RESTART, and DMND variables only check this source); source specification data (via SPECIFY command); an initializer function (optionally provided with the variable definition); or a broadcast default value (optionally provided with the variable definition). These means of initialization are attempted in the order stated above.

OPTL

PANACEA variables with the OPTL class can be thought of as optional. In contrast to REQU variables, PANACEA returns a NULL pointer if the data cannot be found for the variable (by the same procedure as for the REQU variables) instead of terminating with a fatal error.

Many variables which are part of the state of a simulation are nevertheless OPTL. The usual case is for a variable in a package which may be turned off. In such a case, the absence of the variable in the state file is not an impediment to the running of the simulation.

PSEUDO

In some simulations auxiliary variables which can be thought of logically as arrays are implemented as scalars or subsets of other arrays. Ordinarily this is no concern of PANACEA. However, when it is desired to put such data out into post processor files, PANACEA attempts to assist. The generic descriptive information that PANACEA requires of all variables is insufficient to specify how to construct the logical array structure desired in the output files from the actual information as implemented in the simulation coding. The application package will have to supply routines to accomplish this mapping. The PSEUDO class tells PANACEA to handle such a variable in a special way which requires more coordination with the particular package. The PSEUDO class is currently only meaningful in connection with variables of EDIT scope.

Variable Persistence

Another attribute is that of variable persistence. This specifies the actions PANACEA is to take in the PA_DISCONNECT process. The defined persistence categories are:

KEEP

CACHE_F

CACHE_R

REL

KEEP

In many situations the application developer wants to relinquish the applications access to a PANACEA variable's data yet insure that PANACEA retains the data in memory. This allows for rapid re-connections later via PA_CONNECT.

CACHE_F

For larger data arrays, the application may not wish to have access to the data for part of the simulation, not wish PANACEA to retain it in memory either, and still require the data at a later point in the simulation. The CACHE_F persistence tells PANACEA to write the data out to a temporary scratch file and release its own copy from memory. It also tells PANACEA that the variable will never have its size changed via PA_change_dim or PA_change_length.

CACHE_R

For larger data arrays, the application may not wish to have access to the data for part of the simulation, not wish PANACEA to retain it in memory either, and still require the data at a later point in the simulation. The CACHE_R persistence tells PANACEA to write the data out to a temporary scratch file and release its own copy from memory. Unlike CACHE_F, CACHE_R variables can be resized. The distinction is enforced for the sake of execution and disk space efficiency.

REL

Perhaps the most common situation for RUNTIME variables when the data is not going to be used and the application invokes PA_DISCONNECT is for PANACEA to release the memory associated with the PANACEA variable data altogether. It is not retained in any way and would have to be recreated if a PA_CONNECT call were to be made at a later point in the simulation.

The following table summarizes the interaction between scope, class, and persistence:

SCOPE	CLASS			PERSISTENCE		
	REQU	OPTL	PSEUDO	KEEP	CACHE_X	REL
DEFN	+	-	-	-	-	-
RESTART	+	+	-	+	+	+
DMND	+	+	-	+	+	+

RUNTIME	+	+	-	+	+	+
EDIT	-	-	+	NA	NA	NA

In this table "+" means that the behavior of the combination is defined, and "-" means that the behavior of the combination is undefined. The undefined combinations may be defined in later releases of PANACEA.

Variable Centering

PANACEA variables which are not scalar variables as defined by PA_inst_scalar also have an attribute specifying their centering relative to the "spatial" mesh. (NOTE: this concept will be properly generalized in future releases of PANACEA).

The legal values for the centering of variables with an underlying mesh are:

```
Z_CENT         zone centered

N_CENT         node centered

E_CENT         edge centered

F_CENT         face centered

U_CENT         uncentered
```

Variable Allocation

PANACEA can handle either of two situations on a variable by variable basis. One, memory is to be dynamically allocated and managed by the PANACEA database services. Two, memory is statically allocated by a compiler and the PANACEA database services hand out pointers to such spaces.

The legal values for the allocation strategy of a variable are:

STATIC	static allocation (i.e. by compiler)
DYNAMIC	dynamic allocation (i.e. by run time memory manager)

SOURCE VARIABLES

INITIAL VALUE SPECIFICATIONS

PLOT REQUESTS

UNIT SPECIFICATIONS

The PANACEA C API

The following commands are broken down by service category. The PANACEA service categories are:

Database Definition and Control

Variable Definition

Control Definition

Unit Definition and Control

Database Access

Simulation Control

Plot Request Handling

Initial Value Problem Generation Support

Source Variable/Initial Value Data Handling

Time History Data Management

Miscellaneous

Commands are also sorted into two categories: Basic, PANACEA applications must invoke these functions at some appropriate point; and optional, these are provided for the benefit of the application but are not required.

Database Definition and Control Functionality

This group of functions governs the creation and management of the PANACEA database. They are relevant to both the generator and simulation codes.

These functions let applications define packages to PANACEA and query PANACEA about packages in a PANACEA code.

Basic Functions

PA_gen_package

PA_run_time_package

Optional Functions

PA_current_package

PA_install_function

PA_install_identifier

PA_GET_MAX_NAME_SPACE

PA_SET_MAX_NAME_SPACE

PA_CURRENT_PACKAGE

PA_current_package()

At any moment in a generator or simulation code, PANACEA has a notion of the current package. It is the one whose controls are connected to the global variables, swtch, param, and name. Applications may wish to query or access the current package.

This function returns a pointer to the PA_package structure which is PANACEA's current package.

PA_DEF_PACKAGE

```
PA_def_package(char *name,
                              PFInt gcmd,
                              PFInt dfs,
                              PFInt dfu,
                              PFInt dfr,
                              PFInt cnt,
                              PFInt izr,
                              PFInt inr,
                              PFInt mn,
                              PFPPM_mapping psr,
                              PFInt fzr,
```

```
                    PFInt pcmd,
                    char *fname),
```

To define a package in complete generality the following functions must be supplied:

gcmd	installs the functions which handle generator commands
dfs	defines data structures used by package variables
dfu	defines the units of the package
dfr	defines the variables of the package
cnt	defines and sets initial or default values for the controls of the package
izr	initializes the package one time before the main entry point is called
inr	interns package variables into the database prior to writing the state file
mn	the main entry point for the package
psr	makes PM_mappings (for output purposes) of package specific quantities which PANACEA's generic capabilities cannot handle
fzr	finalizes a package one time before exiting the simulation
pcmd	installs the functions which handle post processor commands
fname	names a dictionary file to be processed at runtime by the dfr function

With the exception of the psr function which returns a pointer to a PM_mapping structure if successful and NULL otherwise, these functions must return TRUE if successful and FALSE otherwise. If any of these functions is irrelevant to the package being defined, NULL should be passed in in its place.

The name of the package must also be supplied.

If successful, the function returns a pointer to a PA_package. Since PANACEA keeps an internal list of packages, it is not necessary for the application to do anything with the return value of this function.

PA_GEN_PACKAGE

```
PA_gen_package(char *name,
               PFInt cmd,
               PFInt dfs,
               PFInt dfu,
               PFInt dfr,
               PFInt cnt,
```

```
PFInt inr,
char *fname)
```

To define a package to a generator code the application must supply functions:

cmd	installs the functions which handle generator commands
dfs	defines data structures used by package variables
dfu	defines the units of the package
dfr	defines the variables of the package
cnt	defines and sets initial or default values for the controls of the package
inr	interns package variables into the database prior to writing the state file
fname	names a dictionary file to be processed at runtime by the dfr function

These functions must return TRUE if successful and FALSE otherwise. If any of these functions is irrelevant to the package being defined, NULL should be passed in in its place.

The name of the package must also be supplied.

If successful, the function returns a pointer to a PA_package. Since PANACEA keeps an internal list of packages, it is not necessary for the application to do anything with the return value of this function.

PA_GET_MAX_NAME_SPACE

PA_GET_MAX_NAME_SPACE(int flag)

Returns the value of the name space flag in the argument flag. See the PA_SET_NAME_SPACE function for further information.

PA_INSTALL_FUNCTION

PA_install_function(char *name, PFByte fnc)

This function provides a means for applications to make arbitrary procedures known to the PANACEA database. This is critical for interactive or interpreted applications which may need to invoke a function given only its name. It associates an address to which control can be passed with an ASCII name.

The arguments to this function are: name, an ASCII string naming the function; and fnc, a compiled function which may be invoked by PANACEA. Currently the function installed this way can take no arguments and returns no value.

This function returns nothing.

PA_INSTALL_IDENTIFIER

```
PA_install_identifier(char *name, byte *vr)
```

This function provides a means for applications to make arbitrary compiled variables known to the PANACEA database. This is critical for interactive or interpreted applications which may need to refer to a variable given only its name. It associates an address with which a value can be accessed with an ASCII name.

The arguments to this function are: name, an ASCII string naming the function; and vr, a compiled variable which may be referenced by PANACEA.

This function returns nothing.

PA_RUN_TIME_PACKAGE

```
PA_run_time_package(char *name,
                    PFInt dfs,
                    PFInt dfu,
                    PFInt dfr,
                    PFInt cnt,
                    PFInt izr,
                    PFInt mn,
                    PFPPM_mapping psr,
                    PFInt fzr,
                    char *fname)
```

To define a package to a simulation code the application must supply functions:

dfs defines data structures used by package variables
dfu defines the units of the package
dfr defines the variables of the package
cnt defines and sets initial or default values for the controls of the package
izr initializes the package one time before the main entry point is called
mn the main entry point for the package (executed once each major cycle)
psr returns a PM_mapping pointer for PSEUDO EDIT variables (each cycle)
fzr shuts down the package (executed once after the last call to any main entry)

All of the functions except psr must return TRUE if successful and FALSE otherwise. psr returns a pointer to a PM_mapping associated with a plot request of a PSEUDO EDIT variable. If any of these functions is irrelevant to the package being defined, NULL should be passed in in its place.

The information defined via dfu, dfr, and cnt can alternatively be specified in an ASCII text file. This file can be specified by fname. If fname is not NULL, dfu and cnt are ignored and the file fname is read in and processed using a dfr to supply interpretation of the dictionary fields.

The name of the package must also be supplied.

If successful, the function returns a pointer to a PA_package. Since PANACEA keeps an internal list of packages, it is not necessary for the application to do anything with the return value of this function.

PA_SET_MAX_NAME_SPACE

PA_SET_MAX_NAME_SPACE(int flag)

If flag is TRUE then the name space of PANACEA variables is maximized in that variables are defined to the database under a name made up from the package name, a hyphen, and the variable name. In this scheme, there is less potential for name conflicts. If flag is FALSE then variable are defined to the database simply by the variable name.

Variable Definers

Variables are one of the main objects in PANACEA and consequently their definition to the database is crucial. Both scalar and array variables can be defined in the database and variables may have any data type used in the simulation code system.

Basic Functions

PA_def_var

Optional Functions

PA_definitions()

PA_variables(int flag)

PA_DEF_VAR

```
PA_def_var(char *vname,
           char *vtype,
```

```
                byte *viv,
                byte *vif,
                ...)
```

To define a variable to the PANACEA database certain descriptive information must be supplied.

vname defines the name of the variable

vtype defines the type of the variable (variables may be of any type including structures)

viv a pointer to a default initial value to be broadcast into the array

vif a pointer to a function which can be called to compute initial values for the variable

... defines the attributes, dimensions, physical units, and optionally the data of the variable

The attributes of the variable are specified by attribute identifier, attribute value pairs. When all the attributes are specified the ATTRIBUTE keyword terminates the processing of attribute specifications. By default, variables will be defined as RUNTIME, OPTL, REL, U_CENT, DYNAMIC. To set the scope, the SCOPE keyword is used followed by one of the values: DEFN; RESTART; DMND; RUNTIME; or EDIT. To set the class, the CLASS keyword is used followed by one of the values: REQU; OPTL; or PSEUDO. To set the persistence, the PERSIST keyword is used followed by one of the values: KEEP; CACHE_F, CACHE_R; or REL. To set the centering of the variable with respect to its spatial mesh use the CENTER keyword followed by one of the values: Z_CENT; N_CENT; E_CENT; F_CENT; or U_CENT.

The dimensions specifications consist of a set of pointers to the integer variables which will contain the actual dimensional numbers. It is crucial to realize that at the time that the variable is being defined the sizes of the array dimensions is not known (in general). Therefore, pointers to the scalars which will contain that information must be provided! The list of dimensions is terminated with the PANACEA defined pointer, DIMENSION.

Another feature of dimension specification in PANACEA is that a dimension can be specified in three ways. First a single pointer indicates to PANACEA that the dimension scalar represents the number of elements in that dimension. Second, each time the PANACEA provided pointer, PA_DON, occurs in the dimension list, the next two integer pointers in the argument list are interpreted to be a pointer to the value of the minimum index value of the dimension and a pointer to the value of the number of elements in the dimension of the variable, respectively (i.e. DON means Dimension, Offset, and Number). Third, each time the PANACEA provided pointer, PA_DUL, occurs in the dimension list, the next two integer pointers in the argument list are interpreted to be a pointer to the value of the minimum index value of the dimension and a pointer to the value of the maximum index value of the dimension of the variable, respectively (i.e. DUL means Dimension, Upper, and Lower). The reason for this distinction between the second and third case is that although the arithmetic is simple, the meaning of the scalar variables describing the dimensions is different. Since PANACEA does not deal with the dimension range values but pointers to the values, it must respect the semantics of these dimensioning scalars. This in turn dictates the presence of at least the two cases here.

The physical units of a PANACEA variable are described in terms of a collection of conversion factors from CGS units. There are arrays of factors for both internal and external units. They are indexed the same way and so the effective specification of the physical units of a variable are in terms of the integer indices into these arrays. PA_def_var processes the arguments after the dimension specification terminator, DIMENSION, and before the unit specification terminator, UNITS. It takes the arguments from left to right and multiplies the values in the conversion factor arrays indexed by the values together. If it encounters the PANACEA provided argument, PER, it collects the factors remaining until, UNITS, and divides the two terms to compute the correct conversion factors (internal and external units conversion). The arguments are handled this way so that the call to PA_def_var reads very much like English and the units are "clear" at a glance. PANACEA goes further by providing the following constants as the index values:

Macro	Variable	Unit
RAD	PA_radian	angle
STER	PA_steradian	solid angle
MOLE	PA_mole	number
Q	PA_electric_charge	Coulomb
CM	PA_cm	length
SEC	PA_sec	time
G	PA_gram	mass
EV	PA_eV	energy
K	PA_kelvin	temperature
ERG	PA_erg	energy
CC	PA_cc	volume

Application developers are encouraged to use this practice to make the units they define easy to read in this context of variable definition.

Finally, if a variable is defined with ALLOCATION equal to STATIC, a pointer to data for the variable must be supplied after the UNITS terminator.

The syntax for PA_def_var is:

```
PA_def_var(name, type, init_val_ptr, init_func_ptr,
                     attribute_spec, ..., ATTRIBUTE,
                     dimension_spec, ..., DIMENSION,
                     unit_spec, ..., UNITS
```

```
            [, data_ptr])

attribute_spec := attribute_id, attribute_val
attribute_id   := SCOPE | CLASS | PERSIST |
                  CENTER | ALLOCATION

attribute_val(SCOPE)      := DEFN | RESTART | DMND |
                             RUNTIME | EDIT
attribute_val(CLASS)      := REQU | OPTL | PSEUDO
attribute_val(PERSIST)    := REL | KEEP | CACHE_F | CACHE_R
attribute_val(ALLOCATION):= STATIC | DYNAMIC
attribute_val(CENTER)     := Z_CENT | N_CENT | F_CENT |
                             E_CENT | U_CENT

dimension_spec := n_elements_ptr |
                  PA_DUL, lower_bnd_ptr, upper_bnd_ptr |
                  PA_DON, offset_ptr, n_elements_ptr
unit_spec      := unit_index | PER
unit_index     := RAD | STER | MOLE | Q | CM | SEC |
                  G | EV | K | ERG | CC |
```

No attributes are required to be set (ATTRIBUTE must appear in the arg list) and the default values are:

RUNTIME, OPTL, REL, U_CENT, DYNAMIC

Control Accessors/Definers

Optional Functions

- PA_mk_control(PA_package *new, char *s, int n_a, int n_p, int n_s)
- PA_control_set(char *s)

Unit Conversion/Definition Functionality

Unit/Conversion Definitions

Basic Functions

- PA_def_units(int flag)

  Optional Functions

- PA_def_unit(double fac, ...)

Unit/Conversion Setup

Basic Functions

- PA_set_conversions(int flag)
- PA_physical_constants_ext()
- PA_physical_constants_cgs()

  Optional Functions
- PA_physical_constants_int()

Database Access Functionality

Variable Access for Simulation

Basic Functions

PA_INTERN
PA_CONNECT
PA_DISCONNECT
PA_ACCESS
PA_RELEASE
PA_change_dim
PA_change_size

PA_INTERN

```
PA_INTERN(byte *var, char *name)
```

Intern a given array in the variable data base.

PA_CONNECT

```
PA_CONNECT(byte *var, char *name, int flag)
```

In the PANACEA model the PANACEA database is viewed as the manager of dynamic memory. As such one of its key functions is to dispense pointers to the data associated with PANACEA variables. PA_CONNECT is the principal means of doing this.

The call to this macro takes the application pointer which is to be set, var; the name of the variable in the PANACEA database; and a flag that informs PANACEA that it must track this pointer so that its value may be reset it another database operation changes the space to which the pointer points (see PA_change_dim). If the value of flag is TRUE then PANACEA will reset the application's pointer if the space is reallocated or released by another PANACEA operation.

The following steps are performed when a PA_CONNECT request is made:

The variable specified by name is looked up in the database hash table.
If the variable is not found PA_ERR is called and the application exits with the error message: VARIABLE name NOT IN DATA BASE - PA_GET_ACCESS.

The class of the variable is checked. If the class is PSEUDO PA_ERR is called and the application exits.

If the scope of the variable is DMND and it has not already been read in (presumably by another PA_CONNECT operation) the variable is read in from the state dump file.

Each PANACEA variable has two sets of dimensions: 1) derived from the contents of the addresses of integer scalars (necessary for correct dynamic behavior at run time); and 2) derived from the dimensions as specified in the PDB file (necessary for correct storage behavior). On PA_CONNECT, PANACEA reconciles these two sets of dimensions to obtain consistency and computes the actual current size in elements of the variable. The reconciliation of the two sets of dimensions is done via the following steps:

- Compute the total length implied by the file dimensions.
- Compute the total length implied by the PANACEA dimensions.
- If they agree assume that they are consistent.
- If the file dimensions imply a zero length, assume they are inconsistent and change them to match the PANACEA dimensions.
- If the PANACEA dimensions imply a zero length, assume they are inconsistent and change them to match the file dimensions.
- Otherwise assume the PANACEA dimensions are correct and change the file dimensions to match.

- Set both the PANACEA variable size and the file size to the selected size.

If the class of the variable is REQU the following actions are taken:
- If the PANACEA variable data pointer is not NULL return it.
- If the PANACEA variable data pointer is NULL; the size determined above is positive; and one of following initialization modes is available (in order): a default value; source variable, or initialization function, attempt to initialize a block of space.
- If the preceding step fails PA_ERR is called and the application exits

If the class of the variable is OPTL the following actions are taken:
- If the PANACEA variable data pointer is not NULL return it.
- If the PANACEA variable data pointer is NULL; the size determined above is positive; and one of following initialization modes is available (in order): default value; source variable; or initialization function, attempt to initialize a block of space.
- If the preceding step fails NULL is returned.

Set the PANACEA variable data pointer to the address of the space

Set the pointer var to the address of the space.

If track is TRUE, PANACEA is requested to keep track of this reference by adding it to the list of tracked pointers.

Return.

PA_DISCONNECT

```
PA_DISCONNECT(char *name, byte *ptr)
```

In the course of doing a numerical simulation, an application may wish to relinquish access to the data associated with a PANACEA variable. Most likely this access was obtained via PA_CONNECT. Since PANACEA is monitoring database access, it is not a good idea to simply set the application pointer to NULL because the database may be tracking that pointer and it could mysteriously be reappear later with a value.The correct procedure to invoke PA_DISCONNECT to inform PANACEA that this access is to be terminated.

The arguments to this macro are: name, the name of the PANACEA variable in the database; and ptr, the application pointer whose access to the data associated with the named

PANACEA variable is to be relinquished.

The following steps are carried out when a call to inform the PANACEA database that the application no longer requires a specified reference to the data associated with a PANACEA variable:

The variable specified by name is looked up in the database hash table. If the variable is not found PA_ERR is called and the application exits with the error message: VARIABLE name NOT IN DATA BASE - PA_REL_ACCESS

If the scope of the variable is EDIT the following actions are taken:

- The space associated with the PANACEA variable is freed
- The PANACEA variable's data pointer is set to NULL

Remove the reference ptr from the list of pointers pointing to this data

Set the reference ptr to NULL if the PANACEA variable is not a scalar variable as defined by PA_inst_scalar.

If there are no more references to the data associated with the PANACEA variable the following actions are taken:

- If the persistence of the PANACEA variable is REL, the space associated with the PANACEA variable is freed and the PANACEA data pointer is set to NULL.
- If the persistence of the PANACEA variable is CACHE_F or CACHE_R, the space associated with the PANACEA variable is freed after writing it out to a disk file (this last part not implemented yet) and the PANACEA data pointer is set to NULL.
- If the persistence of the PANACEA variable is KEEP, the space associated with the PANACEA variable is retained by the PANACEA database.
- If the variable is a scalar as defined by PA_inst_scalar, the scalar's value is set to an appropriate representation of 0.

Return

PA_ACCESS

```
PA_ACCESS(char * name, type, long offs, long ne)
```

Return a connection to a sub-space of an array.

PA_RELEASE

```
PA_RELEASE(char * name, byte * ptr, long offs, long ne)
```

Release the access to an array sub-space.

PA_CHANGE_DIM

```
void PA_change_dim(int *pdm, int val)
```

Change the value of an integer quantity, pointed to by pdm, to which the PANACEA variable dimensions of one or more PANACEA variables point to the given value, val, and reallocate all of the PANACEA variables which have this quantity as a dimension. This rather elaborate and expensive operation is the most consistent way of handling PANACEA database variables whose sizes are changing as a result of dynamically recomputing the computational domain (e.g. a spatial or frequency mesh). The alternate method provided by PA_change_size is not consistent and can produce extremely bizarre behavior which is difficult to debug.

The arguments to this function are: pdm, a pointer to the integer value to be changed (this should match at least one of the quantities used in a PA_def_var call used to define the database variables; and val, the new value to be assigned to the location pointed to by pdm.

NOTE: the use of val here is trivial but is intended to help enforce the discipline of using PA_change_dim correctly.

PA_CHANGE_SIZE

```
void PA_change_size(char *name, int flag)
```

Reallocate the PANACEA variable specified by name using the current values implied by set of dimensions specified by flag. With this call it is assumed that the application has changed the value of a dimensioning integer and now wants PANACEA to handle the resizing of the space associated with a particular PANACEA variable. In general, this is an extremely risky thing to do. However, some code systems which have automatically generated code can enforce consistency via the code generator. In such a case, the performance overhead of PA_change_dim is diminished to some extent.

The arguments to this function are: name, a string containing the name of the PANACEA database variable to be reallocated; and flag, an integer having either of the two values PA_FILE or PA_DATABASE signifying which set of dimensions (file or database respectively) to use in enforcing consistency on this one variable.

NOTE: use this function at your own risk!

Structured Data Support

To provide an additional dimension in data handling for FORTRAN packages and to support interactive runtime data structuring, PANACEA supplies services to define, create and release instances, and get or set members of data structure independent of any compiled in data structure such as a C struct or FORTRAN common block. This facility is completely dynamic and interpreted at runtime.

Basic Functions

PA_mk_instance

PA_rl_instance

PA_get_member

PA_set_member

PA_MK_INSTANCE

```
void *PA_mk_instance(char *name,
                     char *type,
                     long n)
```

Allocate and return a pointer to an array of instances of a type (defined in the virtual internal file) which is n items long.The type must have been defined from a dictionary file or from an explicit function call.

The return value of this function is a pointer to the space allocated to hold n elements each of which is of sufficient byte size to hold one item of type type. The instance is also installed in an internal table under the name name so that PANACEA can track the variable.

PA_RL_INSTANCE

```
void PA_rl_instance(char *name)
```

Release the named instance of an entry allocated with PA_mk_instance.

This function has no return value.

PA_GET_MEMBER

```
void *PA_get_member(char *name,
                    char *member)
```

Get the named member of the instance of a structured variable allocated by PA_mk_instance under the name name.

A pointer to the member is returned if successful and NULL is returned otherwise.

PA_SET_MEMBER

```
void PA_set_member(char *name,
                   void *data,
                   char *member)
```

Set the named member of an instance of a data structure return by PA_mk_instance to point to the given data. The instance is identified by name.

This function returns nothing.

Variable Access for Output

Optional Functions

- PA_sub_select(PA_variable *pp, C_array *arr, long *pitems, long *poffs, long *pstr)
- PA_general_select(PA_variable *pp, C_array *arr, unsigned long *pitems, unsigned long *pdims, unsigned long *poffs, unsigned long *pstr, unsigned long *pmax)

Simulation Control

The functions in the section describe the services PANACEA offers the simulation phase code. These are fairly high level functions which encapsulate much of the detailed inner workings of the PANACEA database and the PANACEA packages.

Basic Functions

PA_rd_restart

PA_wr_restart

PA_init_system

PA_terminate

Optional Functions

PA_simulate

PA_run_packages

PA_fin_system

PA_advance_t

PA_RD_RESTART

```
void PA_rd_restart(char *rsname,
                   int convs)
```

This function reads a state file as a prelude to performing simulations starting from the state specified in the file. Statistics about the amount of data in the state file which is actually loaded into memory are printed

The arguments are: rsname, the name of the state file; and convs, the type of conversions to do. The conversion options are:

NONE	perform no conversions
INT_CGS	convert from internal units to CGS units
INT_EXT	convert from internal units to external units
EXT_CGS	convert from external units to CGS units
EXT_INT	convert from external units to internal units
CGS_INT	convert from CGS units to internal units
CGS_EXT	convert from CGS units to external units

The internal system of units is defined by the unit array. The external system of units is defined by the convrsn array.

The system of units of the data in the state file is under the control of the code developer, but it must be consistent.

This function returns no value.

SEE ALSO:PA_wr_restart

PA_SIMULATE

```
void PA_simulate(double tc,
                 int nc,
                 int nz,
                 double ti,
                 double tf,
                 double dtf_init,
                 double dtf_min,
                 double dtf_max,
                 double dtf_inc,
                 char *rsname,
                 char *edname,
                 char *ppname,
                 char *pvname)
```

This function is provided as a template for controlling and coordinating the execution of the packages and other PANACEA services. It is clear that no one routine can satisfy all of the requirements of all simulation systems, and this routine does not really attempt to do so. It is provided as an example to be used in constructing the simulation control for any given application code and to be used by those applications for which it is sufficient.

The arguments are: tc, the current problem time; nc, the current problem cycle; nz, the number of zones/cells/particles in the problem; ti, the starting time; tf, the stopping time; dtf_init, the initial fractional time step; dtf_min, the minimum fractional time step; dtf_max, the maximum, fractional time step; dtf_inc, the fractional increase in the time step each cycle; rsname, the name of the most recently read state file; edname, the name of the ASCII edit file; ppname, the name of the time history file; and pvname, the name of the PVA file. Each file name is a base name ending with two digits. Each time a new file in each of the respective families is closed, the next member of the family is given a name with the number indicated by the final digits advanced by one.

PA_simulate performs the following actions in a loop over time from ti to tf:

1. calls PA_source_variable to update all source variables in the simulation for the current time and time step

2. calls PA_run_package to execute each installed package in the order of installation

3. calls PA_dump_pp to check all plot requests and write out any requested data in the time history and PVA files for the current time and cycle

4. calls PA_file_mon to determine whether the current edit, time history, or PVA file need to be closed and the next member of the family created

5. calls PA_advance_t to poll the packages for their time step votes and determine the next time step

At the conclusion of the loop over time, PA_simulate, prints a message to the terminal announcing that the stop time has been reached, and then it calls PA_fin_system to call the package fzr routines and print the simulation statistics to the terminal.

This routine has no return value.

SEE ALSO:PA_source_variable, PA_run_packages, PA_dump_pp, PA_file_mon, PA_advance_t, PA_fin_system.

PA_RUN_PACKAGES

```
void PA_run_packages(double t,
                     double dt,
                     int cycle)
```

This routine executes the main entry point of each installed package and dumps any information corresponding to a plot request associated with the package.

The arguments are: t, the current problem time; dt, the current problem time step; and cycle, the current problem cycle number. This information is passed to the package’s main entry point via the package structure.

Prior to executing the package main, the memory for PSEUDO EDIT variables is allocated. After the package main is finished, the PSEUDO EDIT variable data are dumped and the memory allocated to it is released.

This function has no return value.

SEE ALSO:PA_simulate

PA_FIN_SYSTEM

```
void PA_fin_system(int nz,
                   int nc)
```

This function runs the finalizer routine for each installed package and prints performance statistics on the packages for the simulation (or partial simulation) just concluded. The performance statistics are based on the CPU time spent in the package as reported by the package and the storage space used by the package in kBytes. Whether this information is gathered is up to the package developer. If no information is stored in a package, PANACEA does not print any statistics for that package. The macros PA_MARK_TIME, PA_ACCM_TIME, PA_MARK_SPACE, and PA_ACCM_SPACE can be used by the package developer to save the timing and memory usage information in the package structure.

The arguments are: nz, the number of zones/cells/particles in the problem; and nc the number of cycles run.

This function has no return value.

SEE ALSO:PA_simulate, PA_MARK_TIME, PA_ACCM_TIME, PA_MARK_SPACE, and PA_ACCM_SPACE

PA_ADVANCE_T

```
double PA_advance_t(double dtmn,
                    double dtn,
                    double dtmx)
```

This function computes a time step for the next major computational cycle based on the time step "votes" returned in each package after the package main entry point is executed. The algorithm is that the smallest time step from the packages is computed, the smaller of this time step, dtn, and dtmx is taken; and the larger of dtmn and the result of the previous step is returned.

The arguments are: dtmn, the minimum allowed time step, dtn, the current time step; and dtmx, the maximum allowed time step.

The computed new time step is returned by this function.

SEE ALSO:PA_simulate

PA_WR_RESTART

```
void PA_wr_restart(char *rsname)
```

This function causes a state file to be written out. The name of the file is specified by rsname and after the file is closed successfully, the name is incremented. By convention files which PANACEA writes have names ending with two digits. When PANACEA increments a file name it adds one to the number represented by the two digits and replaces them in the file name.

All of the variables defined to PANACEA as RESTART or DEFN are written out to the state file. They should be the variables required to define the state so that a future run of the simulation code can start up with only the state file's information.

The argument is: rsname, the name to be used for the state file.

SEE ALSO:PA_rd_restart

PA_INIT_SYSTEM

```
void PA_init_system(double t,
                    double dt,
                    int nc,
                    char *edname,
                    char *ppname,
                    char *pvname)
```

This function initializes a PANACEA simulation phase code. It is not for generation phase programs. It performs the following actions:

1. opens any specified source files and initializes the appropriate source variables
2. opens and initializes the specified ASCII edit file, time history file, and PVA file
3. runs the initializer function for each package
4. processes the plot requests for run time efficiency
5. calls PA_dump_pp to make an initial dump of the information specified by the plot requests

The arguments to this function are: t, the initial problem time; dt, the initial problem time step; nc, the initial problem major cycle number; edname, the name of the ASCII edit file; ppname, the name of the time history file; and pvname, the name of the PVA file.

This function has no return value.

SEE ALSO:PA_dump_pp

PA_TERMINATE

```
void PA_terminate(char *edname,
                  char *ppname,
                  char *pvname,
                  int cycle)
```

This function gracefully shuts down a simulation run. It closes any open ASCII edit files, time history files, PVA files, cache files, or state files. If this function is not called, these files may not be valid data files!

The arguments are: edname, the name of the ASCII edit file; ppname, the name of the time history file; pvname, the name of the PVA file; and cycle, the current problem cycle. These arguments are currently unused, however, that is subject to change.

This function returns nothing.

Plot Request Handling

Optional Functions

- PA_time_plot(char *rname, void *vr)
- PA_dump_time(PA_set_spec *pi, double tc, double dtc, int cycle)
- PA_put_set(PDBfile *file, PM_set *s)
- PA_put_mapping(PG_device *dev, PDBfile *file, PM_mapping *f, int plot_type)
- PA_set_data(char *name, C_array *arr, int *pcent)
- PA_fill_component(double *data, int len, int *pist, int ne)
- PA_build_mapping(PA_plot_request *pr, PFPPM_set build_ran, double t)
- PA_non_time_domain(PA_plot_request *pr)

- PA_PR_RANGE_SIZE(PA_plot_request *pr, long n)
  Reset the size of the plot request range.

- PA_STASH_SP_SCALAR(char *name, long indx, double val)
  Save a scalar value for a spatial pseudo variable plot request.

- PA_STASH_TV_SCALAR(char *name, double val)
  Save a scalar value for a time pseudo variable plot request.

- PA_MESH_RANGE_SEARCH(PA_plot_request *pr, long off, char *fn)
  Loop the range specification of the plot request to setup an invocation of PA_STASH_SP_SCALAR.

- PA_TIME_RANGE_SEARCH(PA_plot_request *pr, char *fn)
  Loop the range specification of the plot request to setup an invocation of PA_STASH_TV_SCALAR.

- PA_END_MESH_SEARCH(PA_plot_request *pr, expr)
  Gracefully terminate the loop started by PA_MESH_RANGE_SEARCH.

- PA_END_TIME_SEARCH(PA_plot_request *pr)
  Gracefully terminate the loop started by PA_TIME_RANGE_SEARCH.

- PA_STORE_TV(PA_plot_request *pr, double val)
  Save away the given time value data.

Generation Support

Generator Command Management

Basic Functions

PA_inst_com

PA_inst_pck_gen_cmmnds()

PA_inst_c(char *cname, byte *cvar, int ctype, int cnum, PFVoid cproc, PFVoid chand)

Optional Functions

PA_get_commands

PA_INST_COM

```
HASHTAB *PA_inst_com(void)
```

Initialize and set up the table of commands needed to parse an ASCII input file and generate an initial state for a simulation. This function does the following things:

- Installs the generic commands
- Installs the package generator commands (see the gencmd argument of the PA_gen_package function)
- Calls PA_definitions to define the unit and conv arrays for PANACEA.
- Calls PA_variables to complete the definition of the database variables (the units are not completed with this call since the input deck may redefine the unit system).

The generic PANACEA supplied generator commands are:

end	end the generation run
read	read the named input file
dump	dump a state file
specify	specify an initial value condition
s	continue an initial value condition specification
graph	make a plot request
package	set the current package
switch	set the value of a switch in the current package
parameter	set the value of a switch in the current package
name	set the value of a name in the current package
unit	set an internal conversion factor (sets the internal system of units)
conversion	set an external conversion factor (sets the external system of units)

PA_GET_COMMANDS

```
PA_get_commands(FILE *fp, void (*errfnc)())
```

This function reads from the specified input stream, parses command lines, and dispatches to functions installed in the PANACEA command table via the PA_inst_c function.

It executes the following steps in a loop:

- Get a line of text
- If no more text is available close the stream (if it is not stdin) and return.
- If the line is a blank or comment line loop
- Lookup up the first token in the PANACEA command table
- If there is an associated function dispatch to it
- If there is no associated function call the errfnc from the argument list.

Generation Time Functions

Basic Functions

PA_name_files(char *base_name, char **ped, char **prs, char **ppp, char **pgf)

Optional Functions

PA_clear()

Source Variable/Initial Value Data Handling

PANACEA provides functionality for handling data which is to be used as boundary condition or source information or to set values of database variables. One application of this capability is to use the results of one simulation to drive a second simulation whose intent is to simulate one phenomenon in an environment which itself is the result of a simulation. To emphasize the inherent unity of sources and to provide a simple mechanism for applications, PANACEA defines the concept of an initial value specification. Initial value specifications are made in the ASCII input defining a problem with the specify command. The data may be provided with the specify or s commands. It may also be provided from a source file which is produced by a utility which the simulation system designer must write. This utility uses PANACEA services to format data in such a way that PANACEA can access it quickly and efficiently at run time. This is also done because it is impossible for PANACEA to know about even the smallest fraction of potential sources for initial value data and their formats.

An initial value specification is the mathematical notion of initial value data which together with a set of differential equations defines an initial value problem. PANACEA defines a structure called a PA_iv_specification to represent an initial value specification. In particular, an initial value specification has:

1. a type: bc, for boundary condition which specifies information on the boundary of the computational mesh; src, for source which specifies a set of values imposed in some region of the interior of the computational mesh; or the name of a database variable whose values throughout the simulation are interpolated from a set of predetermined data

2. a name by which the initial value specification may be selected

3. a file name in which the actual initial value data may be found

4. a flag specifying whether the data is to be interpolated or treated as discrete sets that are to be used as the exact specified times

5. the number of data points in one time slice of the data set

6. various representations of any data that exists in memory (PANACEA manages this with the aim of handling this data as efficiently as possible with respect to both time and space)

To help manage initial value specifications whose data is to be imported from a source file, PANACEA defines the concept of a source variable and a structure called a PA_src_variable. The source variable contains information about the data in a source file and the data structures necessary to manage the interpolations which may be done by PANACEA. Source variables must have counterparts in the database. They fundamentally are devices to help load the appropriate initial value data into a database variable.

The following functions are all optional in that an application can use PANACEA without using the initial value handling services of the library. If the application does use these services, the basic operations at the application level involve accessing initial value data and interpolating values from initial value data structures. There is also a more global operation which tells PANACEA to update all sourced variables.

Optional Functions

PA_get_iv_source

PA_get_source

PA_interp_src

PA_intr_spec

PA_source_variables

PA_GET_IV_SOURCE

PA_iv_specification *PA_get_iv_source(char *name)

This routine searches the list of initial value specifications and returns the named PA_iv_specification. In order for this function to succeed, the given name must match the name used in at least one specify command in the problem definition.

Input to this function is: name, an ASCII string naming the initial value specification for which to search.

Output from this function is a pointer to a PA_iv_specification if successful and NULL otherwise.

PA_GET_SOURCE

PA_src_variable *PA_get_source(char *s, int start_flag)

This function returns a pointer to a source variable specified by the name, s. It is permissible to have many instances of a source variable in a source file or to have many source files with the same source variable. In such a case each call to PA_get_source returns the next available instance of the source variable. If the start_flag is TRUE the first instance is found and returned.

Input to this function is: s, an ASCII string naming the source variable; and start_flag, an integer signalling whether to start from the beginning of the source variable list or from the last position searched.

Output from this function is a pointer to a PA_src_variable if successful and NULL otherwise.

PA_INTERP_SRC

```
void PA_interp_src(void *v,
                   PA_src_variable *svp,
                   int ni,
                   int nf,
                   double t,
                   double dt)
```

This function interpolates the data from the given PA_src_variable into the space pointed to by v from v[ni] to v[nf]. The data is interpolated in time at time t with timestep dt. The interpolation is quadratic.

The input to this function is: v, a pointer to the target array; svp, a pointer to a PA_src_variable containing the source data; ni, an integer starting index into v; nf, an integer ending index into v; t, a double value specifying the interpolation time; and dt, a double value specifying the time step.

This function has no return value.

PA_INTR_SPEC

```
double PA_intr_spec(PA_iv_specification *sp,
                    double t,
                    double val,
                    long off)
```

This function finds the interpolated source value from the given initial value specification information. For efficiency sake when interpolating the initial value data, the index into the data set from each call is saved and used as the starting point for the next interpolation. This means that the initial value data is assumed to be stored in increasing time order.

The input to this function is: sp, a pointer to a PA_iv_specification containing the initial value data to be interpolated; t, a double containing the time value at which the interpolated value is required; val, a double containing a default value if there is no data in the initial value data set for the given time, t; and off, a long offset telling how much of the initial value data to skip past (the initial value data may be prefaced with parameters describing the data that follows).

The return value of this function is the interpolated value of the initial value data set.

PA_SOURCE_VARIABLES

```
void PA_source_variables(double t,
                         double dt)
```

This function sets the values (by interpolation) of the database variables which have been identified with source variables. This function is called once per major time cycle in PA_simulate. Those applications controlling their own time stepping must call this function if they wish to have PANACEA handle source data. In PA_simulate, this function is called before the packages are executed.

This function performs the following steps:

1. Loops over each source variable set up by PA_init_system.

2. If the database variable corresponding to the source variable does not exist or has a NULL data pointer no further action is performed.

3. PA_interp_src is called to interpolate the source variable data in time and fill the data associated with the database variable.

The input to this function is: t, a double containing the time at which the source variable are to be interpolated; and dt, a double containing the corresponding time step.

This function returns no value.

Time History Data Management

The following functions define a service for the management of time history data.

This facility implements a time history data management model designed with the following goals:

- Easy to use
- Flexible enough to accommodate virtually any code system requirements
- Support families of files
- Support restart from any cycle in a simulation (the instance index is the key to this)
- The raw time history data should also be comprehensible.

The model chosen centers on the definition of a time history data structure which is defined at run time. Each such structure has a single domain component and multiple range components. These correspond to a set of time plots for which data is written at a particular frequency as determined by the application. By defining however many such structures are required, code systems can manage many sets of time plots each at their own frequency. New sets can be added over the course of a simulation. These structures are set up via the PA_th_def_rec call. Once defined, the application can buffer up any amount of time history data and then write it out with PA_th_write. Thus applications determine how much data will be kept in the running code.

The time history data is written out in families PDB of files. A family of time history PDB files is started with a call to PA_th_open. The function PA_th_family is used to close out one member of the family and start the next if the current file's size is above a user set threshold. Each file in a family maintains a link to the previous file in the family. This is very useful when applications restart a simulation from an earlier state and, potentially, follow a different branch of execution. Knowing the last time history file in any branch allows the unambiguous reconstruction of the time history data for that branch.

In addition to the time history data, any sort of associated data may be defined via the attribute mechanism in PDBLib. Additionally attributes may be assigned to particular instances of time history data structures via the PA_th_wr_iattr call.

Since PANACEA is a part of PACT, it supplies functions to transpose the time history data files into ULTRA files (also a PACT tool). These functions are: PA_th_trans_name; PA_th_trans_link; and PA_th_trans_family. In this way, applications may directly transpose their time history files if desired. The stand alone program TOUL which transposes a family of time history files into a family of ULTRA files simply calls these functions appropriately. NOTE: all files in the family must be closed before any transpose operation. Functions to merge an arbitrary list of ULTRA files or a family of ULTRA files are also provided. These functions are: PA_merge_files and PA_merge_family.

This facility is substantially independent of the rest of PANACEA and can be used without any other parts of PANACEA.

Basic Functions

PA_th_open, PA_th_def_rec, PA_th_write, PA_th_family

Optional Functions

PA_merge_family, PA_merge_files, PA_th_trans_name, PA_th_trans_link, PA_th_trans_family, PA_th_wr_iattr

PA_MERGE_FAMILY

```
int PA_merge_family(char *base,
                    char *family,
                    int nc)
```

Merge a family of ULTRA files into a single file or family of files. The ULTRA source files are specified here by family base name.

ULTRA source files are assumed to follow the naming convention

family.udd

where family is the base source file name supplied and dd are base 36 digits (0-9a-z).

ULTRA target files produced by this function similarly follow the naming convention

base.udd

where base is the base target file name supplied.

Argument nc, determines how target files are familied by specifying the approximate number of curves per file. However, regardless of the value of nc, all curves from a given source file will reside in a single target file. If nc is zero, all curves will be merged into a single file.

For efficiency this function currently assumes that all curves in a given source file share the domain of the first curve in that file. This is true for ULTRA files produced by the PANACEA time history transpose routines. Curves from arbitrary ULTRA files can be merged, albeit less efficiently, using the save command in the ULTRA utility.

Input to this function is: base, an ASCII string containing the base target file name; family, an ASCII string containing the base source file name; and nc an integer approximate number of curves per ULTRA target file.

TRUE is returned if the call is successful; otherwise, FALSE is returned and the ASCII string PD_err contains any error message that was generated.

See also PA_merge_files.

Example:

```
#include "panace.h"
                 .
                 .
                 .
/* merge files in family foo */
           if (!PA_merge_family("bar", "foo", 0))
              printf(stderr, "%s\n", PD_err);
                 .
                 .
                 .
```

PA_MERGE_FILES

```
int PA_merge_files(char *base,
                    int n,
                    char **names,
                    int nc)
```

Merge a set of ULTRA files into a single file or family of files. The ULTRA source files are specified here explicitly by name.

ULTRA target files produced by this function follow the naming convention base.udd

where base is the base target file name supplied and dd are base 36 digits (0-9a-z).

Argument nc, determines how target files are familied by specifying the approximate number of curves per file. However, regardless of the value of nc, all curves from a given source file will reside in a single target file. If nc is zero, all curves will be merged into a single file.

For efficiency this function currently assumes that all curves in a given source file share the domain of the first curve in that file. This is true for ULTRA files produced by the PANACEA time history transpose routines. Curves from arbitrary ULTRA files can be merged, albeit less efficiently, using the save command in the ULTRA utility.

Input to this function is: base, an ASCII string containing the base file name; n, an integer number of file names supplied in names; names, an array of ASCII strings containing the full names of the ULTRA source files; and nc an integer approximate number of curves per ULTRA target file.

TRUE is returned if the call is successful; otherwise, FALSE is returned and the ASCII string PD_err contains any error message that was generated.

See also PA_merge_family.

Example:

```
#include "panace.h"

           char **names;
                  .
                  .
                  .
       /* only merge foo.u09 through foo.u0b */
           names    = MAKE_N(char *, 3);
           names[0] = SC_strsave("foo.t09");
           names[1] = SC_strsave("foo.t0a");
           names[2] = SC_strsave("foo.t0b");
           if (!PA_merge_files("bar", 3, names, 0))
              printf(stderr, "%s\n", PD_err);
           SFREE(names[0]);
```

```
SFREE(names[1]);
SFREE(names[2]);
SFREE(names);
       .
       .
       .
```

PA_TH_DEF_REC

```
defstr *PA_th_def_rec(PDBfile *file,
                      char *name,
                      char *type,
                      int nmemb,
                      char **members,
                      char **labels)
```

Define a special type to be used for the purpose of gathering time history data in such a fashion that a generic tool (e.g. TOUL) can be used to convert the data into arrays of time ordered data. This defines both a derived data type and an entry in the symbol table of the file to contain the data. PANACEA implicitly adds another entry which contains information that a transposer would use.

The members must follow the rules for member specifications as described in the PDBLib User’s Manual with the exception that if no type is specified the type is taken to be "double". Briefly, a member specification consists of a type, an identifier, and optional dimension specifications. Member identifiers must not contain white space characters or any of the following: ‘*’, ‘[‘, ‘]’, ‘(‘, ‘)’, or ‘.’. The first member is taken to be the domain data (x value) and the remaining members are taken to be ranges. Each data set of the specified type transposes into nmemb-1 curves, that is one x array and nmemb-1 y arrays.

NOTE: A non-scalar member does not transpose into a set of curves. In such a case PDBView is used directly to manipulate the time history data since it has superior capabilities for allowing users to control the semantics of their data.

Since a very common use for this capability is to generate curves for plotting purposes, the labels argument is provided for the application to supply curve labels. The labels argument may be NULL if the application wishes PANACEA to construct labels from the member.

Input to PA_th_def_rec is: file, a pointer to a PDBfile; name, an ASCII string containing the name of the data entry in the symbol table; type, an ASCII string containing the name of the new struct; nmemb, an integer specifying the number of members of the new type; members, an array of nmemb ASCII strings specifying the struct members; and labels, an array of nmemb-1 ASCII strings specifying labels for curves generated by the transposer.

A pointer to the type's defstr is returned if the call is successful; otherwise, NULL is returned and the ASCII string PD_err contains any error message that was generated.

See also PA_th_wr_iattr, PA_th_write, and PA_th_trans_family.

Example:

```
#include "panace.h"

   PDBfile *file;
   defstr *ptr;
   char **members;
          .
          .
          .
    members = MAKE_N(char *, 3);
    members[0] = SC_strsave("time");
    members[1] = SC_strsave("d[20]");
    members[2] = SC_strsave("p at x,y");
    ptr = PA_th_def_rec(file, "t-data", "t-struct", 3, members, NULL);

    SFREE(members[0]);
    SFREE(members[1]);
    SFREE(members[2]);
    SFREE(members);
          .
          .
          .
```

PA_TH_FAMILY

```
PDBfile *PA_th_family(PDBfile *file)
```

Check the current file’s size against the maximum size as specified in the PA_th_open call. If the file is larger than the maximum size, close the file, open the next member of the file family, and return a pointer to the new file.

Input to this function is: file, a pointer to a PDBfile.

A pointer to a PDBfile is returned if the call is successful; otherwise, NULL is returned. The PDBfile pointer that is returned should be checked against the one passed in to tell whether a new file has been opened.

See also PA_th_def_rec, PA_th_write, and PA_th_open.

Example:

```
#include "panace.h"

          PDBfile *oldfile, *newfile;
                 .
                 .
                 .
           newfile = PA_th_family(oldfile);
           if (newfile != oldfile)
              printf("Next member of family opened\n");
                 .
                 .
                 .
```

PA_TH_OPEN

```
PDBfile *PA_th_open(char *name,
                    char *mode,
```

```
long size,
char *prev)
```

Open a new time history data file. This implicitly defines a family of files. The name should be of the form: base.tdd where d is a base 36 digit (i.e. 0-9a-z). This is only a convention, but there are certain consequences for not following it in as much as the familying mechanism assumes that the last two characters form a base 36 number and increments it accordingly. As an application writes data to a time history file, periodic calls to PA_th_family should be made to monitor the file size and when necessary close the current family member and open the next.

Since simulations may be restarted and each code may have a different strategy for continuing time history collection in the event of a restart, it is necessary to allow for name changes in the family. The consequence of this is that each member of a file family must contain the name of the previous file in the family. In that way, the transpose process may unambiguously and under the control of the user or simulation follow a chain of time history files from the end point back to the beginning. The prev argument is used to supply this information. The family of files that follows will be in sequence from the name supplied. Only across restarts, which implies calls to PA_th_open, may the sequence name be changed. A call to PA_th_open may have NULL for prev which indicates the absolute beginning of the sequence, i.e. the transpose will stop in its search for files at this point.

Input to this function is: name, an ASCII string specifying the name of the first file in the family for the current application; mode, an ASCII string specifying the mode of the file (typically this will be "w"); size, a long integer value specifying the target maximum file size for members of the family; and prev, an ASCII string specfiying the name of the previous file in the family sequence.

A pointer to a PDBfile is returned if the call is successful; otherwise, NULL is returned.

See also PA_th_def_rec, PA_th_write, and PA_th_family.

Example:

```
#include "panace.h"

        PDBfile *newfile;
               .
               .
               .
         newfile = PA_th_open("foo.t00", "w", 1000000L, NULL);
         if (newfile == NULL)
```

```
                   printf("Can't open time history file\n");
                      .
                      .
                      .
```

PA_TH_TRANS_FAMILY

```
int PA_th_trans_family(char *name,
                       int ord,
                       int nc)
```

Write an ULTRA file by transposing the time history data from a given set of time history files.

A family of time history files is conventionally named

name.tdd

where name is the base file name and dd are base 36 digits (0-9a-z).

The resulting family of ULTRA files will be named

name.udd

There is the possibility inherent in this model of handling file families that two or more files to be processed will have data for some regions in time. Because of the underlying PDB machinery, the data from the last file processed will predominate. Also, since the files are linked together from latest to earliest, it is sometimes necessary to require that the files be processed in the opposite order in which they are specified. The ord argument is used for this purpose. A value of 1 causes the files to be processed in the order in which they are specified while a value of -1 causes them to be processed in reverse order.

Input to this function is: name, an ASCII string containing the base name of the time history file family; ord, an integer flag specifying the order of processing; and nc a maximum number of curves per file in the ULTRA file family.

TRUE is returned if the call is successful; otherwise, FALSE is returned and the ASCII string PD_err contains any error message that was generated.

See also PA_th_trans_link, PA_th_trans_name, PA_th_def_rec, and PA_th_write.

Example:

```
#include "panace.h"

                 .
                 .
                 .
            if (!PA_th_trans_family("foo", 1, 1000))
               printf(stderr, "%s\n", PD_err);
                 .
                 .
                 .
```

PA_TH_TRANS_LINK

```
int PA_th_trans_link(int n,
                     char **names,
                     int ord,
                     int nc)
```

Write an ULTRA file by transposing the time history data from a specified set of time history files. The time history files are specified here by giving the names of files at the end of file family chains.

On occasion, files in a family are lost. This breaks the chain of files as well as leaving gaps in the data. Since this function can take many file names, it can be used to take into account missing files by supplying the files at the top of the gap(s) of missing files. In the accompanying example it is assumed that the files foo.t09 through foo.t11 are missing.

There is the possibility inherent in this model of handling file families that two or more files to be processed will have data for some regions in time. Because of the underlying PDB machinery, the data from the last file processed will predominate. Also, since the files are linked together from latest to earliest, it is sometimes necessary to require that the files be processed in the opposite order in which they are specified. The ord argument is used for this purpose. A value of 1 causes the files to be processed in the order in which they are specified while a value of -1 causes them to be processed in reverse order.

A family of time history files is conventionally named

name.tdd

where name is the base file name and dd are base 36 digits (0-9a-z).

The resulting family of ULTRA files will be named

name.udd

Input to this function is: n, an integer number of file names supplied in names; names, an array of ASCII strings containing the full names of the time history files; ord, an integer flag specifying the order of processing; and nc a maximum number of curves per file in the ULTRA file family.

TRUE is returned if the call is successful; otherwise, FALSE is returned and the ASCII string PD_err contains any error message that was generated.

See also PA_th_trans_name, PA_trans_family, PA_th_def_rec, and PA_th_write.

Example:

```
#include "panace.h"


           char **names;
                 .
                 .
                 .
       /* files foo.t09 through foo.t11 are missing */
           names    = MAKE_N(char *, 2);
           names[0] = SC_strsave("foo.t1f");
           names[1] = SC_strsave("foo.t08");
           if (!PA_th_trans_link(2, names, -1, 1000))
              err_proc();
           SFREE(names[0]);
           SFREE(names[1]);
           SFREE(names);
                 .
                 .
                 .
```

PA_TH_TRANS_NAME

```
int PA_th_trans_name(int n,
                     char **names,
                     int ord,
                     int nc)
```

Write an ULTRA file by transposing the time history data from a specified set of time history files. The time history files are specified here explicitly by name.

On occasion, it is desirable to transpose only selected files from a family. For example, a user may know that only certain times are of interest and doesn't wish to see the entire data set. This function gives the finest level of control to the application about which files to transpose.

There is the possibility inherent in this model of handling file families that two or more files to be processed will have data for some regions in time. Because of the underlying PDB machinery, the data from the last file processed will predominate. Also, since the files are linked together from latest to earliest, it is sometimes necessary to require that the files be processed in the opposite order in which they are specified. The ord argument is used for this purpose. A value of 1 causes the files to be processed in the order in which they are specified while a value of -1 causes them to be processed in reverse order.

A family of time history files is conventionally named

name.tdd

where name is the base file name and dd are base 36 digits (0-9a-z).

The resulting family of ULTRA files will be named

name.udd

Input to this function is: n, an integer number of file names supplied in names; names, an array of ASCII strings containing the full names of the time history files; ord, an integer flag specifying the order of processing; and nc a maximum number of curves per file in the ULTRA file family.

TRUE is returned if the call is successful; otherwise, FALSE is returned and the ASCII string PD_err contains any error message that was generated.

See also PA_th_trans_link, PA_trans_family, PA_th_def_rec, and PA_th_write.

Example:

```
#include "panace.h"

          char **names;
                .
                .
                .
        /* only transpose foo.t09 through foo.t0b */
          names    = MAKE_N(char *, 3);
          names[0] = SC_strsave("foo.t09");
          names[1] = SC_strsave("foo.t0a");
          names[2] = SC_strsave("foo.t0b");
          if (!PA_th_trans_name(3, names, 1, 1000))
             err_proc();
          SFREE(names[0]);
          SFREE(names[1]);
          SFREE(names[2]);
          SFREE(names);
                .
                .
                .
```

PA_TH_TRANSPOSE

```
int PA_th_transpose(char *name,
                    int nc)
```

WARNING: This is a deprecated function. Use PA_th_trans_family instead.

Write an ULTRA file by transposing the time history data from a given set of time history files.

A family of time history files is conventionally named

name.tdd

where name is the base file name and dd are base 36 digits (0-9a-z).

The resulting family of ULTRA files will be named

name.udd

Input to this function is: name, an ASCII string containing the base name of the time history file family; and nc a maximum number of curves per file in the ULTRA file family.

TRUE is returned if the call is successful; otherwise, FALSE is returned and the ASCII string PD_err contains any error message that was generated.

See also PA_th_trans_link, PA_th_trans_name, PA_th_def_rec, and PA_th_write.

Example:

```
#include "panace.h"

                 .
                 .
                 .
           if (!PA_th_transpose("foo", 1000))
              printf(stderr, "%s\n", PD_err);
                 .
                 .
                 .
```

PA_TH_WR_IATTR

```
int PA_th_wr_iattr(PDBfile *file,
                   char *var,
                   int inst,
                   char *attr,
                   void *avl)
```

Assign an attribute value to an instance of a time history data record.

Input to this function is: file, a pointer to a PDBfile; var, an ASCII string containing the name of the data entry in the symbol table; inst, an integer instance index; attr, an ASCII string containing the name of the attribute; and avl, a pointer to the attribute value.

TRUE is returned if the call is successful; otherwise, FALSE is returned and the ASCII string PD_err contains any error message that was generated.

See also PA_th_def_rec, PA_th_write, and PA_th_trans_family.

Example:

```
#include "panace.h"

          PDBfile *file;
          double *vr
                 .
                 .
                 .
           vr = MAKE_N(double, 9);
                 .
                 .
                 .
           if (!PA_th_wr_iattr(file, "t-data", 3, "date", "today"))
              printf(stderr, "%s\n", PD_err);
                 .
                 .
                 .
```

PA_TH_WRITE

```
int PA_th_write(PDBfile *file,
                char *name,
                char *type,
```

```
int inst,
int nr,
void *vr)
```

Write out nr instances of time history data whose structure has been defined by PA_th_def_rec. This function writes out nr complete instances of a time history record! Using this function an application can manage multiple sets of time history data which are written at different frequencies.

The name and type arguments should match those used in defining the structure with PA_th_def_rec.

Input to this function is:

file	a pointer to a PDBfile
name	an ASCII string containing the name of the data entry in the symbol table
type	an ASCII string containing the name of the time history domain struct
inst	an integer specifying the instance index of the first record in this call
nr	an integer specifying the number of instances of the structure pointed to by vr
vr	a pointer to the data

TRUE is returned if the call is successful; otherwise, FALSE is returned and the ASCII string PD_err contains any error message that was generated.

See also PA_th_wr_iattr, PA_th_def_rec, and PA_th_trans_family.

Example:

```
#include "panace.h"


         PDBfile *file;
         double *vr
                .
                .
                .
          vr = MAKE_N(double, 9);
                .
                .
```

```
        .
if (!PA_th_write(file, "t-data", "t-struct", 0, 3, vr))
   printf(stderr, "%s\n", PD_err);
        .
        .
        .
if (!PA_th_write(file, "t-data", "t-struct", 3, 2, vr))
   printf(stderr, "%s\n", PD_err);
        .
        .
        .
if (!PA_th_write(file, "t-data", "t-struct", 5, 3, vr))
   printf(stderr, "%s\n", PD_err);
        .
        .
        .
```

PA_TH_WR_MEMBER

```
int PA_th_wr_member(PDBfile *file,
                    char *name,
                    char *member,
                    char *type,
                    int inst,
                    void *vr)
```

Write out one member of a time history data record whose structure has been defined by PA_th_def_rec. This function writes out one member of a time history record. Its utility is most apparent when the member is an array which could be large or is part of a very large record.

Input to this function is:

file	a pointer to a PDBfile
name	an ASCII string containing the name of the data entry in the symbol table
member	an ASCII string containing the name of the time history record member
type	an ASCII string containing the name of the time history domain struct
inst	an integer specifying the instance index of the first record in this call
vr	a pointer to the data

See also PA_th_wr_iattr, PA_th_def_rec, and PA_th_trans_family.

Example:

```
#include "panace.h"

          PDBfile *file;
          double *vr
                 .
                 .
                 .
           vr = MAKE_N(double, 10000);
                 .
                 .
                 .
           if (!PA_th_wr_member(file, "t-data", "v_1", "t-struct", 1, vr))
              printf(stderr, "%s\n", PD_err);
                 .
                 .
                 .
```

Miscellaneous

Parsing Support

Optional Functions

- PA_get_field(char *s, char *t, int optp)
- PA_stof(char *s)
- PA_stoi(char *s)

Array Operations

Optional Functions

- PA_scale_array(double *p, int n, double f)
- PA_set_value(double *p, int n, double f)
- PA_copy_array(double *s, double *t, int n)
- PA_index_min(double *p, int n)
- PA_find_index(double *p, double f, int n)
- PA_max_value(double *p, int imin, int n)

Error Handling

Optional Functions

- PA_ERR(int test, char *fmt, ...)

Timing Routines

Optional Functions

- PA_MARK_TIME
  Mark the current cpu time.

- PA_ACCM_TIME(double x)
  Accumulate the time since the previous MARK_TIME.

Comparison Routines

Optional Functions

- PA_SAME(double x1, double y1, double x2, double y2)
  Return TRUE if the points are within TOLERANCE distance of each other.

- PA_CLOSETO(double x1, double x2)
  Return TRUE if the numbers are within TOLERANCE of each other.

The PANACEA FORTRAN API

The FORTRAN interface to PANACEA has some global conventions. First, the function names adhere strictly to the FORTRAN77 naming convention of six characters. The names all begin with "PA" to identify them with PANACEA.

Most functions return the value 1 if they complete without error and 0 otherwise.

As in the C interface, the descriptions of the functions are sorted into service categories as follows:

Database Definition and Control

Variable Definition

Control Definition

Unit Definition and Control

Database Access

Simulation Control

Plot Request Handling

Initial Value Problem Generation Support

Source Variable/Initial Value Data Handling

Time History Data Management

Miscellaneous

Within service category commands are segregated into two groups: Basic, PANACEA applications must invoke these functions at some appropriate point; and optional, these are provided for the benefit of the application but are not required.

Database Definition and Control Functionality

These functions provide for overall control of the PANACEA database and packages.

Basic Functions

PASPCK

Optional Functions

PAGNSP

PAGNSP(integer flag)

Returns the value of the name space flag in the argument flag. See the PASNSP function for further information.

PASNSP

PASNSP(integer flag)

If flag is TRUE then the name space of PANACEA variables is maximized in that variables are defined to the database under a name made up from the package name, a hyphen, and the variable name. In this scheme, there is less potential for name conflicts. If flag is FALSE then variable are defined to the database simply by the variable name.

PASPCK

PASPCK(pnc, pname)

This function sets the current package to be the named one. PANACEA uses the concept of current package to set a context for operations such as variable access and control use. That is to say that such operations are only meaningful when a particular package is in effect to provide information and procedures necessary to their function.

Input to this function is: pnc, the number of characters in the package name; and pname the name of the package.

This function returns 1 if successful and 0 otherwise.

Variable Definers

Control Accessors/Definers

Unit Conversion/Definition Functionality

Database Access Functionality

This group of functions gives application packages access to the database variables or to non-database managed memory.

Basic Functions

PACONN

PADCON

PALOCA

PAMCON

Optional Functions

PALLOC

PACONN

PACONN(pv, pnc, vname, pf)

Connect the given pointer to the data associated with the named variable. PANACEA will record the address of the pointer pv if the flag, pf, is 1. In that case, if another operation causes PANACEA to change the size of or move the block of memory associated with the database variable, PANACEA will update the pointer pv to point to the new space.

Sometimes a connection is made to a local pointer which is on the stack or otherwise temporary. In such a case it could be disastrous for PANACEA to change the contents of memory there. The flag lets the application programmer control this behavior.

The arguments to this function are: pv, the pointer to be connected to the database variable vname; pnc, the number of characters in vname; vname, the name of the variable in the database; and pf, a flag specifying whether or not PANACEA is to record the address of pv so that it can be updated if the space to which it points is moved.

This function returns 1 if successful and 0 otherwise.

PADCON

PADCON(pv, pnc, vname)

Disconnect the pointer, pv, from the database variable named by name. The pointer is set to NULL (0x0) and future references to it will induce a crash or some other possibly undesirable behavior. PANACEA simply guarantees that the pointer will no longer be pointing to the memory block associated with the database variable.

In the case of DMND variables, when the number of pointers connected to the database variable storage falls to zero the variable is written back out to disk. It can be connected to at a later time with PACONN.

The arguments to this function are: pv, the pointer to be disconnected from the database variable vname; pnc, the number of characters in vname; and vname, the name of the variable in the database.

This function returns 1 if successful and 0 otherwise.

PALLOC

PALLOC(pv, pnc, vname, pf, dim_1, ...)

Define the dimensions for the named variable, allocate it, and set pv to point to the new space. Each dimension is ignored unless the corresponding dimension was NULL when the variable was defined.

In certain applications even the application variable which will hold the value of some dimensions is unknown at the start of a simulation run. PANACEA allows for this maximal delay by permitting the application variable which holds the dimension value to be supplied with the request to allocate space. In this call the dimensions which were specified when the variable was defined are ignored.

The arguments to this function are: pv, the pointer to be allocated and associated with the database variable vname; pnc, the number of characters in vname; vname, the name of the variable in the database; pf, a flag specifying whether or not PANACEA is to record the address of pv so that it can be updated if the space to which it points is moved; and dimi are the dimensions of the variable. They must match in number the dimension specifications used in the definition of the variable.

This function returns 1 if successful and 0 otherwise.

PALOCA

PALOCA(pv, pbpi, pn, dim_1, ...)

Allocate some local space and set the pv to point to the new space.

The arguments to this function are: pv, the pointer to be connected to the allocated space; pbpi, the number of bytes required for a single element of the variable; pn, the number of dimension for the variable; and dimi, the dimensions for the variable.

This function returns 1 if successful and 0 otherwise.

PAMCON

PAMCON(pnc, vn, pp, pn)

Connect the pointers in pp to the variables named by the entries in vn. There must be pn pointers in pp and pn strings in vn.

The purpose of this routine is to connect a number of pointers to their corresponding database variables in a single operation. It is assumed that PANACEA will track all of these variables and update the pointers if the memory associated them is reallocated. This is equivalent to having the flag in PACONN set to 1.

The arguments to this function are: pnc, an array containing the length of the variable names in vn; vn, an array of names of the variables in the database; pp, an array of pointers to be connected to the database variables named in vn; and pn, the number of variables to be connected. Each of the arrays pnc, vn, and pp must be pn long.

This function returns 1 if successful and 0 otherwise.

Simulation Control

Plot Request Handling

Generation Support

Source Variable/Initial Value Data Handling

Time History Data Management

The following functions define a service for the management of time history data.

This facility implements a time history data management model designed with the following goals:

- Easy to use
- Flexible enough to accommodate virtually any code system requirements
- Support families of files
- Support restart from any cycle in a simulation (the instance index is the key to this)
- The raw time history data should also be comprehensible.

The model chosen centers on the definition of a time history data structure which is defined at run time. Each such structure has a single domain component and multiple range components. These correspond to a set of time plots for which data is written at a particular frequency as determined by the application. By defining however many such structures are required, code systems can manage many sets of time plots each at their own frequency. New sets can be added over the course of a simulation. These structures are set up via the PAAREC, PABREC, and PAEREC calls. Once defined, the application can buffer up any amount of time history data and then write it out with PAWREC. Thus applications determine how much data will be kept in the running code.

The time history data is written out in families PDB of files. A family of time history PDB files is started with a call to PATHOP. The function PATHFM is used to close out one member of the family and start the next if the current file's size is above a user set threshold. Each file in a family maintains a link to the previous file in the family. This is very useful when applications restart a simulation from an earlier state and, potentially, follow a different branch of execution. Knowing the last time history file in any branch allows the unambiguous reconstruction of the time history data for that branch.

In addition to the time history data, any sort of associated data may be defined via the attribute mechanism in PDBLib. Additionally attributes may be assigned to particular instances of time history data structures via the PAWRIA call.

Since PANACEA is a part of PACT, it supplies functions to transpose the time history data files into ULTRA files (also a PACT tool). These functions are: PATRNN; PATRNL; and PATRNF. In this way, applications may directly transpose their time history files if desired. The stand alone program TOUL which transposes a family of time history files into a family of ULTRA files simply calls these functions appropriately. NOTE: all files in the family must be closed before any transpose operation. Functions to merge an arbitrary list of ULTRA files or a family of ULTRA files are also provided. These functions are: PAMRGN and PAMRGF.

This facility is substantially independent of the rest of PANACEA and can be used without any other parts of PANACEA.

Basic Functions

PAAREC

PABREC

PAEREC

PAWREC

Optional Functions

PAMRGF

PAMRGN

PATRNN

PATRNL

PATRNF

PAWRIA

PAAREC

```
integer PAAREC(integer fileid,
               integer recid,
               integer nm,
               char *memb,
               integer nl,
               char *labl)
```

Add a member to the definition of the time history data structure specified.

This function is a part of the process which defines a special derived type to be used for gathering time history data that a generic tool (e.g. TOUL) can convert into arrays of time ordered data.

The member, memb. must follow the rules for member specifications as described in the PDBLib User’s Manual with the exception that if no type is specified the type is taken to be "double" (real*8). Briefly, a member specification consists of a type, an identifier, and optional dimension specifications. Member identifiers must not contain white space characters or any of the following: ‘*’, ‘[‘, ‘]’, ‘(‘, ‘)’, or ‘.’.

NOTE: A non-scalar member does not transpose into a set of curves. In such a case PDBView is used directly to manipulate the time history data since it has superior capabilities for allowing users to control the semantics of their data.

Since a very common use for this capability is to generate curves for plotting purposes, the labl argument is provided for the application to supply curve labels. The nl argument may be 0 if the application wishes PANACEA to construct labels from the member.

Input to PAAREC is: fileid, an integer identifying a PDB file; recid, an integer identifying a derived type definition started by PABREC; nm, an integer number of characters in the member; memb, an ASCII string containing the name of the member; nl, an integer number of characters in the labl string; and labl, an ASCII string containing a curve label for this member versus the domain member.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function PFGERR.

See also PABREC, PAEREC, and PAWREC.

Example:

```
       integer paarec
               integer fileid, recid
                       .
                       .
                       .
               if (paarec(fileid, recid, 5, 'd[20]', 0, ' ') .eq. 0)
              $    call errproc
                       .
                       .
                       .
```

PABREC

```
 integer PABREC(integer fileid,
                integer nf,
```

```
char *name,
integer nt,
char *type,
integer nd,
char *domain)
```

Begin the definition of a special type to be used for gathering time history data that a generic tool (e.g. TOUL) can convert into arrays of time ordered data. This begins a process which defines both a derived data type and an entry in the symbol table of the file to contain the data. PANACEA implicitly adds another entry which contains information that a transposer would use.

The name of the domain member is supplied here since each time history struct has only one domain variable (x data).

The domain must follow the rules for member specifications as described in the PDBLib User's Manual with the exception that if no type is specified the type is taken to be "double". Briefly, a member specification consists of a type, an identifier, and optional dimension specifications. Member identifiers must not contain white space characters or any of the following: '*', '[', ']', '(', ')', or '.'. The first member is taken to be the domain data (x value) and the remaining members are taken to be ranges. Each data set of the specified type transposes into nmemb-1 curves, that is one x array and nmemb-1 y arrays.

Input to PABREC is: fileid, an integer identifying a PDB file; nf, an integer number of characters in name; name, an ASCII string containing the name of the data entry in the symbol table; nt, an integer number of characters in the type string; type, an ASCII string containing the name of the new derived type; nd, an integer number of characters in the domain string; and domain, an ASCII string labelling the domain member.

A non-zero integer identifier for this time history record structure is returned if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function PFGERR.

See also PAAREC, PAEREC, and PAWREC.

Example:

```
nteger pabrec
          integer fileid, recid
                 .
                 .
                 .
          recid = pabrec(fileid, 6, 't-data', 8, 't-struct', 4, 'time')
```

```
                    .
                    .
                    .
```

PAEREC

```
 integer PAEREC(integer fileid,
               integer recid)
```

Complete the definition of a time history data structure and enter the definition in the PDB file.

Input to PAEREC is: fileid, an integer identifying a PDB file; and recid, an integer identifying a derived type definition started by PABREC.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function PFGERR.

See also PAAREC, PABREC, and PAWREC.

Example:

```
     integer paerec
              integer fileid, recid
                     .
                     .
                     .
              if (paerec(fileid, recid) .eq. 0)
             $    call errproc
                     .
                     .
                     .
```

PAMRGF

```
integer PAMRGF(integer nb,
               char *base,
               integer nf,
               char *family,
               integer nc)
```

Merge a family of ULTRA files into a single file or family of files. The ULTRA source files are specified here by family base name.

ULTRA source files are assumed to follow the naming convention

family.udd

where family is the base source file name supplied and dd are base 36 digits (0-9a-z).

ULTRA target files produced by this function similarly follow the naming convention

base.udd

where base is the base target file name supplied.

Argument nc, determines how target files are familied by specifying the approximate number of curves per file. However, regardless of the value of nc, all curves from a given source file will reside in a single target file. If nc is zero, all curves will be merged into a single file.

For efficiency this function currently assumes that all curves in a given source file share the domain of the first curve in that file. This is true for ULTRA files produced by the PANACEA time history transpose routines. Curves from arbitrary ULTRA files can be merged, albeit less efficiently, using the save command in the ULTRA utility.

Input to this function is: nb, an integer number of characters in base; base, an ASCII string containing the base target file name; nf, an integer number of characters in famly; family, an ASCII string containing the base source file name; and nc an integer approximate number of curves per ULTRA target file.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function PFGERR.

See also PAMRGN.

Example:

```
      integer pamrgf
                .
                .
                .
       if (pamrgf(3, 'bar', 3, 'foo', 1000) .eq. 0)
      $    call errproc
                .
                .
                .
```

PAMRGN

```
integer PAMRGN(integer nb,
               char *base,
               integer nn,
               char *names,
               integer nc)
```

Merge a set of ULTRA files into a single file or family of files. The ULTRA source files are specified here explicitly by name.

ULTRA target files produced by this function follow the naming convention

base.udd

where base is the base target file name supplied and dd are base 36 digits (0-9a-z).

Argument nc, determines how target files are familied by specifying the approximate number of curves per file. However, regardless of the value of nc, all curves from a given source file will reside in a single target file. If nc is zero, all curves will be merged into a single file.

For efficiency this function currently assumes that all curves in a given source file share the domain of the first curve in that file. This is true for ULTRA files produced by the PANACEA time history transpose routines. Curves from arbitrary ULTRA files can be merged, albeit less efficiently, using the save command in the ULTRA utility.

Input to this function is: nb, an integer number of characters in base; base, an ASCII string containing the base file name; nn, an integer number of characters in names; names, an

ASCII string containing the space delimited full names of the ULTRA source files; and nc an integer approximate number of curves per ULTRA target file.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function PFGERR.

See also PAMRGF.

Example:

```
         integer pamrgn
               character*8 names(3)
                      .
                      .
                      .
            c  merge files foo.u09 through foo.u0b
               names(1) = 'foo.u09 '
               names(2) = 'foo.u0a '
               names(3) = 'foo.u0b '
               if (pamrgn(3, 'bar', 24, names, 1000) .eq. 0)
              $    call errproc
                      .
                      .
                      .
```

PATHFM

integer PATHFM(integer fileid)

Check the current file's size against the maximum size as specified in the PATHOP call. If the file is larger than the maximum size, close the file, open the next member of the file family, and return an id for the new file.

Input to this function is: fileid, an integer file identifier.

An integer PDBfile identifier is returned if the call is successful; otherwise, -1 is returned and an error message may be retrieved by invoking function PFGERR. The return value should be checked against the original one to see whether or not a new file has been opened.

See also PABREC, PAAREC, PAEREC, PAWREC, and PATHOP.

Example:

```
      integer pathfm
      integer nfid, fid
             .
             .
             .
      nfid = pathfm(fid)
      if (nfid .eq. -1)
     $   call errproc

      if (nfid .ne. fid)
     $   fid = nfid
             .
             .
             .
```

PATHOP

```
integer PATHOP(integer nf,
               char *name,
               integer nm,
               char *mode,
               integer sz,
               integer np,
               char *prev)
```

Open a new time history data file. This implicitly defines a family of files. The name should be of the form: base.tdd where d is a base 36 digit (i.e. 0-9a-z). This is only a convention, but there are certain consequences for not following it in as much as the familying mechanism assumes that the last two characters form a base 36 number and increments it accordingly. As an application writes data to a time history file, periodic calls to PATHFM should be made to monitor the file size and when necessary close the current family member and open the next.

Since simulations may be restarted and each code may have a different strategy for continuing time history collection in the event of a restart, it is necessary to allow for name changes in the family. The consequence of this is that each member of a file family must contain the name of the previous file in the family. In that way, the transpose process may unambiguously and under the control of the user or simulation follow a chain of time history files from the end point back to the beginning. The prev argument is used to supply this information. The family of files that follows will be in sequence from the name supplied. Only across restarts, which implies calls to PATHOP, may the sequence name be changed. A call to PATHOP may have 0 for np which indicates the absolute beginning of the sequence, i.e. the transpose will stop in its search for files at this point.

Input to this function is: nf, an integer number of characters in name; name, an ASCII string specifying the name of the first file in the family for the current application; nm, an integer number of characters in mode; mode, an ASCII string specifying the mode of the file (typically this will be "w"); and sz, an integer value specifying the target maximum file size for members of the family.

An integer PDBfile identifier is returned if the call is successful; otherwise, -1 is returned and an error message may be retrieved by invoking function PFGERR.

See also PABREC, PAAREC, PAEREC, PAWREC, and PATHFM.

Example:

```
      integer pathop
              .
              .
              .
       fid = pathop(3, ‘foo’, 1, ‘w’, 1000000, 4, ‘blat’)
       if (fid .eq. -1)
      $    call errproc
              .
              .
              .
```

PATHTL

```
integer PATHTL(char *names,
               integer ord,
               int nc)
```

WARNING: This is a deprecated function. Use PATRNL instead.

Write an ULTRA file by transposing the time history data from a specified set of time history files. The time history files are specified here by giving the names of files at the end of file family chains.

On occasion, files in a family are lost. This breaks the chain of files as well as leaving gaps in the data. Since this function can take many file names, it can be used to take into account missing files by supplying the files at the top of the gap(s) of missing files. In the accompanying example it is assumed that the files foo.t09 through foo.t11 are missing.

There is the possibility inherent in this model of handling file families that two or more files to be processed will have data for some regions in time. Because of the underlying PDB machinery, the data from the last file processed will predominate. Also, since the files are linked together from latest to earliest, it is sometimes necessary to require that the files be processed in the opposite order in which they are specified. The ord argument is used for this purpose. A value of 1 causes the files to be processed in the order in which they are specified while a value of -1 causes them to be processed in reverse order.

A family of time history files is conventionally named

name.tdd

where name is the base file name and dd are base 36 digits (0-9a-z).

The resulting family of ULTRA files will be named

name.udd

Input to this function is: names, an ASCII string containing the space delimited full names of the time history files; ord, an integer flag specifying the order of processing; and nc a maximum number of curves per file in the ULTRA file family.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function PFGERR.

See also PATRNN and PATRNF.

Example:

```
integer pathtl

character*8 names(2)
```

```
        .
        .
        .
c  files foo.t09 through foo.t11 are missing
   names(1) = 'foo.t1f '
   names(2) = 'foo.t08 '
   if (pathtl(names, -1, 1000) .eq. 0)
  $    call errproc
        .
        .
        .
```

PATHTN

```
integer PATHTN(char *names,
               integer ord,
               int nc)
```

WARNING: This is a deprecated function. Use PATRNN instead.

Write an ULTRA file by transposing the time history data from a specified set of time history files. The time history files are specified here explicitly by name.

On occasion, it is desirable to transpose only selected files from a family. For example, a user may know that only certain times are of interest and doesn't wish to see the entire data set. This function gives the finest level of control to the application about which files to transpose.

There is the possibility inherent in this model of handling file families that two or more files to be processed will have data for some regions in time. Because of the underlying PDB machinery, the data from the last file processed will predominate. Also, since the files are linked together from latest to earliest, it is sometimes necessary to require that the files be processed in the opposite order in which they are specified. The ord argument is used for this purpose. A value of 1 causes the files to be processed in the order in which they are specified while a value of -1 causes them to be processed in reverse order.

A family of time history files is conventionally named

name.tdd

where name is the base file name and dd are base 36 digits (0-9a-z).

The resulting family of ULTRA files will be named

name.udd

Input to this function is: names, an ASCII string containing the space delimited full names of the time history files; ord, an integer flag specifying the order of processing; and nc a maximum number of curves per file in the ULTRA file family.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function PFGERR.

See also PATRNL and PATRNF.

Example:

```
      integer pathtn
      character*16 names(4)
             .
             .
             .
c only transpose foo.t09 through foo.t0b
      names(1) = 'foo.t09 '
      names(2) = 'foo.t0a '
      names(3) = 'foo.t0b '
      names(4) = ' '
      if (pathtn(names, 1, 1000) .eq. 0)
     $    call errproc
             .
             .
             .
```

PATHTR

```
 integer PATHTR(integer nf,
               char *name,
```

```
integer nc)
```

WARNING This is a deprecated function. Use PATRNF instead.

Write an ULTRA file by transposing the time history data from a given set of time history files.

A family of time history files is conventionally named

name.tdd

where name is the base file name and dd are base 36 digits (0-9a-z).

The resulting family of ULTRA files will be named

name.udd

Input to this function is: nf, an integer number of characters in name; name, an ASCII string containing the base name of the time history file family; and nc a maximum number of curves per file in the ULTRA file family.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function PFGERR.

See also PATRNL and PATRNN.

Example:

```
      integer pathtr
             .
             .
             .
      if (pathtr(3, 'foo', 1000) .eq. 0)
     $    call errproc
             .
             .
             .
```

PATRNF

```
integer PATRNF(integer nf,
```

```
char *name,
integer ord,
integer nc)
```

Write an ULTRA file by transposing the time history data from a given set of time history files.

A family of time history files is conventionally named

name.tdd

where name is the base file name and dd are base 36 digits (0-9a-z).

The resulting family of ULTRA files will be named

name.udd

There is the possibility inherent in this model of handling file families that two or more files to be processed will have data for some regions in time. Because of the underlying PDB machinery, the data from the last file processed will predominate. Also, since the files are linked together from latest to earliest, it is sometimes necessary to require that the files be processed in the opposite order in which they are specified. The ord argument is used for this purpose. A value of 1 causes the files to be processed in the order in which they are specified while a value of -1 causes them to be processed in reverse order.

Input to this function is: nf, an integer number of characters in name; name, an ASCII string containing the base name of the time history file family; ord, an integer flag specifying the order of processing; and nc a maximum number of curves per file in the ULTRA file family.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function PFGERR.

See also PATRNL and PATRNN.

Example:

```
       integer patrnf
              .
              .
              .
       if (patrnf(3, 'foo', 1, 1000) .eq. 0)
      $    call errproc
```

.

.

.

PATRNL

```
integer PATRNL(int nn,
               char *names,
               integer ord,
               int nc)
```

Write an ULTRA file by transposing the time history data from a specified set of time history files. The time history files are specified here by giving the names of files at the end of file family chains.

On occasion, files in a family are lost. This breaks the chain of files as well as leaving gaps in the data. Since this function can take many file names, it can be used to take into account missing files by supplying the files at the top of the gap(s) of missing files. In the accompanying example it is assumed that the files foo.t09 through foo.t11 are missing.

There is the possibility inherent in this model of handling file families that two or more files to be processed will have data for some regions in time. Because of the underlying PDB machinery, the data from the last file processed will predominate. Also, since the files are linked together from latest to earliest, it is sometimes necessary to require that the files be processed in the opposite order in which they are specified. The ord argument is used for this purpose. A value of 1 causes the files to be processed in the order in which they are specified while a value of -1 causes them to be processed in reverse order.

A family of time history files is conventionally named

```
name.tdd
```

where name is the base file name and dd are base 36 digits (0-9a-z).

The resulting family of ULTRA files will be named

```
name.udd
```

Input to this function is: nn, an integer number of characters in names; names, an ASCII string containing the space delimited full names of the time history files; ord, an integer flag specifying the order of processing; and nc a maximum number of curves per file in the ULTRA file family.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function PFGERR.

See also PATRNN and PATRNF.

Example:

```
       integer patrnl
       character*8 names(2)
              .
              .
              .
c      files foo.t09 through foo.t11 are missing
       names(1) = 'foo.t1f '
       names(2) = 'foo.t08 '
       if (patrnl(16, names, -1, 1000) .eq. 0)
      $     call errproc
              .
              .
              .
```

PATRNN

```
integer PATRNN(int nn,
               char *names,
               integer ord,
               int nc)
```

Write an ULTRA file by transposing the time history data from a specified set of time history files. The time history files are specified here explicitly by name.

On occasion, it is desirable to transpose only selected files from a family. For example, a user may know that only certain times are of interest and doesn't wish to see the entire data set. This function gives the finest level of control to the application about which files to transpose.

There is the possibility inherent in this model of handling file families that two or more files to be processed will have data for some regions in time. Because of the underlying PDB machinery, the data from the last file processed will predominate. Also, since the files are linked together from latest to earliest, it is sometimes necessary to require that the files be processed in the opposite order in which they are specified. The ord argument is used for this purpose. A value of 1 causes the files to be processed in the order in which they are specified while a value of -1 causes them to be processed in reverse order.

A family of time history files is conventionally named

```
name.tdd
```

where name is the base file name and dd are base 36 digits (0-9a-z).

The resulting family of ULTRA files will be named

```
name.udd
```

Input to this function is: nn, an integer number of characters in names; names, an ASCII string containing the space delimited full names of the time history files; ord, an integer flag specifying the order of processing; and nc a maximum number of curves per file in the ULTRA file family.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function PFGERR.

See also PATRNL and PATRNF.

Example:

```
      integer patrnn
      character*8 names(3)
             .
             .
             .
c only transpose foo.t09 through foo.t0b
      names(1) = 'foo.t09 '
      names(2) = 'foo.t0a '
      names(3) = 'foo.t0b '
      if (patrnn(24, names, 1, 1000) .eq. 0)
     $    call errproc
             .
```

```
                .
                .
```

PAWREC

```
integer PAWREC(integer fileid,
               integer recid,
               integer inst,
               integer nr,
               vr)
```

Write out nr instances of time history data whose structure has been defined by PABREC, PAAREC, and PAEREC. This function writes out nr complete instances of a time history record! Using this function an application can manage multiple sets of time history data which are written at different frequencies.

Input to this function is: fileid, an integer identifying a PDB file; recid, an integer identifier specifying a time history record type; inst, an integer specifying the instance index of the records to be written; nr, an integer specifying the number of instances of the structure pointed to by vr; and vr, the data to be written.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function PFGERR.

See also PAAREC, PABREC, PAEREC, and PAWRIA.

Example:

```
      integer pawrec
      integer fileid, recid
      real*8 vr(9)
             .
             .
             .
       if (pawrec(fileid, recid, 14, 3, vr) .eq. 0)
      $    call errproc
             .
```

```
                 .
                 .
```

PAWMEM

```
integer PAWMEM(integer fileid,
               integer recid,
               integer nc,
               char *name,
               integer inst,
               vr)
```

Write out one member of a time history data record whose structure has been defined by PABREC, PAAREC, and PAEREC. This function writes out one member of a time history record. Its utility is most apparent when the member is an array which could be large or is part of a very large record.

Input to this function is: fileid, an integer identifying a PDB file; recid, an integer identifier specifying a time history record type; nc, an integer number of characters in the member string; member, an ASCII string naming the member desired; inst, an integer specifying the instance number; and vr, the data to be written.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function PFGERR.

See also PAAREC, PABREC, PAEREC, and PAWRIA.

Example:

```
      integer pawmem
      integer fileid, recid
      real*8 vr(100,100)
             .
             .
             .
```

```
 if (pawmem(fileid, recid, 3, ‘v_1’, 14, vr) .eq. 0)
$    call errproc
        .
        .
        .
```

PAWRIA

```
integer PAWRIA(integer fileid,
               integer nv,
               char *var
               integer inst,
               integer na,
               char *attr,
               value)
```

Assign an attribute value to the indicated time history data instance.

Input to this function is: fileid, an integer identifying a PDB file; nv, an integer number of characters in the string var; var, an ASCII string naming the time history data entry; inst, an integer specifying the instance index of the records to be written; na, an integer specifying the number of characters in the attribute name; attr, an ASCII string containing the name of an attribute; and; and value, the attribute value.

The return value is 1, if successful; otherwise, 0 is returned and an error message may be retrieved by invoking function PFGERR.

See also PAAREC, PABREC, PAEREC, and PAWREC.

Example:

```
integer pawria
integer fileid
character*8 datestr
```

```
                .
                .
                .
          if (pawria(fileid, 6, 't-data', 3, 4, 'date', datestr)
         $     .eq. 0) call errproc
                .
                .
                .
```

FORTRAN API Time History Example

Here is an example of using the time history functionality of PANACEA.

```
c
c PATHFT.F - FORTRAN test of PANACEA Time History Functionality
c

c------------------------------------------------------------------------
c------------------------------------------------------------------------

c MAIN - start it out here

      program pantest

      common / comi / ncycle, filid, recid
      common / comr / time, v1, v2, v3
      real*8 time, v1, v2, v3

      call initlz

c ... generate some data
      do 100 ncycle = 1, 10

         time = 0.01*float(ncycle)
         v1   = 10.0 + 10.0*time
         v2   = 20.0*time
         v3   = v2*(1.0 - time)

         call output
 100  continue

      call finish

      stop
      end

c------------------------------------------------------------------------
c------------------------------------------------------------------------

c FINISH - close the files and transpose
```

```
      subroutine finish

      common / comi / ncycle, filid, recid
      integer patrnf, pfclos, pathfm
      integer filid, nfid

      nfid = pathfm(filid)
      if (nfid .eq. -1)
     &    call errorh
      if (nfid .ne. filid)
     &   filid = nfid

      if (pfclos(filid) .eq. 0)
     & call errorh

      if (patrnf(4, 'thft', 100) .eq. 0) then
         call errorh
         stop 8
      endif

      return
      end

c------------------------------------------------------------------------
c------------------------------------------------------------------------

c INITLZ - initialize the TH records

      subroutine initlz

      common / comi / ncycle, filid, recid
      integer paarec, pabrec, paerec, pathop, pfopen
      integer filid, recid
      character*8 name, type, domain
      integer nf, nt, nd

c ... open a TH file family
      filid = pathop(8, 'thft.t00', 1, 'w', 100000, 0, ' ')
      if (filid .le. 0)
     &   stop 7

      name   = 'tdata'
      nf     = 5
      type   = 't-struct'
      nt     = 8
      domain = 'time'
      nd     = 4

c ... define a record structure to be written to the pdbfile
c ... this defines the domain variable (time in this case)
      recid = pabrec(filid, nf, name, nt, type, nd, domain)
      if (recid .eq. 0)
     &   call errorh

c ... add in any number of members to the record structure
      if (paarec(filid, recid, 4, 'v1_1', 15, 'region boundary') .eq. 0)
     &   call errorh

      if (paarec(filid, recid, 4, 'v1_2', 0, ' ') .eq. 0)
     &   call errorh
```

```
       if (paarec(filid, recid, 2, 'v2', 13, 'density*value') .eq. 0)
     &    call errorh

c ... end the record structure
      if (paerec(filid, recid) .eq. 0)
     &   call errorh

      return
      end

c-----------------------------------------------------------------------
c-----------------------------------------------------------------------

c OUTPUT - fill the buffer and dump it

      subroutine output

      common / comi / ncycle, filid, recid
      common / comr / time, v1, v2, v3
      real*8 time, v1, v2, v3
      integer pawrec
      real*8 tstruct(4)
      integer filid, recid, inst

c ... put the values into the proper order in the record structure
c ... the domain element is always the first member
      tstruct(1) = time
      tstruct(2) = v1
      tstruct(3) = v3
      tstruct(4) = v2

      inst = ncycle

      if (pawrec(filid, recid, inst, 1, tstruct(1)) .eq. 0)
     &   call errorh

      return
      end

c-----------------------------------------------------------------------
c-----------------------------------------------------------------------

c ERRORH - error reporter

      subroutine errorh

      character*(256) err
      integer nchr

      call pfgerr(nchr, err)

      write(*, *)err

      end

c-----------------------------------------------------------------------
c-----------------------------------------------------------------------
```

Miscellaneous

PANACEA Initial Value Problem Generators

Generic Commands Supplied by PANACEA

dump
Build the database and write a restart dump.

end
Shutdown the generator.

graph
Specify a plot request.

name
Set the value of a name for the current package.

package
Specify the package to which subsequent SWITCH, PARAMETER, and NAME commands apply.

parameter
Set the value of a parameter for the current package.

read
Read a file of generator commands.

s
Continuation of last specify command.

specify

Specify an initial value constraint (BC, SRC, or <variable>). The precise syntax for this command is:

```
specify <bc name | src name | name> spec-list
 spec-list :=  spec-list spec | spec
      spec :=  from file |
               interpolate |
               discrete |
               in name |
               at value |
               along value |
               value
```

The specifications at and along are simply put their associated value into the array of floating point numbers which the specify command constructs. In short, they are for the clarity of the input command. The in specification invokes the PA_reg_hook to map a name to a number which goes in the array which specify builds. The intention again is to provide an additional level of clarity by allowing users of a PANACEA code to be able to specify at least some information by name. Any value can be denoted by an alias which is defined by a call to PA_alias.

The array of number which this command builds up can be handed back to those part of the packages which query the initial value specifications.

switch
Set the value of a switch for the current package.

PANACEA Supplied Physical Constants

PANACEA provides a service for handling physical units in simulation packages. It is geared toward solving problems associated with importing and exporting packages between PANACEA codes and toward letting the code developer and code users have control over the systems of units in which problems are run and debugged.

Part of this service hinges on providing fundamental physical constants which the code developer can use in expressions appearing in the set of equations which their packages solve. Coupled with a complete definition of the set of units (see the next section) PANACEA is able to change the values of its physical constants to reflect the system of units in which the user or code developer wishes to work.

At the beginning of a code run, these constants are initialized with their CGS values as indicated below. Once the system of units is defined (either by the input deck for the initial value problem generator or from the restart dump for the main simulation code) all of these constants are converted into the defined system of units.

The constants given below are divided into three categories: ones with no units (pure numbers); those with physical units that cannot be expressed in terms of more fundamental physical constants; and those which can be expressed in terms of more fundamental constants.

Pure Numbers

alpha
Fine structure constant = 7.297353080e-3

Coulomb
Coulomb in fundamental charges = 6.241506363e18

N0
Avogadro's number = 6.02213665e23

Fundamental Physical Constants

c
Speed of light (cm/sec) = 2.99792458e10

Gn
Newtonian gravitational constant (cm3/g-sec2) = 6.673231e-8

Hbar
Planck's constant divided by 2Pi (erg-sec) = 1.05457267e-27

kBoltz
Boltzman constant (erg/K) = 1.380658e-16

M_e
electron mass (g) = 9.109389754e-28

Derived Physical Constants

e
Electron charge, sqrt(alpha*Hbar*c) (esu) = 4.80320680e-10

eV_erg
eV to erg, 1.0e7/Coulomb (erg/eV) = 1.60217733e-12

HbarC
Hbar*c (eV-cm) = 1.97327054e-5

M_a
Atomic mass unit, 1/N0 (g) = 1.660540210e-24

M_e_eV
Electron mass, Me*c*c/eV_erg (eV) = 5.10999065e5

Ryd
Rydberg, (M_e*c2*alpha2)/2 (eV) = 13.605698140

In addition to theses constants, the constants #define'd by PML are available to PANACEA packages. These are:

DEG_RAD
Conversion from degrees to radians = 0.017453292519943295

PI
π = 3.1415926535897931

RAD_DEG
Conversion from radians to degrees = 57.295779513082323

Physical Units in PANACEA

In PANACEA the view is taken that all expressions involving fundamental physical constants should be expressed in terms of scalar variables representing those constants, NOT as numerical constants. There are two main reasons for this: PANACEA can offer the code developer many options for converting between systems of units; and the expressions involving such constants are self-documenting.

A critical part of PANACEA's ability to perform these services is its ability to specify and manipulate the physical units which a simulation package requires. PANACEA starts with a reasonable set of physical units:

Type	Macro	Variable
angle	RAD	PA_radian
	STER	PA_steradian
charge	Q	PA_electric_charge
energy	EV	PA_eV
	ERG	PA_erg
length	CM	PA_cm
mass	G	PA_gram
number	MOLE	PA_mole
temperature	K	PA_kelvin
time	SEC	PA_sec
volume	CC	PA_cc

The code developer can define new units with a call to PA_def_unit. PA_def_unit takes a numerical value and a list of previously defined units and returns the index of a newly defined unit. For example, the units CM, SEC, ERG and CC are defined as:

```
PA_cm  = PA_def_unit(1.0, UNITS);

PA_sec = PA_def_unit(1.0, UNITS);

PA_erg = PA_def_unit(1.0, G, CM, CM, PER, SEC, SEC, UNITS);

PA_cc  = PA_def_unit(1.0, CM, CM, CM, UNITS);
```

In this way both primitive units, i.e. units which don't depend on any previously defined unit, and compound units may be defined and used by PANACEA code systems.

In the table above, the macro designation means that the name refers to a #define'd constant and the variable designation means that the name refers to a global scalar variable. In C it turns out to be useful to have both the macro and variable forms. So codes which are going to define units should have a set of #define's corresponding to their PA_def_unit calls. In the case of the above example:

```
#define CM         PA_cm

#define SEC        PA_sec

#define ERG        PA_erg

#define CC         PA_cc
```

would be the appropriate sort of association.

In the PANACEA model there are three systems of units: CGS; internal; and external. The physical constants are initialized in CGS units. The user of a PANACEA code specifies input and receives output in the external unit system. In the simulation routines quantities are expressed in the internal system of units.

There are two arrays of double precision floating point numbers which keep the conversion factors for each of the defined units. The array unit keeps the conversion factors between CGS and internal units. The array convrsn keeps the conversion factors between CGS and external units. For example, suppose a code function needs a length in the three systems of units and these are held in the variables: length_internal, length_cgs, and length_external. The values of length_internal and length_external can be obtained from length_cgs as follows:

```
length_internal = length_cgs*unit[CM]

length_external = length_cgs*convrsn[CM]
```

The two situations in which this is most useful is during initial value problem generation when the data which has been read in has not yet been used to create the data structures and arrays to be used in the simulation itself and during the simulation when data from source files is brought in.

PANACEA applies conversions automatically when putting out data for post processing purposes. It also lets the code developer specify what the units of the state files (restart dumps) is to be and handles any necessary conversions.

An example of setting up unit systems is given below. This routine is invoked by the initial value problem generator. Depending on the value of the argument the internal system of units is defined to be "natural units" with Hbar and c both 1 or a modified CGS system with temperatures in eV.

```
/* INIT_CONT - initialize the controls */


init_cont(def_unit_flag)

   int def_unit_flag;

   {PA_def_units(FALSE);


/* set up the physical constants in CGS units */

    PA_physical_constants_cgs();


/* define the internal system of units

 * if def_unit_flag it TRUE, "natural units" with Hbar = c = 1

 * are defined

 */
```

```
    if (def_unit_flag)
       {unit[G]    = g_icm;
        unit[SEC] = c;
        unit[K]    = eV_icm*(kBoltz/eV_erg);
        unit[EV]   = eV_icm*unit[Q]*unit[Q]/unit[CM];}

/* the alternate internal system of units is the modified CGS system
 * with temperatures in eV
 */
    else
       unit[K] = kBoltz/eV_erg;

/* define the external units to be modified CGS with
 * temperatures in eV
 */
    convrsn[K] = kBoltz/eV_erg;

/* make the conversion factors consistent with these changes */
    PA_set_conversions(TRUE);

/* set up the physical constants in the external system of units */
    PA_physical_constants_ext();

    return;}
```

In the above example, pay special attention to the calls to PA_def_units, PA_physical_constants_cgs, PA_set_conversions, and PA_physical_constants_ext. The call to PA_def_units causes PANACEA to define all of its units and initialize the unit and convrsn arrays. The call to PA_physical_constants_cgs initializes the physical constants in CGS units. The call to PA_set_conversions passes over unit and convrsn and makes the entire set of units consistent with the changes made between the PA_def_units call and the PA_set_conversions call. Lastly, the call to PA_physical_constants_ext converts the physical constants to the external system of units.

At this point the initial value problem generator has as consistent set of units and physical constants. If the user changes the systems of units in the input deck, the calls to PA_set_conversions and PA_physical_constants_ext should be repeated to keep every thing consistent.

In the main simulation code everything is taken care of by PANACEA. In the process of reading the state file (restart dump), calls to PA_def_units, PA_set_conversions, and PA_physical_constants_cgs are made by PANACEA. The only difference is that at the end of the process of reading the restart dump, PANACEA calls PA_physical_constants_int to convert the physical constants to the internal system of units.

Source Data Handling in PANACEA

The PANACEA Cookbook

The purpose of this section is to provide some concrete guidance to users who want to get started with PANACEA. The first two subsections describe how to analyze a simulation code with a view to building a new code from scratch or to converting an existing code to use PANACEA services. The last subsection gives extracts of the source code to the PANACEA code system, ABC. The extracts were taken to exemplify the PANACEA interface and to give complete (i.e. with full detail) functions which define PANACEA packages.

Building a PANACEA Code System

The first place to start when building a PANACEA code is at the top. Since PANACEA is a model of a simulation code, the new code should be conceptualized in terms of the PANACEA model.

Break down the simulation into conceptual packages. That is, decide how to best modularize your code system at the level of systems of equations to be simulated. Try to think in terms of giving a package to another code developer so that he or she has everything that is needed to generate, run, and post process data for the package.

Define each variable in terms of type, attributes (RESTART, REQU, etc.), and the package which will own it. PANACEA can perform its services only for variables in the database. Other variables which are temporary, local, or for some other reason do not belong in the database should be identified and allotted to the packages which will handle them.

Decide how to handle issues associated with computational meshes. If feasible plan on putting mesh handling routines (routines that deal only with meshes - no physics) into one global package. Remember that mesh handling includes mesh generation or the facility to link to a mesh generator in the initial value problem generator.

For each identified package do the following steps:

1. Eliminate dimensional numeric constants

Replace dimensional numeric constants with variables whose values are constructed from the fundamental physical constants supplied by PANACEA (or plan on defining additional constants - see the section on Physical Units in PANACEA).

For example, replace an expression such as

    x = 2.99e10*t;

with

    x = c*t;

When this process is complete, the only numeric constants in the code should be pure numbers, i.e. dimensionless.

2. Build a function to define the package variables

With each package there will be a set of variables which the package owns. A helpful way to identify these is to ask what would be required to hand this package to another code system. Variables will fall into three main classes: RESTART, RUNTIME, and EDIT. The variable dimensions are typically stored in the swtch array for the package for efficiency in handling the state file.

```
def_foo(pck)

   PA_package *pck;

   {P_dim_1 = &swtch[4];

    P_dim_2 = &swtch[5];

    P_dim_3 = &Two_I;

    P_dim_4 = &Three_I;


/* RESTART VARIABLES */

    PA_def_var("foo_var_1", SC_INTEGER_S, NULL, NULL,

               SCOPE, RESTART, CLASS, OPTL, ATTRIBUTE,

               P_dim_1, DIMENSION, UNITS);


/* RUNTIME VARIABLES */
```

```
    PA_def_var("foo_var_2", SC_DOUBLE_S, NULL, NULL,
               ATTRIBUTE,
               P_dim_2, P_dim_3, DIMENSION, UNITS);

/* EDIT VARIABLES */
    PA_def_var("foo_var_3", SC_INTEGER_S, NULL, NULL,
               SCOPE, EDIT, CLASS, PSEUDO, ATTRIBUTE,
               P_dim_4, DIMENSION, UNITS);

    return(TRUE);}
```

Note: 1) the use of PA_def_var; 2) the way dimensions are handled.

3. Build a function to define the controls

With each package there will be a set of controlling parameters. PANACEA provides for an array of integer parameters, double precision parameters, and ASCII strings associated with each package. This permits efficient handling of these controls and does not hamper the code developer in associating them with named scalar variables.

Define the controls for the package and write the function to allocate them and initialize them.

```
cont_foo(pck)
   PA_package *pck;
   {static int n_names = 2, n_params = 2, n_swtches = 5;

    PA_mk_control(pck, "foo", n_names, n_params, n_swtches);

    swtch[1]   = TRUE;    /* main switch - use this package if TRUE */
    swtch[2]   = 1;                                 /* option #1 */
    swtch[3]   = 0;                                  /* option #2 */
    swtch[4]   = 0;                    /* number of bar_1 values */
    swtch[5]   = 0;                    /* number of bar_2 values */
```

```
    param[1]   = 1.0;                                  /* multiplier 1 */

    param[2]   = 1.0;                                  /* multiplier 2 */


    return(TRUE);}
```

Note: 1) the use of PA_mk_control; and 2) the static integers n_names, n_params and n_swtches which define the length of the swtch, name, and param arrays respectively.

4. Build the package initializer function

This function will be executed once at start up and should connect database variables with local pointers as well as allocating local variables which will endure for the entire simulation run.

```
int *loc_var_1;

double *loc_var_2;


foo_init(pck)

   PA_package *pck;

   {N_foo_1     = swtch[2];

    N_foo_2     = swtch[3];


/* PA_CONNECT for database variables */

    PA_CONNECT(foo_var_1, "variable_1", int *, TRUE);


/* MAKE_N for non-database local variables */

    loc_var_1   = MAKE_N(int, N_foo_1);

    loc_var_2   = MAKE_N(double, N_foo_1*N_foo_2);


    return(TRUE);}
```

Note: 1) the use of PA_CONNECT and MAKE_N; and 2) the dimension constants extracted from the swtch array.

5. Build a wrapper for the main entry point

Since many packages will want to vote on the timestep for the next major computational cycle and since it will be useful for most packages to monitor their performance, it will be a good idea to build a wrapper function around the package main entry point. It is also a good place to invoke initializations which occur every time the main entry point is invoked as well as clean up after it has been executed.

```
main_foo(pck)
   PA_package *pck;
   {int foo_z;
    double foo_dt;
    static int first = TRUE;

/* check that this package was requested */
    if (swtch[1] == FALSE)
       return;

    if (first)
       pck->space = (double) Ssp_alloc;

    PA_MARK_TIME;

    init_cycle();

/* do the real work of foo*/
    foo_work(&foo_dt, &foo_z);

/* record the hydro timestep vote */
    pck->dt      = foo_dt;
    pck->dt_zone = foo_z;

    PA_ACCM_TIME(pck->time);
```

```
    if (first)
       {pck->space = (double) Ssp_alloc - pck->space;
        first = FALSE;};

    return;}
```

Note: the use of PA_MARK_TIME and PA_ACCM_TIME.

The above list covers the main simulation code. Most simulation codes will need to have some means of building initial state files (restart dumps). This means writing the following additional routines for the initial value problem generator code:

6. Build functions to handle each command

Each input command that the initial value problem generator will have to parse requires a function which does the actual parsing and takes the appropriate actions. One function may handle many different commands, but it is best to keep the functions simple and that usually makes the correspondence between commands and functions one to one. For an existing code, the coding for these routines probably already exists but may have to be reorganized.

```
  void command_1h()
     {char *token_1, *token_2;
      int i1, i2;

      token_1 = PA_get_field("TOKEN_1", "COMMAND_1", REQU);
      token_2 = PA_get_field("TOKEN_2", "COMMAND_1", REQU);

      i1 = PA_stoi(PA_get_field("INTEGER_1", "COMMAND_1", REQU));
      i2 = PA_stoi(PA_get_field("INTEGER_2", "COMMAND_1", REQU));

      N_foo_1  = i1;
      N_foo_2  = i1*i2;

      return;}
```

Note: the use of PA_get_field and PA_stoi.

7. Build a function which installs the commands

Not all packages require specific generator commands. The generic commands supplied by PANACEA may be sufficient. However, if commands have been defined in the previous step they must be made known to PANACEA. All commands are installed in a hash table which is used by PANACEA to process ASCII input files or interactive commands.

```
foo_cmmnds()

   {PA_inst_c("command_1", NULL, FALSE, 0, command_1h,

              PA_zargs, commands);


/* optionally initialize some constants */

    N_foo_1 = 0;

    N_foo_2 = 0;


    return;}
```

Note: the use of PA_inst_c and the reference to the function command_1h.

8. Build a function to intern the package variable in the database

Before a state (restart) dump can be made two things must happen. First, the data read in from the input deck must be transformed into data structures usable by the simulation routines. Second, those data structures must be made available to the database, a process called interning. A function is supplied by the code developer whose responsibility is to ensure that the package's contribution to the database is complete and ready to be dumped.

```
intern_foo()

   {

    swtch[4]   = N_foo_1;

    swtch[5]   = N_foo_2;


    PA_INTERN(foo_var_1, "variable_1", int);


    return;}
```

Note: 1) the use of PA_INTERN; and 2) the setting of the control array values (here swtch).

With this done the only remaining task is to produce the counterparts to A.C, the source file which contains the main entry point for the initial value generator program, and B.C, the source file which contains the main entry point for the run time simulation program. Whether they are written in C or FORTRAN is immaterial. Fundamentally, they define the packages for the code system, initialize everything (or have PANACEA do it), read the input, pass control to whatever routines coordinate the main activity of the code, and do the appropriate clean up at the end.

9. Install the package in the generator code

Make the foo package known to the PANACEA initial value problem generator code, A.

```
PA_gen_package("foo", foo_cmmnds, NULL, NULL, def_foo,

              cont_foo, intern_foo);
```

10. Install the foo package in the simulation code

Make the foo package known to the PANACEA main simulation code, B.

```
PA_run_time_package("foo", NULL, NULL, def_foo, cont_foo,

                    init_foo, main_foo, NULL, NULL);
```

This may seem like quite a lot of work. For an existing code, in some sense it is akin to starting over and building a new code, but the big advantage is that most of the coding already exists, has been tested, and has a base of test input problems. On the other hand, the payoff can be quite large rather quickly. The output data for the code will be completely portable. There is at least one existing system for visualizing the data produced which is very flexible and available on all sorts of computer platforms. Furthermore, for code systems which share the same computational meshes, there may be readily available routines that will provide interim or permanent capabilities (mesh generators, common packages, etc.).

The Monolithic Approach

One way to approach converting an existing code system into a PANACEA code system is to start by viewing the entire simulation code as a single, global package. The advantage is that a few relatively simple pieces of interface coding will get the conversion going fairly quickly. Since there will be small amounts of interface coding, the connections to PANACEA services can be written one by one and tested before proceeding. The entire process becomes more of an evolutionary one than a major code rewrite exercise.

The basic steps are the same as outlined in the previous section. The advantage is that each of those steps only has to be done once (to start with).

Once the monolithic code is running reliably, the smaller more modular packages can be extracted from the original global package. This process amounts to splitting the monolithic interface apart into a series of smaller interfaces. Once all of the identifiable packages have been extracted from the original monolithic package, the remainder should be a fairly natural PANACEA style global package.

ABC: A PANACEA Code

ABC is a simple 2d hydrodynamics code which shows how a PANACEA code system is structured. The A code is an initial value problem generator, the B code is the main simulation code, and the C code is a post processing code which transforms PANACEA output files for a particular visualization system.

Although ABC was constructed solely as a test of PANACEA, it still illustrates some of the important points that must be addressed when building a PANACEA code. The construction of a PANACEA package is logically the same whether the entire package is being built from scratch or whether an existing package is being turned into a PANACEA package.

The parts of the source to ABC which should provide helpful examples of the material documented in this manual are given here. The actual code for ABC is available and interested readers can contact the author for copies of it.

TOUL

TOUL is a stand alone utility to transpose time history files written by PANACEA into a family of ULTRA files.

A family of time history files is conventionally named

```
name.tdd
```

where name is the base file name and dd are base 36 digits (0-9a-z).

The resulting family of ULTRA files will be named

```
name.udd
```

Its usage is:

```
toul [-d] [-h] [+/-o] [-f files | -l files | base]
     -d  Turn on debug mode for memory info
     -f  Use explicitly specified TH files
     -h  Print this help message and exit
```

-l Use internal links in TH files
+o Process TH files in the order specified (default)
-o Process TH files in reverse order

See the discussions of PA_th_trans_link, PA_th_trans_name, PATRNL, and PATRNN for additional information regarding the -f, -l, -o, and +o options.

There are a few distinct modes of using TOUL.

First, if you have a complete set of time history files with a single base name, you can supply the base name to TOUL. For example, suppose you have a set of time history files foo.t00 through foo.t1s. You can type:

toul foo

This will result in a family of ULTRA files beginning with foo.u00.

Second, you may have a family of files from which some are missing or because of a restart you may have a different base name for some files in the family. You can use the -l option in such cases. For example, suppose that files foo.t09 through foo.t12 are missing in the previous example. You can type:

toul -l foo.t1s foo.t08

The curves in the resulting ULTRA files will have a gap corresponding to the missing data from foo.t09 through foo.t12.

Third, you may wish to transpose an explicit set of file. You can use the -f option to accomplish this. For example, suppose that you just want to see the data from foo.t1e through foo.t1g in the ongoing example. You can type:

toul -f foo.t1e foo.t1f foo.t1g

Because this is the most general level on which you can approach the transposition process, the resulting curves in the ULTRA files can look a little strange. You may have to get used to the gaps that might result from pieces of data widely spaced in time.

PANACHE

PANACHE is a tool to automate the process of building interface coding between numerical simulation code packages and PANACEA. PANACHE does this by processing a dictionary which contains compact and concise descriptions of variables, functions, commands, and actions.

A dictionary contains three basic kinds of information: global code system declarations; one or more package descriptions; and directives for generating and possibly compiling the code system.

When the dictionary has been read in and processed, PANACHE will generate: for each package defined, the interface between the package and PANACEA; and the module(s) which contain the main routine(s) and some related high level control functions.

The code generated this way can be compiled and linked with the simulation packages into the component parts of a simulation code system (generator and simulation codes or a combined code).

Certain conventions are used for sake of clarity.

```
[...]* means zero or more of the items in the brackets

[...]+ means one or more of the items in the brackets

The following variables are pre-defined in PANACHE:

NULL - used to indicate a null pointer (ala NULL in C).
```

Commands

This section describes the commands which PANACHE understands. It uses these commands to build up an internal view of the simulation code system as well as the package interfaces. A set of such commands defining a code system and its interface is called a dictionary.

Global Definition Commands

These commands define information that is not particular to a single package but is global to the entire simulation system.

PANACEA contains various routines which can control a simulation. These routines do NOT purport to be completely general. They are however fairly general and in any case can be used as templates for code systems which need to do something different. PANACHE will let you use these generic routines if you supply certain information.

The system scalars define the time step controls used by the high level PANACEA simulation managers. The system files define the naming of state files, post-processor files, edit files, and PVA files.

The system variables are REQUIRED if you do not specify your own simulation manager routines. If you want to use your own routines, you must specify one of the following two sets of system functions:

```
"generator-init-problem"

"simulation-init-problem"
```

```
"simulation-run-problem"

"simulation-fin-problem"
```

or

```
"combined-init-problem"

"combined-run-problem"

"combined-fin-problem"
```

system-scalars

Specify scalar variables which contain global information required by PANACEA.

```
Usage: (system-scalars
        t        ; system time
        tstart   ; system start time
        tstop    ; system stop time
        dtf      ; system time step fraction
        dtfmin   ; minimum time step fraction
        dtfmax   ; maximum time step fraction
        dtfinc   ; maximum time step fractional increase
        cycle    ; system time cycle number
        nzones)  ; number of computational units (zones)
```

system-files

Specify ASCII variables which contain the names of files used by PANACEA.

```
Usage: (system-files
        sfname   ; state file name
        edname   ; edit file name
        ppname   ; pp file name
        pvname)  ; PVA file name
```

system-functions

Specify functions which PANACEA will call under certain conditions. Each function is paired with a name. All pairs are optional, but functions must be paired with the names. The current meaningful names for PANACEA are:

"generator-init-problem"

"simulation-init-problem"

"simulation-run-problem"

"simulation-fin-problem"

"combined-init-problem"

"combined-run-problem"

"combined-fin-problem"

"generator-termination"

"simulation-termination"

"interrupt-handler"

"generator-error-handler"

"region-name-to-index"

"pre-intern"

"domain-builder"

"mapping-builder"

Only specified functions can be called. PANACEA doesn't attempt to call it if you don't define it. Any functions used here that are not PANACEA functions (PANACEA functions all begin with 'PA_' or '_PA_') must be declared in a declare-function statement.

Usage: (system-functions [name function]+)

Example: (system-functions

```
"generator-termination"   global_end_graphics
"simulation-termination"  B_term
"interrupt-handler"       PA_interrupt_handler
"generator-error-handler" A_gen_err
"region-name-to-index"    part_reg
"pre-intern"              load_reg
"domain-builder"          LR_build_domain
"mapping-builder"         LR_build_mapping)
```

Package Definition Commands

These commands are used for specifying package level details of the interface between PANACEA and the simulation package.

command
Defines generator commands.

> *Usage:* (command [name spec]+)
> spec := function handler |
> type array index
> type := integer | real | string

The second form of spec allows an application to name an element of an array as a command which sets the value of the element.

> *Example:* (command
> side sideh PA_zargs
> dimension integer swtch 1)

control
Defines PANACEA control items.

> *Usage:* (control [type index val comment]+)
> type := ascii | integer | real

> *Example:* (control
> real 1 0.0 "current problem time (internal units)"
> ascii 8 NULL "input deck name (ASCII)"
> integer 14 0 "generation time mesh dumps if TRUE")

declare-function
Declare one or more functions pertaining to the package. These functions may be ones that are defined in hand coded routines but need to be referenced in generated code. They may also be given so that a complete package header can be generated. At this writing declarations involving pointers must be enclosed in quotes.

> *Usage:* (declare-function [function-spec]+)

function-spec := type name arg-list
arg-list := (arg*)
arg := "type [‘*‘]*name"
type :=

Example: (declare-function
void load_reg ()
"char *" foo ("int s" "double t")

declare-variable
Declare one or more variables pertaining to the package. These variables may be ones that are referenced in hand coded routines but need to be referenced and or defined in generate code. They may also be given so that a complete package header can be generated. At this writing declarations involving pointers must be enclosed in quotes.

NOTE: these are NOT database variable definitions, see def-var for that purpose. These are variables which the compiler must be able to correctly understand. Database variables are created at run time and are mainly irrelevant to the compiler.

Usage: (declare-variable [variable-spec]+)
variable-spec := [class] type name dims initial-value
class := static | extern | const
type := any defined type (e.g. double)
dims := ()
initial-value := number

Example: (declare-variable
double bar ()
static int foo 3)

def-var
Define one or more database variables.

Usage: (def-var variable-spec type
mesh-spec
[attribute-list
[unit-list
[init-val
[init-fnc

		[data-pointer]]]]])
variable-spec	:=	variable | ([variable]+)
type	:=	any defined type (e.g. double)
mesh-spec	:=	mesh | ([mesh]+)
attribute-list	:=	[attribute]*
attribute	:=	scope | class | persist | allocation | center
scope	:=	restart | runtime | demand | edit | defn
class	:=	requ | optl | pseudo
persist	:=	keep | cache-f | cache-r | rel
allocation	:=	static | dynamic
center	:=	zone | node | edge | face | un-centered
unit-list	:=	(unit*)
unit	:=	user-defined-unit | per | rad | ster | mole | Q | cm | sec | g | eV | K | erg | cc
init-val	:=	a declare-variable'd name
init-fnc	:=	a declare-function'd name
data-pointer	:=	a declare-variable'd name

Example:

```
(def-var (zone reg_map) double zonal (restart requ zone))
(def-var lindex long spatial (restart))
(def-var knob-a integer ()
         (restart static)
         ()
         kav () knoba)
(def-var cache-f-4 double spatial
         (restart cache-r zone)
         (K)
         pi)
```

generate
Causes PANACHE to generate compilable code from the currently defined dictionary.

Usage: (generate language type name1 [name2])

language	:=	C | C++ | F77
type	:=	+ | -

The language argument allows the user to specify the target language of the generated code. At this time only C and C++ are support. Plans are for future versions to produce Fortran77 code. If type is ‘+’ use SX extensions; and if ‘-’ use PANACEA only. The argument name1 is the name of the generator code if name2 is also given; otherwise it is the name of the combined code. name2 is the name of the simulation code if given. Two names implies two codes and one name implies a combined code.

Example: (generate C + fubar)
(generate C - fubar)
(generate C++ + a b)
(generate C++ - a b)

include
Specify one or more header files to be included by the header for the current package.

Usage: (include [file]*)

The files are the literal names of header files which the preprocessor must be able to find as specified.

Example: (include foo.h somewhere/bar.h)

mesh
Define a mesh for the package. A mesh here corresponds to an array dimension.

Usage: (mesh [mesh-spec]+)
mesh-spec := name variable

Example: (mesh zonal N_zones)

package
Begins the definition of the named package. The definition continues until the next package command or until a generate command. The package name must be a legal C or Fortran identifier since it will be used to generate function and variable names.

Usage: (package name)
Example: (package "global")

package-functions
Defines the package functions that will be called by PANACEA. The following list shows all of the package functions available in PANACEA:

"install-generator-commands"

"install-type-definitions"

"intern-variables"

"define-units"

"define-variables"

"define-controls"

"initialize"

"main-entry"

"post-process-output"

"finish"

"install-pp-commands"

As with the system-function command, package functions must be paired with the name by which PANACHE knows them. Functions which PANACHE will generate need not be defined this way since PANACHE will implicitly define them.

Usage: (package-functions [name function ...])

Example: (package-functions
"intern-variables" global_intern
"initialize" global_init
"main-entry" global_main)

typedef
Define a structured data type. Database variables may be declared to be of any type (or any depth of pointers) which is either primitive or typedef'd. The primitive types are: char, short, int, long, float, double. Pointers are denoted with *'s as in C.

Usage: (typedef name [member]+)

member := "type [*]*member_name"

type := any defined type (e.g. double)

Example: (typedef foo
"int *a"
"double b")

Example: ABC Dictionary

In this section we present a sample dictionary. This dictionary generates the interface coding for the ABC PANACEA test code. The actual listing is broken down into sections to allow for discussion.

The ABC System Information

In many PANACEA codes there is some information about the entire code system which does not belong to any one package. Global time information is one example. The system of file families managed by PANACEA is another. At this point, this information is required to generate coding for the main routines where the A and B (or the combined AB code) start up.

```
;
; ABC.DICT - dictionary for ABC the PANACEA test code
;
; Source Version: 2.0
; Software Release #92-0043
;
; include "pact-copyright.h"
;

;---------------------------------------------------------------------------

; SYSTEM WIDE INFORMATION

;---------------------------------------------------------------------------

; NOTE: this information is required!

(system-scalars
        param[1]         ; system time
        param[2]         ; system start time
        param[3]         ; system stop time
        param[4]         ; system time step fraction
        param[5]         ; minimum time step fraction
        param[6]         ; maximum time step fraction
        param[7]         ; maximum time step fractional increase
        swtch[3]         ; system time cycle number
        swtch[12])       ; number of computational units (zones)

(system-files
        name[2]  ; state file name
        name[3]  ; edit file name
```

```
        name[4]  ; pp file name
        name[5])         ; PVA file name

(system-functions
        "generator-termination"          global_end_graphics
        "interrupt-handler"              PA_interrupt_handler
        "region-name-to-index"           part_reg
        "pre-intern"             load_reg
        "domain-builder"                 LR_build_domain
        "mapping-builder"                LR_build_mapping)
```

The ABC Global Package

The global package in the PANACEA model has a special place in that it may provide for a computational mesh or some other central activity. This package provides a 2d mesh generator for the 2d hydrodynamics package in the next section.

```
;-----------------------------------------------------------------------------

; GLOBAL PACKAGE

;-----------------------------------------------------------------------------

(package "global")

(package-functions
         "intern-variables"              global_intern
         "initialize"            global_init
         "main-entry"            global_main)

(declare-function
         "PM_set *" LR_build_domain ("char *base_name" "C_array *arr" "doub
         "PM_mapping *" LR_build_mapping ("PA_plot_request *pr" "double t")
         void global_end_graphics ()
         int part_reg ("char *s")
         void make_mesh ()
         void clearh ()
         void curveh ()
         void parth ()
         void recth ()
         void sideh ())

(declare-variable
         double t ()
         double dt ()
         int cycle ()
         static int N_one () 1)

(mesh    zonal N_zones
         nodal N_nodes
         scalar N_one)

(def-var (zone reg_map) double zonal
         (restart requ zone))

(def-var (node nodet k-ratio l-ratio k-product l-product) double nodal
         (restart requ node))
```

```
(def-var (r rx ry) double nodal
         (restart node)
         (cm))

(def-var (vx vy) double nodal
         (restart node)
         (cm per sec))
(def-var (t dt) double scalar
         (requ)
         (sec))

(control         ascii 1 NULL "geometry - slab, cylinder, sphere"
         ascii 2 NULL "restart file name (PDB)"
         ascii 3 NULL "edit file name (ASCII)"
         ascii 4 NULL "post processor file (PDB)"
         ascii 5 NULL "graph file name (PDB)"
         ascii 6 NULL "X host name - :0.0"
         ascii 7 NULL "display type MONOCHROME or COLOR"
         ascii 8 NULL "input deck name (ASCII)"
         ascii 9 NULL "PS/EPSF conformance"

         integer 1 0 "dimensionality - 0, 1, 2, 3"
         integer 2 0 "geometry - 0 none, 1 slab, 2 cylinder, 3 sphere"
         integer 3 0 "current cycle number (INTERNAL)"
         integer 4 0 "N_plots, number of plot requests (INTERNAL)"
         integer 5 0 "N_regs, number of regions (INTERNAL)"
         integer 6 0 "N_nodes, number of logical points (INTERNAL)"
         integer 7 0 "lrz, index of last real zone (INTERNAL)"
         integer 8 0 "frn, index of first real node (INTERNAL)"
         integer 9 0 "lrn, index of last real node (INTERNAL)"
         integer 10 0 "maximum k index - kmax (INTERNAL)"
         integer 11 0 "maximum l index - lmax (INTERNAL)"
         integer 12 0 "N_zones, number of logical zones (INTERNAL)"
         integer 13 0 "frz, index of first real zone (INTERNAL)"
         integer 14 0 "generation time mesh dumps if TRUE"
         integer 15 1 "mesh generation method"
         integer 16 1 "impose mesh generation constraint"

         real 1 0.0 "current problem time (internal units)"
         real 2 0.0 "start time (internal units)"
         real 3 1.0e100 "stop time (internal units)"
         real 4 1.0e-4 "initial/current dt fraction of tstop-tstart"
         real 5 1.0e-6 "minimum dt fraction of tstop-tstart"
         real 6 0.1 "maximum dt fraction of tstop-tstart"
         real 7 2.0 "fractional increase in dt per cycle"
         real 8 1.0 "1.0 -> 9 point, 0.0 -> 5 point differencing of spatial
         real 9 1.0 "1.0 -> 9 point, 0.0 -> 5 point differencing of ratio m
         real 10 0.5 "0.0 -> k oriented, 1.0 -> l oriented")

(command         make make_mesh PA_zargs
         clear clearh PA_zargs
         curve curveh PA_zargs
         part parth PA_zargs
         rect recth PA_zargs
         side sideh PA_zargs
         dimension integer swtch 1
         start-time real param 2
         stop-time real param 3
         geometry string name 1
         host string name 5)
```

The ABC Hydro Package

The ABC hydro package is a simple 2d hydro package which does actual hydrodynamics so that certain meaningful tests of PANACEA can be carried out (especially regarding post processing and package control functions).

```
;---------------------------------------------------------------------------

; HYDRO PACKAGE

;---------------------------------------------------------------------------

(package "hydro")

(mesh spatial N_zones)

(def-var (ne ni) double spatial
         (restart requ zone)
         (per cc))

(def-var (pe pi) double spatial
         (restart requ zone)
         (erg per cc))

(def-var rho double spatial
         (restart requ zone)
         (g per cc))

(def-var (rpdve rpdvi rqdve rqdvi) double spatial
         (zone)
         (ev per sec))

(control         integer 1 1 "hydro switch"
         integer 2 0 "dimensionality - 0, 1, 2, 3"
         integer 3 0 "geometry - 0 none, 1 slab, 2 cylinder, 3 sphere"
         integer 4 0 "N_zones, (INTERNAL)"
         integer 5 0 "hydro edits: 0 none, 1 major, 2 minor"
         integer 6 1 "include radiation momentum deposition"

         real 1 0.2 "Courant timestep multiplier"
         real 2 2.0 "rotational q multiplier"
         real 3 1.0 "linear q multiplier"
         real 4 1.5 "quadratic q multiplier"
         real 5 0.2 "fractional volume change for timestep"
         real 6 0.2 "fractional q weighted volume change for timestep"
         real 7 0.2 "dt multiplier for dv/v timestep control"
         real 8 1.0 "pressure multiplier (p = mult*nT)"
         real 9 0.0 "minimum sound speed"
         real 10 1.0 "radiation pressure multiplier"
         real 11 1.5 "half the number of degrees of freedom"
         real 12 1.0e-6 "minimum hydro dt frac"
         real 13 0.1 "maximum hydro dt frac"
         real 14 2.0 "fractional increase in dt per cycle"
         real 15 1.0 "fraction of Christensen q to use")
```

The ABC Miscellaneous Package

This package is used to test various facets of PANACEA functionality. It has almost no significance as far as a simulation goes. It demonstrates more of the ways in which the dictionary defining function of PANACHE can be used.

```
;---------------------------------------------------------------------------

; MISC PACKAGE

;---------------------------------------------------------------------------

(package "misc")

(typedef foo
         "double real"
         "double imaginary")

(mesh    spatial N_zones
         scalar N_one)

(declare-function
         void load_reg ()
         void materialh ()
         void componenth ())

(declare-variable
         static int kav () 10
         static double kbv () 2.71
         static double pi () 3.1415
         int knoba ()
         double knobb ())

; RESTART VARIABLES

(def-var ab double spatial
         (restart requ zone)
         (g per mole))

(def-var zb double spatial (restart requ zone))

(def-var (te ti) double spatial
         (restart requ zone)
         (K))

; TEST VARIABLES

(def-var lindex long spatial (restart))

(def-var findex float spatial (restart))

(def-var knob-a integer nil
        (restart static)
        nil
        kav nil knoba)

(def-var knob-b double nil
        (restart static)
        nil
        kbv nil knobb)

(def-var cache-f-1 double spatial
```

```
        (restart requ cache-f zone)
        (K))

(def-var cache-f-4 double spatial
        (restart cache-r zone)
        (K)
        pi)

; RUNTIME VARIABLES

(def-var tr double spatial
        (demand zone)
        (K))

(def-var T-cmplx complex spatial
        (demand zone)
        (K))

(def-var cache-f-2 double spatial
        (cache-f zone)
        (K)
        pi)

(def-var cache-f-3 double spatial
        (cache-r zone)
        (K)
        pi)

(command material materialh PA_zargs
         component componenth PA_zargs)
```

Completing ABC

Once the system level information and the packages have been defined, PANACHE can be directed to generate the source code for a variety of configurations and in this case the compilation executed.

```
;---------------------------------------------------------------------------

; COMPLETION

;---------------------------------------------------------------------------

;(generate + "ab")
(generate - "ab")
;(generate + "a" "b")
;(generate - "a" "b")

(compile)
```

Related Documentation

PANACEA is a part of the PACT portable code development and visualization tool set. It depends on most of PACT. Readers may find useful material in the other PACT documents.

The list of PACT Documents is:

PACT User's Guide	UCRL-MA-112087	
SCORE User's Manual	UCRL-MA-108976 Rev.1	
PPC User's Manual	UCRL-MA-108964 Rev.1	
PML User's Manual	UCRL-MA-108965 Rev.1	
PDBLib User's Manual	M-270 Rev.2	
PGS User's Manual	UCRL-MA-108966 Rev.1	
PANACEA User's Manual	M-276 Rev.2	Current Document
ULTRA II User's Manual	UCRL-MA-108967 Rev.1	
PDBDiff User's Manual	UCRL-MA-108975 Rev.1	
PDBView User's Manual	UCRL-MA-108968 Rev.1	
SX User's Manual	UCRL-MA-112315	

Some additional references to work that readers may find interesting and useful are:

1. Harold Ableson, and Gerald Jay Sussman, Structure and Interpretation of Computer Programs, MIT Press, Cambridge, Mass. (1986).

2. G. O. Cook, Jr., "ALPAL, A Program to Generate Physics Simulation Codes from Natural Descriptions", Int. J. Mod. Phys. C 1, 1990.

Last modified: Fri Jul 16 15:28:23 PDT 1999

Modified by: rodrigues2@llnl.gov