Application Programmer's Guide to the Plasma Control System (preliminary draft)

John Ferron, Mike Walker, Ben Penaflor, Bob Johnson
General Atomics

February 28, 2011

Contents

Obtaining this document

This document is available online with a WWW browser at:
`http://web.gat.com/pcs/master/master.html`.
The Postscript file for printing (about 471 pages) can be obtained at:
`http://web.gat.com/pcs/master.ps`.
The pdf file for printing (built to be compatible with Acrobat 5 or later) can be obtained at:
`http://web.gat.com/pcs/master.pdf`.

Chapter 1

Introduction

The plasma control system (PCS) is designed so that its feedback control capability is configurable by any programmer with knowledge of the C programming language. A change in control capability is performed by adding a control "algorithm" that specifies the method for control of some quantity.

From the point of view of someone who wants to implement a feedback control algorithm, the plasma control system software can be considered in three parts:

- The basic infrastructure of the system that provides the general purpose tools and support routines around which the specific control applications are built.
- Installation-specific code that defines the input and output signals, control categories and real-time computer setup for a particular location where the control system is installed.
- The application specific code that implements the desired control algorithms.

This document describes how to write the application specific code that implements the desired control algorithm.

This document is organized as follows:

1. Section 2 provides detailed background information that is needed in order to design the application specific code. An understanding of the content of this section is a prerequisite for writing feedback algorithm code.
2. Sections 3, 4, 10 and 11 describe how to specify and document the control application code, how the code should be structured, how to test the code and how to install the new application into the control system.
3. Section 14 is a reference section containing detailed descriptions of the specific support routines that are referenced in the other sections.

The reader should be familiar with operation of the PCS as described in the user's manual. It is assumed here that the installation-specific code (as summarized earlier in this section) has already been provided.

For the most efficient implementation of a new control algorithm, the following procedure is suggested.

1. Write an outline describing what the control algorithm must do in real time in order to implement the required feedback control function.

2. Do as much testing of the algorithm as possible outside the environment of the control system. For instance, implement any complex calculation routines and check their results using any convenient programming language and environment. This type of preparation is easiest outside of the constraints imposed by the programming environment of the control system. Ideally, the end product of this work would be C language code that can be transferred directly into the control system software.

3. Write an outline description of the data required by the real time code and the corresponding data items that will appear on the user interface.

4. Produce the standard control algorithm documentation and specifications (see Sec. 3) for the software to be implemented in the control system. The result of this work will be a design for how the algorithm fits into the control system and how the control system operator will interface with the algorithm.

5. Implement the control system code for the algorithm and install it into the control system (Secs. 4 and 11).

6. Test the implementation of the algorithm in the control system (Sec. 10).

Chapter 2

Background information for application code authors

2.1 PCS overview

The PCS computing hardware can have a variety of configurations depending on how it has been implemented at a specific installation. Here are a couple of examples.

1. A combination of "host" computer and "real time" computers (see Fig. **??**).
   - The host is a general purpose computer that provides the basic services such as disk storage and network interface on behalf of the real time computers. Typically the host computer accesses the real time computers over a bus connection such as the VME bus.
   - One or more "real time" computers. These are the computers that actually do the feedback control during a tokamak discharge. The interface to these computers is provided by the host computer. The host can start and stop programs on the real time computers, access the memory of the real time computers, etc.
2. A network of "self-hosted" real time computers (see Fig. **??**). In this case the real time computers have an operating system that allows them to provide their own network and disk services.

In all cases there is some way for the real time computers to communicate in real time. Also, there are typically one or more computers that are used to interface with the PCS operators. The PCS is designed to accommodate a variety of computing hardware configurations.

The PCS software has several major components, each of which runs in a separate process on one of the computers (see Fig. **??**) in the control system.

- user interface: allows the operator to specify the input data for the control system.

- waveform server: manages the database of information on how the discharge should run.
- lock server: synchronizes the PCS processes with the tokamak shot cycle.
- real time host process: provides host computer services for the real time computer (manages loading the real time computer memory etc.).
- real time code: executes on the real time control computer to do the actual control of the plasma. There is real time code that is specific to each of the real time computers.
- message server: receives log messages from the other processes and makes them available to the "view log" component of the user interface. Also keeps track of whether the other processes seem to be operating properly.

There can be one real time host process for each of the real time processes or the two parts can be combined into one process. A complex control application could have a piece of code in each of these processes (except for the lock server and message server which have no application specific code). The most basic control application would have code written for just the waveform server process and the real time process.

Most of the application specific code is written in the C language. The exceptions are an occasional optimized routine written in assembly language for the real time computer and user interface routines which are written in the IDL language (see Research Systems `http://www.rsinc.com`) .

The C language code for a particular application is collected in a "master" file for the application. The master file contains several sections of code, each of which has a specific function. Although the various sections of code may be included into one or more of the processes listed above when the code is compiled, in order to understand how to implement a control application, the code author should consider primarily the **function** of the code rather than exactly where the code is placed at compile time.

Sections 4 and 14 of this document describe how to write these master files. To add a control application to the PCS, all that is required is to write the appropriate master file and add the name of the file to a list of algorithm master files. The new application is automatically integrated with the existing applications in the PCS when the PCS code is recompiled.

2.2 Model for the control system operation

The PCS is designed to operate in a mode in which starting at some instant the control system begins operation, follows a preprogrammed time evolution and then stops operating at some preset time. The duration of the interval in which the control system is operating can be very long, so the control system could be viewed as being capable of handling steady state systems. The basic model for input that is presented to the operator, though, is that input parameters vary as a function of time starting at some instant that is synchronized with other hardware. The PCS provides feedback control of various quantities to hold them at programmed values. The programmed values can vary in time in a manner typically determined by the operator. Also, other switches and data can be provided that can vary in time.

During setup for the shot, the memory of each real time process is loaded with the required data by its corresponding host real time process. Then, after the discharge starts, each real time process is on its own. Each real time process uses the data that was loaded into its memory to decide what to do and how to run the discharge.

So, the application code has two main parts:

- Code that runs in real time to control the tokamak.
- Code that prepares and provides the required input data for the real time computer.

In the remainder of this section of background information, the first group of subsections discusses the code that runs in real time. The second group of subsections discusses the code that prepares and provides the required input data for the real time process.

2.3 Feedback control overview

The hardware that actually does the real time control typically has these components (see Fig. **??**):

- Digitizers (A/D converters): these convert the analog signals from the tokamak diagnostics to digital values. The diagnostics give measurements of quantities that are to be controlled.
- Real time computers, each of which has access to the data from the digitizers.
- D/A converters: these circuits convert a digital value to an analog value. These are used to send commands from the control system to the various “knobs” that are available to change the discharge parameters (power supplies, gas valves etc.).
- Digital I/O: A digital input/output board is used to interface to external devices such as neutral beams that can be controlled with a binary value (i.e. on/off). The digital input/output board is also used to acquire input data that can be represented as an on/off value.

The method used by the real time computers to communicate with each other and with the digital and analog input and output devices will vary between control system installations. For instance, often the VME bus is used by the real time computer to communicate with the D/A converters and the digital I/O board. Normally the acquisition of the data and the transfer of the command values to the output devices is transparent to the application programmer since there are some standard routines available to perform these functions that are provided as part of the installation-specific code of the PCS.

During the shot, the work of each real time process is basically quite simple. It executes a single code loop continuously until the shot is over:

1. The digitizers are used to sample each of the input diagnostic signals.
2. For each control category a control algorithm is executed. The control algorithm typically performs the following steps.
   (a) The value of some quantity to be controlled is calculated.
   (b) The calculated value is subtracted from the preprogrammed target value of that quantity. The difference is the feedback "error."
   (c) Any necessary massaging of the error (such as calculating the PID value, see Sec. 5) is performed.
   (d) The required new control knob value is calculated.
   (e) The control knob value is written to the appropriate D/A converter.
3. Loop back to step 1.

This feedback control cycle continues until a preprogrammed time to stop is reached.

The preprogrammed target value can change with time, and other input data that is either constant or varies with time can be used.

Of course, this is a just simple outline of an algorithm and the actual procedure can be specified in detail by the application programmer.

2.4 Organization of applications

Feedback control applications are grouped into "categories." Each category can be responsible for the control of as many quantities as desired. The way in which the various quantities in a category are controlled at any given time during the shot is determined by the "algorithm" chosen by the operator.

A category is simply a convenient grouping of quantities to be controlled. It provides some organization for the convenience of the operator and the category defines the granularity of the choice of the control algorithm.

A control algorithm is the complete definition of how the various quantities for a given category are controlled. It defines what data is provided to the real time process and the procedures used in the feedback control. There can be as many algorithms defined in the control system as desired for each category.

The way in which the control system should behave as a function of time is defined by a shot "phase." The definition of a shot phase includes the specific control algorithm that should be used and the time evolution of the quantities specified as input data for the algorithm. For each control category, the control system operator can define as many shot phases as desired.

The time evolution of a discharge for a specific control category is defined by specifying a "sequence" of shot phases. There is a separate sequence of phases defined for each category. The control system operator can specify that the discharge should begin by using one particular phase, then switch to a second phase, and then to a third phase etc. The operator can specify that the control system should switch the shot phase as often as desired. A given shot phase can be used more than once during a shot to, for instance, repeat some behavior more than once.

For each control category, a single shot phase is in use at any given time during a discharge. Thus, only one algorithm is active at any given time for a given category and a switch in shot phase is used to switch the control algorithm that is in use. Or, multiple phases using the same algorithm but having different input data could be defined.

The sequence of phases to be used during a shot is defined in one or more "phase sequence waveforms" for each category. There is a "primary" phase sequence that is always used first at the beginning of a discharge. Normally the primary phase sequence defines the time evolution of the shot phases to be used. However, if some code determines during the discharge that it is necessary, an asynchronous switch can be made from one phase sequence to another. This type of asynchronous switch to an alternate phase sequence can be used to react to an exceptional event in a preprogrammed manner.

Each control category is defined by code contained in a "category master file." The division of the control applications into categories is normally performed by the programmer who installs the PCS software. Each control algorithm is defined by code in an "algorithm master file." This algorithm master file is included in the list of algorithm files defined for the particular category. Most work in implementing control applications is in writing the algorithm master files. How to write an algorithm master file is the primary focus of this document.

2.5 Data structures in the real time process

This section describes the data structures available to the code executing on a real time processor. There is a fixed set of data that is loaded into the memory for a real time process before the shot. These data specify completely what the time evolution of the discharge should be. In addition, there is the set of data values obtained by digitizing the signals from the various diagnostics on the tokamak. Finally, there is a set of data structures into which the algorithm code can record results in order to pass these results to other code in the control system or to archive data for analysis after the discharge.

The data structures available in real time are shown in the following list. For some structures, a reference is given to a section that provides more detailed information. Note that the PCS software allows the group of real time processors to contain a mixture of processor architectures. The various architectures might use differing numbers of bytes to hold some of the standard C language data types, specifically the types `pointer` and `long`. Several of the standard data structures provided by the PCS on the real time processor are specifically designed to consist of data types that are the same size on each of the processor architectures: `float` and `int`. Because both of these data types require 4 bytes, several of the standard data structures are designed to be arrays of 4 byte values. Data of type `float` and `int` may be intermixed within these arrays.

- real time heap layout structure: Abbreviated "rtheap" in many places in the code, this is a structure that contains various constants and pointers to all of the other structures in the memory of the real time process. During the setup procedure for a shot, the memory of the real time process is organized for each shot in the manner required for that shot. Thus the memory is primarily treated as a "heap" of free memory to be allocated. Additional memory (like shared memory) can also be used for certain data sections, especially for communicating between the real time processors (see Sec. 13.2). For information on how to locate the various data structures in the memory of the real time process see Sec. 13.4.

- data vector `rtheap->datavector`: an array of values of type `float` obtained from the digitized diagnostic signals. The exact meaning of the values in this array will be installation dependent. In general, these values are equal to the digitized value (usually in digitizer bits) minus a baseline value obtained at the beginning of the discharge.

- physics vector `rtheap->physicsvector`: an array of values of type `float` which contains the input values converted to physics units. The exact meaning of the values in this array will be installation dependent. In general, these values are equal to (the digitized value minus a baseline value) multiplied by calibration values to convert from input units to physical units.

- target vector `rtheap->adtarget`: an array of values having 3 parts

  – continuous floating point values (type `float`).

  – step function floating point values (type `float`).

  – step function integer values (type `int`).

  See Sec. 2.5.1 for more information.

- pointer target vector `rtheap->pointer_target`: an array of type `pointer`. Note that values of type `pointer` will use either 4 or 8 bytes depending on the processor architecture. See Sec. 2.5.1 for more information.

- shape vector `rtheap->shapevector`: a general purpose array of locations to store computed values of type `float` or `int`.
- error vector `rtheap->errorvector`: an array where the feedback error values are written.
- P vector `rtheap->pvector`: an array into which is written the result of the PID operation on the error vector (see Sec. 5).
- pidtau vector `rtheap->pidtauvector`: an array of pointers to lookup tables used in the PID calculation (see Sec. 5).
- function vector`rtheap->functionvector`: an array of pointers to the application specific subroutines that are called to do the feedback computations (see Sec. 2.5.2).
- command vectors: two arrays in which the values written to any output devices (like D/A convertors) are stored.
  - fpcommand vector `rtheap->fpcommand`: a floating point array containing the exact calculated values of the commands for the tokamak control knobs.
  - intcommand `rtheap->intcommand`: an integer array containing the values actually written to any output devices connected to tokamak control knobs. The floating point values are clipped between maximum and minimum levels and then converted to integer format before being output.

  For most feedback control applications, the values in the intcommand and fpcommand vectors will be the same apart from the conversion to integer and the clipping. However, some control applications may vary from this convention.
- parameter data `rtheap->current_parameter_data`: blocks of memory into which the application specific code can have preloaded any desired data (see Sec. 2.5.3 and Sec. 2.11). Some of this data can be parameter data blocks that are initialized to contain zeros that the algorithm code can use for any purpose, typically for storage of values between control cycles. The sizes of these blocks are specified by the algorithm-specific code. These blocks of memory are allocated prior to the shot and fixed in size during the shot. See Sec. 2.5.4 and Sec. 2.11.10 for more information.
- communication vector `rtheap->adcombuffer`: an array of memory locations on each of the real time processors into which code on a different real time processor can write data. This vector exists in order to provide a simple method for communication between applications running on two different real time computers. For more information see Sec. 2.9.

- communication vector `rtheap->adcombufloop`: an array of memory locations which is a copy of the adcombuffer and which all algorithms should access. Normally, the adcombuffer is copied to the adcombufloop at the end of a control cycle so the content of adcombufloop will not change during the control cycle. For more information see Sec. 2.9.

- DMA buffer `rtheap->dmabuffer`: a two-dimensional array to store the input values. The exact meaning of the values in this array will be installation dependent. In general, this array contains room for all input data channels, the current time, the previous time, and any digital input/output values for each cycle that the user wants to save.

- installation buffer `rtheap->install_buffer`: an installation specific structure usually containing system-wide values needed for communicating between real time processors. For more information see Sec. 15.2.10.

During any given loop of the feedback control cycle, input data values must be gotten either from digitizers or other sources. This is usually done by the algorithm that is in the acquisition category. So the control cycle begins with the collecting of input data, putting it into the data vector (usually in bits), subtracting off a baseline value calculated before the discharge begins, setting the values of `rtheap->currenttime` and `rtheap->previoustime`, and filling the contents of the DMA buffer so the input data can be archived after the discharge. In addition, the raw data can be converted to physics units and those values can be put into the physics vector.

In general, during one call to an application specific routine, the data vector values (or better, the physics vector values) are used to compute the present value of the quantity to be controlled, a value from the target vector that defines the desired value for the controlled quantity is subtracted to produce an error value, and the error value is written to a predetermined location in the error vector. Some intermediate computation results might be written to predetermined locations in the shape vector in order to archive the results or communicate the results to another application specific function.

The remainder of the work, through the actual output of any control values to the output devices, is usually performed by standard routines provided by the PCS application specific code or the infrastructure. The P vector and command vectors are used primarily by these routines. However, the application code author can provide custom routines.

Note that all allocation of memory is performed during setup for a shot. There is no dynamic allocation or freeing of memory during the process of real time control. Since the real time computer does not use virtual memory, only the physical memory present with the computer is available. Preallocation of all memory ensures that before the shot begins, it can be established that the amount of memory required to accomplish the control specified by the operator is actually available.

2.5.1 The target and pointer target vectors

For the application specific code, the target vector is the primary source of input data that defines how the feedback loop should function. The value that a controlled quantity should have (the "target value") is specified in this vector. Also, various other input data that can vary in time are provided in the target vector, such as switches to specify options in the application, feedback gain values etc. A control algorithm would typically reference one or more individual elements of the vector to obtain input values.

The pointer target vector is similar except that it contains only pointers. A separate vector is provided because the size of a pointer can vary depending on the processor architecture and can be different from the size of the values in the target vector.

During one feedback control loop, the contents of the target vector and the pointer target vector are fixed. Between control loop cycles, the content of these vectors is updated by the PCS infrastructure code to have the values specified by the PCS operator for the current time in the discharge.

The target vector has three blocks of values:

1. The first block contains the "continuous" target vector elements. The continuous elements are floating point values that vary "smoothly" in time; that is, the values change at a rapid enough rate and in small enough steps that the changes in the controlled quantity appear to be slow and smooth.

   The time evolution of a continuous target vector element is specified by a series of connected line segments in the parameter space (time, controlled quantity). The vector value makes a step along the line segment at time intervals equal to a "phase clock tick" which is typically 1 millisecond.

   The continuous target vector element values are calculated in real time from the equation of a line segment:

   $$\text{value} = \text{slope} \times \text{time} + \text{intercept}.$$

   The slope and intercept used in the calculation change at the time of a new vertex derived from the waveforms the operator specifies.

2. Float step target: this is a block of floating point values that change in time in a step function manner. The value is constant in time until the time to make a change. Then the float step target vector element jumps to the new value. This type of target vector element is useful for specifying values that are essentially constants and do not need to vary smoothly in time. During real time execution there is overhead associated with calculation of the continuous target vector values so if a value does not need to vary continuously there is a performance advantage to using a float step value.

3. Integer step target values: this is a block of integer values that change in time in a step function manner. These are useful as switches.

The values in the pointer target vector are typically pointers to a particular block of data. The pointer values can change in time, for instance, to point to a particular element of an array of structures. The specified element might be given by a waveform value.

2.5.2 The function vector

The complete set of applications for control of the tokamak is broken into categories and at any given time during a discharge there is a specific control algorithm executing for each category (Sec. 2.4). Each of the control algorithms could be composed of several parts implemented in separate functions. Some of the procedures for the algorithm in one category could depend on some of the results from another category. So, it might be necessary to execute one part of a given category's algorithm, then part of the algorithm for another category, then back to the first category etc. That is, although the control of the tokamak is broken into categories for ease of interaction with the PCS operator, the algorithms for the individual categories might interact and the order of execution could be important.

The order of execution of the various parts of each algorithm is determined by the content of the function vector. The function vector is an array of pointers to functions. The programmer who installs the PCS determines what functions must execute on each of the real time processors and what the order must be. Then, each category is assigned one or more elements of the function vector, the order of these assignments depending on the control applications. The algorithm master file specifies the function that is to execute in each of the time slots allocated to the category of that algorithm. The pointers to these functions are loaded into the function vector at the appropriate points in the discharge. For instance, when the algorithm for a given category changes during the discharge, the new function pointers are loaded into the function vector.

For example, consider the following function vector organization.

```
1. category A, 1 element
2. category B, 1 element
3. category A, 1 element
4. category C, 1 element
5. category B, 1 element
6. category A, 1 element
```

Category A uses 3 locations in the function vector, category B uses two locations and category C uses 1 location. The execution of routines for the three categories is intermixed so, for instance, the execution of the second function for category A (in element 3 of the function vector) can use data calculated by the first function executed for category B in element 2 of the function vector. The final step in the work for category A might be to send results to the D/A converters and this might be the purpose for element 6 of the function vector, the third element for category A.

The author of an algorithm for category A can specify up to three different functions to be executed. The algorithm master file contains two arrays that are used to specify the functions to be executed: one is a list of element numbers in the function vector and the other is a list of names of functions that correspond to those function vector elements.

The algorithm does not need to make use of all elements of the function vector assigned to its category. The category master file specifies a default function for each element of the function vector assigned to that category. These function assignments will be used unless specifically overridden by the algorithm definition. Normally these default functions do nothing.

2.5.3 Parameter data

The target vector provides a series of single values that can have some prespecified time evolution. This is a primary data input source for algorithm code.

However, the way the target vector values are set is defined and controlled by the PCS infrastructure and the number of values is limited. An algorithm may have need for large blocks of data that have a structure that is specific to the algorithm.

For this purpose, the PCS provides support for an area of memory for each shot phase that contains "parameter data". These data might consist of lookup tables, large data structures such as multidimensional arrays etc. To provide for organization of this area of memory, the parameter data are divided into "blocks," each of which might contain data for a particular purpose or with a particular organization.

These data are considered "static" because the contents of the memory locations where the data are stored are not changed by the PCS infrastructure in real time. This contrasts with the content of the target vector which is updated at intervals in real time to provide time varying input values.

There is a great deal of flexibility available to the programmer in the way parameter data blocks are handled. This allows parameter data blocks to be used for many possible purposes (Sec. 2.11.1).

Section 2.11 provides the details on how to utilize the parameter data facility.

2.5.4 Scratch memory introduction

An algorithm may require space on the real time processor in which to hold miscellaneous data. For this purpose the capability to allocate "scratch memory" is provided. Scratch memory is simply one or more areas of memory, initialized during the first stages of preparation for each shot to contain zeros, that can be used by an algorithm for any purpose. Each shot phase has its own set of scratch memory regions.

Scratch memory behaves like memory allocated for a C language variable with the `static` attribute in that its content is saved between calls to functions. Also, unless specifically modified by the algorithm code, the memory content is retained on entry and reentry to a shot phase.

Scratch memory should be used instead of static variables because the space available for storage of static variables can be limited on the real time computer, whereas the scratch memory is allocated as part of the "heap" memory that is organized for each shot. If a given algorithm is not in use for a particular shot, no scratch memory is used for it whereas the static variables are always allocated for all algorithms compiled into the real time code.

The scratch memory can be written by any of the routines executed for the algorithm during shot setup by the host_cpu process, or by the real time process. Also, the scratch memory can be written by the phase entry or reentry routine of the algorithm and by the algorithm routines on the real time processor executed each control cycle.

More details on the usage of scratch memory are provided in Sec. 2.11.10.

2.5.5 Global and static variables on the real time processor

In general, global and static variables should not be used as part of the code for the real time processor. As explained here, use of these types of variables can cause the real time algorithm to fail under some circumstances. Use of global and static variables also doesn't fit within the structured programming model of the PCS so the result is code that is harder to understand.

As an example, a temptation is to use global variables or static variables (perhaps visible only in the context of a specific function) to hold values for use later by the real time code. However, this will fail in the case that a user creates more than one phase using the same algorithm. The same variable storage space will be used for both of the phases and the data for the first phase executed will be overwritten when the second phase is used. This would be a particularly repeatable problem when the common variable storage space is written by a pre-shot initialization routine. In this case the code for both phases will be executed before the real time control begins.

The use of static variables defined within the context of a particular function is deceptive. It is possible to have the compiler initialize the variable. For instance:

```
static int flag = 0;
```

In this case the temptation is to believe that the variable is set to 0 at the beginning of each discharge. In fact, the variable is initialized only when the executable is first started; that is, when the PCS is started up. The PCS code is not restarted for each discharge. So, the static variable will retain the value it had at the end of the previous discharge unless it is reinitialized during a pre-shot initialization routine for each discharge.

The way to handle this that fits within the structured programming environment of the PCS is to use a scratch parameter data block. Each phase has its own scratch and parameter blocks so the problem of overwriting data will not occur. Scratch parameter data block content can be easily initialized prior to each discharge using a pre-shot initialization routine.

Another temptation is to use a global variable to hold a pointer to a particular data structure such as a parameter data block. If necessary, the pointer vector should be used to hold a pointer for this purpose. There are functions available that will cause a pointer to, for instance, a specified parameter data block to be placed in a specified element of the pointer vector. Alternatively, the locations of most data structures can be attained through the `rtheap` structure (for instance, see Sec. 2.11.7).

In some cases a global variable might be useful to deliberately share data among several shot phases. However, the method that fits the structure of the PCS programming model is to share data through the various vectors. To share data, for instance, a parameter data block can be created to hold the data. Each phase will have its own block. Then, put a pointer to one of these blocks in an element of the pointer target vector. A pre-shot initialization routine can fill the vector element with the pointer. The routine will be executed for every phase using the algorithm. The last pointer written will be the one actually used in real time. The real time code then reads the pointer vector element to get the pointer to the data area that will be shared by all phases.

2.5.6 Allocation of space in the real time data structures

Each of the real time data structures (e.g. target vector, shape vector, error vector etc.) is divided into blocks of values that are designated for the use of a particular category. A given category might have several blocks assigned to it in a given data structure.

An algorithm written for a given category has access to the elements of the data structures assigned to all categories on a real time processor. Usually, the algorithm should only access the elements set aside for its category, but it is possible to access the elements of a data structure created for another category. In such a case, it is important to order the functions in the function vector correctly so the content of these elements can be shared between categories. Thus, the function that writes a value to an element needs to come before the function that reads the value. Also, it is important to check that an algorithm that writes a value is actually active before the value is read.

So, one of the first steps in the procedure of designing a control algorithm is to decide what the input and output data should be and where the values should be located in the elements of the data structures available to the algorithm's category. The algorithm author should be very careful not to write into data structure areas assigned to other categories unless this is desired (generally, the algorithm author should assign an element in the category where that element is to be written so there is no confusion).

2.6 Pre-shot setup and post-shot cleanup

In the period before, during and after a discharge the waveform server, the lock server, the host real time processes and the real time processes cooperate to prepare for the shot, execute the real time control and clean up after the shot. An application-specific control algorithm will almost always have code that executes during real time to do the necessary feedback control, but in addition it can optionally also have code that participates in the pre-shot setup and post-shot cleanup. It is also possible to specify code that is executed when a shot phase is entered in case some initialization must be done for each execution of a given shot phase. All of this optional code is specified in the algorithm master file. The remainder of this section gives an outline of the pre-shot setup and post-shot cleanup procedures and shows where the algorithm-specific code can be executed. Section 4 describes how to incorporate algorithm-specific pre-shot and post-shot code into the algorithm master file.

"Lockout" is the term given to the time during the setup for a tokamak discharge when the PCS begins preparation for a new shot. At this time the operator is "locked out" of the control system so that no more changes in the setup for the discharge can be made. The following functions are performed after lockout to execute the setup of the real time processors for the discharge.

1. The required allocation of memory for each of the real time processes is determined. A check is made to ensure that the required amount of memory is actually available.

2. The memory to be used for each real time process is initialized to zero. This ensures that all data structures that are not otherwise initialized start out at a known value.

3. The necessary processed data is transferred from the waveform server to each of the real time processes.

4. In each of the host real time processes, application-specific pre-shot initialization code is executed.

   (a) Any code specific to the real time computer is executed.

   (b) Any code that is specific to the control categories that will execute on that real time computer is executed.

   (c) For each shot phase of each category that executes on the real time computer, any pre-shot initialization code that is specific to the algorithm assigned to the shot phase is executed (see Sec. 4.1).

At this point the control system has been initialized for the upcoming discharge.

Actually, after lockout is reached, the operator can still force changes to be accepted for the new discharge by "unlocking" the control system. Then, to restart setup for the discharge the operator must "relock." After the relock the complete shot setup procedure described above is executed again.

After "final lockout" time is reached, it is no longer possible to unlock/relock. The control system is committed to execution of the shot (except that up to the "primary start time" the control system will recognize the "abort" trigger). The real time control process starts and does the following.

1. Some miscellaneous pre-shot initialization in the real time process.
2. Application-specific pre-shot initialization code is executed in the real time process.
   (a) Any code specific to the real time process is executed.
   (b) Any code that is specific to the control categories that will execute in the real time process is executed.
   (c) For each shot phase of each category that executes in real time, any pre-shot initialization code that is specific to the algorithm assigned to the shot phase is executed (see Sec. 4.6.3).
3. Each real time process then waits for the "clock clear trigger" that is used to clear the counter that is used to keep track of time during the discharge (see Sec. 2.8). After this trigger the control software can reliably determine the elapsed time from the beginning of the discharge.
4. The real time process monitors the clock counter, waiting for the "primary start time." This is the time at which the primary phase sequences begin.
5. The function that actually does all of the real time control is called. It is at this point that real time feedback control activity begins. The real time control continues until the preprogrammed time to stop.

   During the real time control, for each category there is a list of shot phases that may be used. The definition of the shot phase provides the control algorithm to be executed. It is possible that the selected algorithm requires some code to be executed when a shot phase is entered, for example, to clear out an array or sample some data value for use during the remainder of the phase. The algorithm code can specify two functions for this purpose (see Sec. 4.6.3).

   (a) A function that is called when a shot phase is entered at the beginning of the phase.
   (b) A function that is called when a shot phase is entered at any time other than at the beginning of the phase.

   If it is not necessary to differentiate between these functions, the same function call can be used in both cases.

6. Miscellaneous post-shot functions are performed by the real time process.
7. Application-specific post-shot cleanup code is executed.
   (a) For each shot phase of each category that executes in real time, any post-shot cleanup code that is specific to the algorithm assigned to the shot phase is executed.
   (b) Any post-shot cleanup code that is specific to the control categories that will execute on that real time computer is executed (see Sec. 4.6.3).
   (c) Any post-shot cleanup code specific to the real time process is executed.

After the discharge, the host real time process performs post-shot functions.

1. Miscellaneous post-shot cleanup code is executed.
2. Application-specific post-shot cleanup code is executed.
   (a) For each shot phase of each category that executes in real time, any post-shot cleanup code that is specific to the algorithm assigned to the shot phase is executed (see Sec. 4.1).
   (b) Any post-shot cleanup code that is specific to the control categories that will execute in that real time process is executed.
   (c) Any post-shot cleanup code specific to the real time process is executed.
3. At this point, the lockout server is told that the real time portion of the shot has been completed. When all of the real time processes have declared the end of the real time portion of the shot cycle, the lockout server begins to supervise the archival of the data for the shot. Each host process, one at a time, is given permission to archive data.
4. While the host process is waiting for permission to do the general purpose data archiving, for each shot phase of each category that executes in real time, a post-shot "calculation" function is called if one is provided (see Sec. 4.1). This allows each algorithm to do some algorithm-specific analysis that doesn't need to be part of the real time portion of the shot sequence. The host process is otherwise idle while waiting for permission to archive data, so this portion of the shot cycle is a good opportunity to perform this type of calculation.
5. When the host process receives permission to archive data, for each shot phase of each category that executes in real time, a post-shot "archiving" function is called if one is provided (see Sec. 4.1). This allows an algorithm to do some algorithm-specific archival of data that isn't provided for in the archival of the standard PCS data.
6. If necessary, the raw data collected by the real time process are archived in the tokamak database.

7. The memory of the real time process is dumped to a file (see Sec. 13.4).

8. The target vector, pointer target vector and pidtau vector values are recreated and added to the memory dump file.

9. The control system response vectors (target, shape, error, P, fpcommand, intcommand) are archived in the tokamak database (see Sec. 2.7).

2.7 Archival of the PCS response

The PCS processes a large amount of data during a discharge without operator intervention. Even though a real time display of PCS signals is now possible, the discharge evolution is so rapid that an operator would not be able to understand the data on a real time display rapidly enough to detect whether the PCS is functioning as expected. For this reason it is desireable to save some samples of the PCS input data and computed results for analysis after the discharge.

For sampling of computed results, the error, shape, P and command vectors are actually configured as arrays of vectors. During any given control cycle, the address of these vectors is fixed. However, between control cycles, the PCS infrastructure code can change the pointer to these vectors so that the data computed in the next cycle are written to a different location than on the previous cycle. In this way, the results from the previous cycle will not be overwritten and so will be preserved. At the end of the discharge, the array for each vector holds samples of the time evolution of the content of the vector.

The "data vector" is a fixed location into which the PCS writes the values from the digitizers on each cycle after subtraction of the baseline and conversion to floating point format. However, the buffer into which the raw data from the digitizers are written by the data acquisition hardware is configured as an array of buffers. At fixed time intervals the PCS infrastructure code changes the address used for the input buffer, leaving behind the data acquired during the previous cycle. In this way the input data are sampled.

The time during the discharge that the data sampling starts, the time interval between samples and the number of samples saved can be set by the PCS operator. In most cases, the starting time and interval are configured by the installation to be waveforms. These parameters thus allow the various data values to be sampled at roughly equal time intervals during a single, specified portion of the shot. Note that the time stamp associated with each set of data is the time at which the digitizers were triggered to sample the analog input data. The processed data marked with this time stamp are the data produced from the raw digitizer data with the same time stamp. These days the time interval between triggers to the digitizer is usually uniform so the interval between data samples should be constant.

Since there is limited space to store data, it is usually desireable to vary the sampling rate throughout the discharge. For example, before a discharge begins, the sampling rate may be longer since little may happen. To provide more flexibility, there is a provision to allow the code for an algorithm to force the data from a particular control cycle to be saved. This is done by writing the value `0xfffffffe` to the unsigned integer `datasample_flag` in the `rtheap` structure. For example,

```
    rtheap->datasample_flag = 0xfffffffe;
```

The value of `datasample_flag` is reset to 0 after the data are saved so the algorithm must set `datasample_flag` on every cycle for which saving of data is to be forced.

Some control algorithm code might need access to the content of a vector element as computed on the previous control cycle. For instance, if a result is computed on one cycle and then used on the next cycle. Since the address of the various vectors can change between cycles, the address of each vector on the previous cycle is preserved and is accessible to the algorithm code. These addresses are available in the `rtheap` data structure.

The sampled raw and computed data of the PCS are stored in the tokamak data archive for each shot. Each element of a data vector is assigned a pointname. So, to plot the value of a PCS vector element vs. time the operator simply plots the correct pointname.

However, the meaning of a particular vector element depends on the algorithm. Therefore, the pointname assigned to a vector element is determined by the algorithm that was actually in use during the discharge. Since the algorithm in use during a discharge can change in time, the data from a given vector element from the time periods that a given algorithm was executing are archived under the name determined by that algorithm. The data from time periods that a different algorithm was executing are archived under a different pointname. If multiple algorithms assign a vector element to be used for the same purpose, then all of those algorithms can assign the same pointname to the chosen vector element. All of the data from vector elements with the same pointname assigned are archived together under one pointname, independent of whether the algorithm in use varied.

Part of the process of writing algorithm code, then, is to identify which vector elements that are used by the algorithm are to be archived in the tokamak database and what the pointname for each vector element should be. This information is included in the algorithm's master file. In addition, the algorithm author can assign an "inherent number" and a "physics zero" to each vector element. When the data are accessed from the tokamak data base, the inherent number and physics zero can be used to convert the units of the archived vector element values.

$$\text{physicsunits} = \text{inherentnumber} \times \text{data} + \text{physicszero}$$

In this way, the exact values that were used by the control system are archived, but it is also possible for the algorithm author to archive the conversion factors between the units as used by the control system and units that might be relevant to the operator. If no conversion is necessary, the inherent number should be 1.0 and the physics zero should be 0.0.

It is not necessary to archive every element of a vector that is used by an algorithm. Normally, only those that would be useful for later diagnosis are archived. There is a flag in the algorithm master file for each vector element that indicates whether the data for that element should be archived. It is, however, necessary to assign every element a name since the name is used often in PCS diagnostics and makes understanding the diagnostics much easier.

The target vector values are all predetermined input values, not computed or raw data acquired during the discharges. Therefore, the target vector is not sampled and saved during the discharge. Rather, the target vector element values are recomputed after the shot to obtain the values that they had on each cycle when the raw and processed data were sampled. The recomputed target vector data are also archived in the tokamak database using pointnames determined by the algorithm author.

The pointer target vector contains only pointers. The value of a pointer is not meaningful outside the context of the PCS software. So, the values in the pointer vector are not archived in the tokamak database. However, the pointer vector is reconstructed after the discharge just like the target vector and the values are saved in the memory dump file (see Sec. 13.4) for use in diagnostics.

2.8 PCS timing

Elapsed time during a discharge is measured by a counter that counts the cycles of a master clock. One cycle of this master clock is called a "fine scale clock tick." This is the finest time resolution available in the control system. (One fine scale clock tick is, for example, 1 microsecond).

Elapsed time during a discharge is also measured in units of a "phase clock tick." A phase clock tick is an integral number of fine scale clock ticks in length. For example, a phase clock tick is 1 millisecond, or 1000 fine scale clock ticks.

Changes in control system parameters that can vary with time are made at intervals of a phase clock tick. For instance, the continuous target vector values are recalculated at intervals of a phase clock tick. Since a phase clock tick defines the time resolution with which input values can be specified, the time of any waveform vertex specified by the PCS operator is always rounded to the nearest phase clock tick.

An elapsed time value during a discharge can be either an "absolute" time value or a "relative" time value.

- Absolute time values are measured with respect to the "PCS time zero" which is not the same as the plasma $t = 0$. The PCS time zero is at the time of the clock clear trigger (Sec. 2.6), a synchronous trigger that occurs, in a typical PCS installation, before the plasma $t = 0$.

- Time within a shot phase or within a phase sequence is measured relative to the start of the phase or phase sequence. This relative time is in units of phase clock ticks.

2.9 Using multiple real time processors in parallel

There can be multiple real time processors installed in the PCS. These processors may be in one or many separate computers. Each processor can run its own process and be given its own physical CPU number. Or a processor can be used to run a separate thread in order to do parallel processing. These processors operate in parallel during the discharge to perform the required control functions. There are several possible reasons for using multiple real time processors.

- Adding processors increases the total processing power that is available for real time computation so that more compute intensive applications can be implemented.
- The amount of memory available on a single computer may be limited, so adding computers adds to the total memory capacity of the system.
- The control applications may require a mixture of cycle times. For instance, there might be an application requiring a very fast cycle time and there may be applications that perform more computation so the cycle time for these applications would be slower. Typically an application requires all of the capacity of a computer. So mixing slow and fast applications may require multiple computers.
- It might be desirable to use multiple processors for a control algorithm for any or all of the three reasons listed above.

The multiple processors operate essentially independently during the discharge. Each of the processors has its own set of preloaded input data and its own code to execute. Each processor executes its own independent control cycle by calling the functions referenced in the function vector (Sec. 2.5.2). If synchronization of the control cycles on the individual processors is required, this can be implemented by control algorithm code.

The algorithm author specifies how the code and data are to be distributed among the multiple processors. Section 2.9.1 describes how this is done. If necessary, the real time algorithm code on one processor communicates with the code on other processors in order to synchronize execution or exchange data. Section 2.9.2 discusses the programming interface for interprocessor communication. Section 2.9.3 is an overview of the interprocess communication functions. Section 2.9.4 discusses the easy to use "real time message facility" to send data between two processors. Section 2.9.5 discusses some ways in which the multiple processors could interfere with each other that the algorithm programmer should be aware of.

The extent to which the real time measurements of diagnostic data are available on each of the processors will depend on the particular PCS implementation. This is normally handled by the algorithm executing in the data acquisition category. In the case that provides the most flexibility, the input data from all of the digitizers are transferred to all of the real time processors.

2.9.1 Task and data distribution

Each control category is assigned one or more processors on which to execute code in real time. Each real time processor assigned to a category is called a "virtual CPU." The available real time processors are called "physical CPUs." The PCS programmer who defines the PCS configuration specifies the mapping between virtual and physical CPUs. Usually a given category uses only a fraction of the available physical CPUs. In fact, often a category requires only one CPU. Usually, code for more than one category will execute on each physical CPU, but a given physical CPU will not necessarily execute code for all of the categories.

The mapping between virtual and physical CPUs is transparent to the programmer of a control algorithm. The algorithm programmer needs to know only the number of real time processors available for the control category to which the algorithm belongs. The algorithm programmer then distributes the work for the algorithm among this quantity of virtual CPUs.

A virtual computer is named with a capital letter, A, B, C etc., while physical CPUs are named with numbers, 1, 2, etc. The macros `CPUA`, `CPUB`, etc are used when specifying a virtual CPU. The macros `CPU1`, `CPU2`, etc. are used when specifying a physical CPU. The algorithm author will never actually need to use the physical CPU macros. All specification of a CPU in algorithm code refers to a virtual CPU. The algorithm programmer writes code to refer to a virtual computer so that the algorithm code is independent of the details of the PCS configuration. The correspondence between physical and virtual processors can then be altered without requiring changes in the algorithm code.

An exception to the practice of programming a control algorithm in terms of virtual CPUs rather than physical CPUs comes when there is an interaction with the algorithm code of another category. In this case, it is usually necessary to arrange the PCS configuration specifically so that a particular virtual CPU of one category is assigned to the same physical CPU as a particular virtual CPU of the other category.

The algorithm author must always specify the virtual CPU on which each piece of processed data and each section of code is placed.

Processed data (Sec. 2.10.2), data that are copied to the memory of the real time processor, are always created in the algorithm's `_vectors` function (see Sec. 2.10.5). This function calls other functions that create the content for a vector element or a parameter data block.

The functions that create the content of a vector element (many of these functions are discussed in Sec. 14.2) always have an argument that is used to specify the virtual CPU to which the processed data should be copied. If multiple vector elements on different CPUs are to contain the same values, the function to create the vector element data is called multiple times, specifying a different virtual CPU/vector element number combination in each case.

The functions that create a parameter data block (Sec. 14.3) have an options argument (Sec. 2.11.4.4) which can specify which virtual CPUs the parameter data block content should be copied.

Conditional compilation macros are used to specify which code is compiled as part of the executable image created for each real time processor (Sec. 4.15) and that processor's `host_cpu` process (Sec. 4.13). When a given executable is compiled, macros are defined that indicate the virtual CPU for each category that is assigned to the physical CPU to which the executable corresponds. The macro names have the format `category_CPUx`, where "`category`" is the identifier of the category in uppercase, and `x` is the uppercase letter indicating the virtual CPU. A preprocessor test should be made to determine if a given macro is defined to determine whether to compile each section of code. For instance, code for the category with the identifier `mycategory`, might look like the following.

```
#ifdef MYCATEGORY_CPUA
/* Code that should be available on CPU A for this category. */
#endif
#ifdef MYCATEGORY_CPUB
/* Code that should be available on CPU B for this category. */
#endif
```

Some useful macros that provide necessary information about the PCS configuration are defined automatically when the PCS is built. See Sec. 2.16 for a description of these macros. For instance, a given physical CPU will not necessarily run code for all of the control categories installed into the control system. So, in some cases it is necessary to know the "index" of a given category on the physical CPU. This refers to the place of the given category in the list of categories which execute code on the specified CPU. This index is available as a macro (Sec. 2.16.1). Also, routines used to communicate between the real time processors (Sec. 2.9.3) will need to be able to specify the physical CPU to which data should be copied. This is done using a macro that specifies the physical CPU that corresponds to a particular virtual CPU of a particular category (Sec. 2.16.3).

2.9.2 Communication between real time processors

In a PCS with multiple real time processors, there may be a need to transfer data in real time between processors. This might be the case, for instance, if an algorithm uses more than one processor.

The actual physical mechanism used to communicate data between processors will depend on the PCS hardware implementation. For instance, one processor might be able to write directly into the memory of another processor over a VME bus or a proprietary interprocessor connection. Or, it might be necessary to transfer data using some sort of networking system.

In order to make the code doing interprocessor communication independent of the actual communication hardware mechanism, a generic application programming interface (API) has been designed for use in the PCS. This API is described in this section. Most of the code that implements the interprocessor communication, though, is installation-specific. There is little about this API that is determined by the infrastructure. So a different API could be used if necessary. The primary reason to use this API is to facilitate collaboration between various locations where the PCS code is in use.

The PCS makes use, on each real time processor, of one large memory area which is referred to as the PCS "memory heap." All of the PCS preloaded data and data stored during the shot are placed in this memory heap. This includes all of the data structures discussed in Sec. 2.5. All of the PCS code executed on the real time processor has access to the structure of type `rt_heap_misc`, often called `rtheap` in the code. Using the content of this structure, algorithm code can locate any of the data in the PCS memory heap.

The interprocessor communication API is based on a communication method in which one real time CPU writes directly into the PCS memory heap on another real time CPU. There are two basic ways in which this is done.

1. A processor writes into the communication vector on another processor.

2. A processor writes directly into some arbitrary location in the PCS memory heap on another processor. Usually the location into which data are written is provided by the "real time message facility" function that registers a real time message (Sec. 14.4.1). This function is called by the algorithm in the waveform server code. An older method was to use a scratch parameter data block (Sec. 2.5.4) that was provided specifically by an algorithm for this purpose.

The choice of which of these methods to use depends on several factors.

1. The quantity of data to be copied. The communication vector is best for relatively small amounts of data (see below).

2. The point in the control cycle at which the data must be available to the code executing on the processor receiving the data. The communication vector content is synchronized with the control cycle (see below).

3. Any special methods required to access the memory area in which the data are written. This would be important on processors without hardware to automatically detect invalid cache memory content (see Sec. 2.9.5).

The communication vector and its purpose are discussed in Sec. 2.9.5. It is useful for communicating relatively small amounts of data. Consideration of the amount of data arises because the communication vector is copied at the beginning of each control cycle. If the communication vector is large this operation might consume an undesirably large amount of time.

The communication vector content is synchronized with the control cycle. As discussed in Sec. 2.9.5, there are actually 2 copies of the communication vector. Data are written into the copy at `rtheap->adcombuffer` and the algorithm code reads a second copy of the vector at `rtheap->adcombufloop`. The data at `rtheap->adcombuffer` are copied to `rtheap->adcombufloop` at the beginning of each control cycle. Thus, the data in this second copy remain unchanged during a control cycle so that all algorithm code is guaranteed to see the same data during the cycle. New data are written to the copy at `rtheap->adcombuffer` to provide data for the next cycle.

The synchronization of the communication vector with the control cycle makes it a good place for writing handshaking flags. For instance, an algorithm might need to transfer a large amount of data to another CPU where it would be used for analysis. The algorithm might use an alternating buffer method in which there are 2 buffers in which the data can be written. One is written while the content of the other is used for analysis. The buffer is written on a target CPU by another CPU at a time that is random with respect to the target CPU's cycle. When the process of writing the buffer has completed, a flag value is written to the communication vector on the target CPU indicating that the new data are available. On the next cycle, the algorithm on the target CPU detects the changed value of the flag in the communication vector and begins to use the new data for analysis.

As discussed in Sec. 2.9.5, on processors without hardware to detect invalid cache memory entries, it is essential to only read communication vector data from the copy at `rtheap->adcombufloop`. This is the copy that provides synchronization with the control cycle. However, if cache coherency is not an issue, the copy of the communication vector at `rtheap->adcombuffer` can be used for communication without synchronization with the control cycle.

The technique of writing directly into an arbitrary area in the PCS memory heap can also be used to communicate without synchronization with the control cycle.

2.9.3 The interprocessor communication functions

The following is a brief summary of the interprocessor communication functions. Additional details on the function calls can be found in Sec. 15.2.

- `rtcipc_write`: Write into a buffer on any of the PCS processors at an arbitrary location in the PCS memory heap (Sec.15.2.6).
- `rtcipc_write_commbuff`: Copy an array of values to a specified location in the communication vector on any of the PCS processors (Sec.15.2.7).
- `rtcipc_write_commbuff_float`: Copy a single value of type `float` to a specified location in the communication vector on any of the PCS processors (Sec.15.2.8).
- `rtcipc_write_commbuff_int`: Copy a single value of type `int` to a specified location in the communication vector on any of the PCS processors (Sec.15.2.9).
- `rtcipc_write_install_buffer`: Copy data to a general installation-specific location on any of the PCS processors (Sec.15.2.10).
- `rtcipc_write_timetostopflag`: Write a value to the timetostop_flag on a given PCS processor. (Sec.15.2.11).

- `rtcipc_read_timetostopflag`: Read a value of the timetostop_flag from a given PCS processor. Note that this requires another processor to write into this location. (Sec.15.2.12).
- `rtcipc_write_paramdata`: Write data to a parameter data block on a given PCS processor. (Sec.15.2.13).
- `rtcipc_write_rtmessage`: This is the most general function that copies data from one processor to another. See Sec.2.9.4 for more information about the real time message facility. (Sec.15.2.14).

The functions that write into the communication vector have a slightly special behavior in the case in which the CPU to which data should be copied is the same CPU from which the data originate. In this case the `rtcipc_write_commbuff*` functions simply copy directly into the communication buffer at `rtheap->adcombuffer`. The data in the buffer at `rtheap->adcombuffer` are synchronized with the control cycle, though, and it might be desirable to communicate data from one algorithm to another on the same cycle. To facilitate this, the `rtcipc_write_commbuff*` functions also copy the data to the control cycle synchronized communication buffer location (Sec. 2.9.5) at `rtheap->adcombufloop`. Usually the communication vector isn't used for passing data between algorithm code executing on the same CPU. A parameter data block or the shape vector are better choices. However, sometimes it is necessary to write algorithm code in a way that allows for the possibility that in some PCS configurations the communication might be to a different CPU and in some configurations the communication might be to the same CPU. In this situation the `rtcipc_write_commbuff*` functions would be used.

2.9.4 Real time message facility

As described in the preceding sections, the communication between real time processors is normally done with the interprocessor communication API. However, the functions in the API to communicate between real time processors require additional programming and setup. For example, the "installation" buffer needs to be defined along with offsets to each of its individual fields in order to use the `rtcipc_write_install_buffer` function. Or a scratch parameter data block needs to be created and the function `putblock_offsets` needs to be called to use the function `rtcipc_write_paramdata`. Also, the functions in the API do not handle memory regions other than the real time heap (see Sec. 13.2 for more information about memory regions) so "messages" cannot be shared between processors in a multi-processor computer.

An easier set of functions exists to minimize the steps required to communicate data between real time processors. This set of functions comprise the "real time message facility". With these functions a message is "registered" with an arbitrary name in the waveform server. The size of the message, the source processor, and the destination processor are given. This provides enough information so that memory can be allocated on the destination processor where the message is written, and, optionally, space can be allocated for a send buffer on the source processor. Different options allow for many different types of messages. All this information is stored in the waveform server so that the right amount of memory can be allocated on the real time processors.

Then, on the source real time processor, one function is used to get the address of the send buffer, and another function is used to send the message. Between the calls to these two functions, the real time code on the source processor simply needs to fill in the buffer. The destination real time processor then calls a single function to read the message. Since the details of the message are specified when the message is registered, the real time code just does the bare minimum to complete the communication.

The message can be sent between physical processors, between processors in a computer using shared memory, or even passed between different phases and/or categories on the same processor. The details of the message transfer and where exactly the buffers are located are hidden from the algorithm author. If the source and destination CPUs are reconfigured, the real time message facility should handle the new configuration automatically.

The following briefly describes each of these functions.

- `rtmessage_register`: Register a message. This function will store away the size of the message, the source processor, and the destination processor. This information is tagged with an arbitrary name. One argument is a character string which allows for many different options using keywords. Another argument specifies a data object descriptor which gives the size of the message on the real time processor. Optionally, this argument can be NULL and the size specified as a keyword. By default, space for a source buffer is set aside on the source processor and space for the destination buffer is set aside on the destination processor. Call this function in the waveform server part of the algorithm code (usually in the `alg_parameters` function). See Sec. 14.4.1.

- `rtmessage_get_buffer`: Get a buffer to write to. This will return a pointer to the "send" buffer if one was allocated. Call this function on the source processor. See Sec. 15.2.2.

- `rtmessage_write`: Send the entire message. The details of how the message is delivered depend on what kind of message was registered, but the algorithm author need not worry about these details in the real time code. Call this function on the source processor. See Sec. 15.2.3.

- `rtmessage_partial_write`: Send a partial message. This function allows the message to be sent in multiple pieces. It is left to the author to work out the details of these multiple transfers. Call this function on the source processor. See Sec. 15.2.4.

- `rtmessage_read`: Return a pointer to the received message. The details depend on what kind of message was registered, but the algorithm author only needs to use the address of the buffer that is returned. For example, this function may wait for a new message to arrive, but the details of waiting are taken care of automatically. Call this function on the destination processor. See Sec. 15.2.5.

2.9.5 Resource contention

There are situations in which the multiple real time processors in the PCS may interact. Typically the interaction involves sharing of some resource, so it places limitations on the performance improvement that can be obtained by adding more real time processors to the control system. Applications should be designed so that each of the multiple processors operate as independently as possible. Some of the possible interactions between processors are the following.

1. **Data acquisition**: The data acquisition method is installation-specific. However, some typical considerations are the following.

   - There may be a specific CPU with the job of collecting the input data and distributing it to the other CPUs. So, delays in this process could be important.
   - Because the real time CPUs might often use different cycle times, the data acquisition will typically be done by the CPU that has the fastest cycle time, the master. The data from all digitizers might be transferred in parallel to all CPUs, or each CPU might collect data from a subset of the diagnostics. Other CPUs would be slaves, receiving data at a multiple of the sample interval defined by the master clock. The slave processors normally execute a slower feedback cycle than the master processors so some of the data acquisition is actually triggered by the master CPU or the master clock more rapidly than required by the slave processors. The slave processors will typically ignore the excess sets of data.

2. **Shared communication bus**: The method of communication between real time CPUs might include a shared bus, such as the VME bus. More than one of the real time processors could have need to access the bus for data input or output. The use of the bus is arbitrated by the bus hardware. Access to the bus could be delayed if another processor is already using the bus. This can cause delays in the control cycle.

3. **Communication vector**: The various real time processors might have a need to communicate with each other in order to exchange data. This communication is performed by one processor writing into the memory of another processor (Sec. 2.9.2).

   The communication vector is provided to make this interprocessor communication simple by transparently handling the problems that can arise when copies of memory locations can be held in cache. In a multi-processor system with shared memory access, as in a VME system, a given memory location could be written by more than one processor. If a processor holds a copy of a memory location in its cache memory and a second processor changes the content of that memory location, the cached value will no longer be valid. On processors that do not have hardware to detect when a cached copy of memory becomes invalid, the invalid value from the cache memory could still be obtained. In this, memory buffers that are to be used for communication between processors should always be accessed in a manner that prevents those memory locations from entering the processor's cache.

The communication vector is provided to make the interprocessor communication easy for relatively small amounts of data. One processor provides data for a second processor by writing the data into the communication vector on the second processor. The address of the communication vector on other processors is available in the `rtheap->othercpu` structure for the appropriate processor. The second processor consults the communication vector whenever necessary to obtain data. It obtains the most recent value written there.

To avoid reading invalid communication vector data from cache, the PCS holds 2 copies of the communication vector. One copy is the location where new data should be written: `rtheap->adcombuffer`. This array should never be read by algorithm code. Instead, at the beginning of each control cycle the vector at `rtheap->adcombuffer` is copied into the vector at `rtheap->adcombufloop` using instructions that prevent the data at `rtheap->adcombuffer` from being copied into cache memory. Algorithm code should always read the communication vector content by accessing the array at `rtheap->adcombufloop`.

This technique also serves to synchronize the data in the communication vector with the control cycle. The data in `rtheap->adcombufloop` remains unchanged during a control cycle. New data written to the communication vector at `rtheap->adcombuffer` during a control cycle are recognized during the following control cycle.

2.10 Data processing and storage in the waveform server

The waveform server process is the central storage location for the PCS setup data. The graphical user interface interacts with the waveform server to make changes in the setup database. Within the waveform server, code provided by the algorithm author creates any special data that are required by the real time code. All of these setup data are stored within the waveform server in a standard format. When the moment comes for setting up for a new discharge, the appropriate data are copied from the waveform server to the real time processors. This process is handled by the PCS infrastructure code. The infrastructure makes the data available on the real time processor in a standard format.

So, the job of the algorithm author is to organize within the waveform server the data required by the control algorithm. The mechanisms to used to interact with data within the waveform server are described in this section.

2.10.1 Raw operator data

The PCS operator is insulated from the details of the real time processor operation. The operator does not need to know how many real time processors are actually used or the details of the data written into the memory of the real time processors.

Instead the operator specifies values for the physical quantities that are the easiest for the operator to understand. The values the operator specifies are determined by the control algorithm author.

The operator provides the control system with "raw input values" for various "data items." A data item is either a waveform (Sec. 2.10.1.1) or a set of static data (Sec. 2.10.1.2). The list of data items for an algorithm appears on the navigation window of the user interface. The user selects a data item for editing by clicking on its name in the list. There is an appropriate editing widget for each data item. The data provided directly by the operator is what is referred to as raw data.

Data items can, optionally, be grouped into "subsets" if this is desired by the algorithm author. If an algorithm uses many data items (e.g. more than 10), it is convenient to divide them into logical groups to make a specific data item easier for the operator to locate. Rather than searching through a long list of data items, the operator looks at a short list of subsets, selects a subset and then looks at a relatively short list of data items. The subset number 1 for an algorithm should contain the data items used most often by the operator since the list of data items in this subset is displayed by default.

2.10.1.1 Waveforms

A waveform is used to specify a value that can vary with time. The operator puts vertices on a plot of a quantity versus time, and this set of vertices is the raw input data for the waveform. In contrast, a static data item specifies a set of arbitrary constant values that can be customized to the algorithm.

A waveform has several characteristics determined by the algorithm author.

Waveforms are of two basic types:

- Continuous
- Step function

For a continuous waveform, the values the waveform takes between vertices are determined by connecting the vertices by a straight line.

A step function waveform has a constant value as a function of time equal to the Y axis value of the most recent vertex. Thus at each vertex the waveform steps to a new value.

As mentioned in Sec. 2.8, the X axis values of the vertices of a waveform are always rounded to be equal to an integral multiple of a phase clock tick. The Y axis for a waveform can be:

- continuous: the vertices can have any Y axis value.
- gridded: the vertices must have a Y axis value that falls on a uniformly spaced grid. The values specified by the operator are rounded onto this grid.
- discrete: the algorithm author specifies a list of values that the Y value of a vertex can have. The value specified by the operator is rounded to the nearest of these discrete values. The discrete values do not have to be uniformly spaced and can be labeled on the waveform plot with a text string so they can represent one of a list of choices for the operator such as "high", "medium", "low" etc.

2.10.1.2 Static data items

A data item is either a waveform or a set of static data. The list of data items for an algorithm appears on the navigation window of the user interface. The user selects a data item for editing by clicking on its name in the list. A waveform is used to specify a value that can vary with time. In contrast, a static data item specifies a set of constant values.

Parameter data blocks (Sec. 2.11) are used to build static data items. A static data item is composed of one or more "associated" parameter data blocks. The data are distributed among the various blocks in a manner that is convenient for the control algorithm. When a static data item is edited or restored from an archived PCS setup the associated parameter data blocks are handled as a group.

The data item name for a static data item must be the same as the prefix of the parameter data block name (Sec. 2.11.4.3) for each of the associated parameter data blocks making up the static data item. This is what associates the various blocks with each other.

The names of waveforms and static data items both appear in the navigation window data item list. When the user selects a waveform, the waveform editing interface is used to change the vertices defined for the waveform. When the user selects a static data item, usually a separate window will be created that has the correct interface to allow modification of the parameter data block(s) that make up the selected static data item. Static data items are identified in the data item list by an ellipsis (...) that is appended to the data item name.

Both waveforms and static data items are declared to exist by the algorithm code by including a "waveform definition structure" in the data item descriptor list (Sec. 4.7). The structure was originally designed to specify a waveform, but in order to specify a static data item, the same structure is used and certain elements of the structure contain information specific to static data rather than a waveform (Sec. 4.7).

- The `description` string in the waveform structure must begin with the string "StaticData". This identifies the data item as static data.

  The portion of the description string following "StaticData" is used to specify the format of the static data contained in the data item. For instance, for a data item that is an array of matrices, the description string could indicate this and also indicate the matrix size. For example, the description string might be “StaticData: 16x20 matrix”. The total length of the description string is limited to 40 characters.

The format portion of the description string allows determination of compatibility between the parameter data block formats in two static data items when restoring from an archived PCS setup. When a data item is restored from an archived PCS setup, its description string must match the description string of the data item that it will overwrite.

The list of static data item formats must be maintained in a central location for use by all algorithms. This is to ensure that when a particular string is chosen to describe a static data item format, it is unique along all format name strings that have ever been used in a particular installation of the PCS. By convention, this central location is the file `installdefs.h`. In this file a macro is defined for each static data item format name. By convention the macro name begins with `SD_`. For example,

```
/*
                             "<--------  40 characters  ------------->"
*/
#define SD_ACQSETUPDATA      "StaticData: Acq. Setup Data"
#define SD_MATRIX_18x20      "StaticData: 18x20 Matrix"
```

Then, the macro is used in the data item descriptor to define the description string.

- The `xunits` string in the waveform structure contains the name of an IDL routine that should be called when the data item is selected by the operator to create the user interface for viewing and editing the appropriate parameter data blocks.
- Other unused elements of the waveform structure can, optionally, be used to pass parameters to the IDL routine. This depends on the requirements of the editing/viewing user interface routine.

The format of static data can be unique to the algorithm and is determined by the algorithm author. Typically, the algorithm author writes a custom X window interface to allow the operator to adjust static data values.

A static data item has the following primary components.

- One or more parameter data blocks that hold the actual raw data that the operator edits.
- One or more parameter data blocks that hold auxiliary data (e.g. labels for the editor widget).
- An IDL procedure that implements the editor for the user interface that allows the PCS operator to modify the raw data. Because many static data items have a unique format, it is often necessary to write a customized editor procedure.

To create a static data item, first decide whether one of the standard static data item formats (Sec. 2.12) can be used. Using a standard static data item format is much easier than doing all the work to create a custom static data item. Here are the required steps to create a static data item using one of the standard formats.

1. If not done already, fill in the following fields in the descriptor (Sec. 4.6.1) for the algorithm: `paramarchivefcn`, `paramrestorefcn`, `parameterfunction`, `paramforhostfcn`, `paramforuserfcn`, `paramfromuserfcn`, `paramsizefcn`. These fields in the algorithm descriptor provide the names of functions that provide some of the parameter data block tools for the algorithm. These fields are only necessary if the algorithm uses parameter data blocks. See Sec. 4.6.1, 2.11.4.1 and 2.11.5 for more information.

2. Add the data item descriptor to the descriptor array for the algorithm (Sec. 4.7). Use the macro for the "description" string that has been defined for the chosen standard static data format. See the discussion earlier in this section for additional information. In the `xunits` string in the descriptor, insert the name of the IDL procedure that implements the user interface editor for the chosen static data item type (Sec. 2.12).

3. Add code to the `alg_parameters` function (Sec. 2.11.4.1) for the algorithm to create the necessary parameter data blocks and initialize each one with default data. Use the function that is provided for the chosen standard static data item type (Sec. 2.12). Note that all of the parameter data blocks for the static data item are considered to contain "raw" data so that they are always initialized in the `alg_parameters` function.

If it is necessary to create a custom static data item, here is an outline of the required steps.

1. If not done already, fill in the following fields in the descriptor (Sec. 4.6.1) for the algorithm: `paramarchivefcn`, `paramrestorefcn`, `parameterfunction`, `paramforhostfcn`, `paramforuserfcn`, `paramfromuserfcn`, `paramsizefcn`. These fields in the algorithm descriptor provide the names of functions that provide some of the parameter data block tools for the algorithm. These fields are only necessary if the algorithm uses parameter data blocks. See Sec. 4.6.1, 2.11.4.1 and 2.11.5 for more information.

2. Decide what the data item name will be. This is important because this name must be used consistently throughout the components of the data item. See the discussion in Sec. 2.11.4.3.

3. Decide what the "format" definition for the static data item will be. This must be included in the "description" string in the data item descriptor. See the discussion earlier in this section for additional information.

4. Add the data item descriptor to the descriptor array for the algorithm (Sec. 4.7).

5. Decide what the organization of the data will be in the associated parameter data blocks that make up the static data item. Take into account the capability to modify the data format in the future while maintaining compatibility with the data already archived in PCS setups for older shots (Sec. 2.11.5.3).

6. Add code to the `alg_parameters` function (Sec. 2.11.4.1) for the algorithm to create the necessary parameter data blocks and initialize each one with default data. Note that all of the parameter data blocks for the static data item are considered to contain "raw" data so that they are always initialized in the `alg_parameters` function. In order to create some of the parameter data blocks, it may be necessary to define a C language structure that will be contained in a block and its associated descriptor (Sec. 2.11.4.5). When creating the parameter data blocks, be certain to specify the correct options (Sec. 2.11.4.4) for the block. Particularly relevant to a static data item is the option indicating that the data in the block should be archived with the PCS setup. Usually raw data parameter blocks should be archived. Also, any blocks associated with the static data item must be made available to the user interface.

7. Decide the name of the IDL procedure that will implement the widgets for editing the static data. Put this name in the data item descriptor (Sec. 4.7).

8. Write the IDL code to implement the data item editor (Sec. 2.11.8). Add the file containing the code to `compileparamidl.pro` and `makeinstall` so that the code will be included when the PCS is built.

Whether a standard or a custom static data item is used, the data item defines only raw parameter data blocks. If it is necessary to derive processed data from the raw data, the necessary code must be added to the `alg_vectors` function and the data entry numbers of the processed data must be added to the data item descriptor. See Sec. 2.10.2 for the starting point to understand processed data.

2.10.2 Processed data

Usually, the data provided to the real time CPU are "processed" data that are derived from the "raw" data provided by the operator. (The exception is that a parameter data block containing raw data can be transferred directly to the real time CPU.) The algorithm author specifies how the raw data are used to produce the processed data.

For instance, the time evolution of a continuous target vector element is typically determined by using a waveform or combination of waveforms as input. In the simplest case, a waveform is scaled and offset and the resulting value becomes the target vector element value. In this example, the units familiar to the PCS operator are converted to units useful to the control system algorithm. For example, algorithms should be designed so that numbers used by the control system are near 1 to avoid possible overflow or underflow problems.

So, instead of working in amps as programmed by the operator, the control algorithm might scale the operator's input to MA if a value is typically 1×10^6 amps.

In the most general case, any arbitrary combination of the raw data, composed of waveforms and static data, can be used to compute the processed data provided to the real time CPU.

The values of the processed data are updated as the raw input data changes so that the calculations do not all need to be done during the setup that is performed immediately prior to a shot. This avoids delays in getting ready for a shot. So, when a piece of raw data changes, the processed data that is calculated from that raw data is updated immediately.

The algorithm author specifies what processed data must be recomputed when a given piece of raw data changes. Each piece of processed data is identified by a unique "data entry number." A data entry number (Sec. 2.10.4) generally specifies a unique location for storage of data on one of the real time CPUs. For each piece of raw data there is a list (see `targets` in Sec. 4.7) of data entry numbers of processed data that must be changed when that piece of raw data changes.

For each data entry number, there is a piece of code that computes the associated piece of processed data. The algorithm author provides this code, usually by using one of a set of standard routines provided by the PCS infrastructure.

The routine that calculates the processed data is called the "vectors" routine (Sec. 2.10.5). There is one of these routines provided for each algorithm.

2.10.3 Generating vector indices

In many places in an `alg_master.h` file, the author will need to refer to a particular element of a given vector or a particular parameter data block. This is because in the real time code most of the data input and output is through the standard vectors and parameter data blocks.

The algorithm author, by convention, puts in the first section of the `alg_master.h` file, the `CONTROLDEF` section (Sec. 4.3), a series of definitions of macros that specify vector element and parameter data block numbers. These definitions have the following format.

```
\ntt{XC\_NAME\_ANOTHERNAME}
```

where `X` is the character string indicating one of the data structures (a vector or parameter data block), `C` is a letter identifying the virtual CPU on which the vector element or parameter data block is located (e.g. A, B, C...), and `NAME` is the abbreviation for your algorithm (you can choose a value for `NAME` that strikes your fancy). By convention, `NAME` is the same as the algorithm "identifier" with all characters converted to capital letters. `ANOTHERNAME` is an arbitrary name for the data structure element. `ANOTHERNAME` generally has some special meaning in the context of the algorithm code.

For instance: `TA_DNULL_ZXP` indicates an element of the continuous portion of the target vector on virtual CPU A used by the algorithm with the identifier `dnull`. Similarly, `EA_DNULL_DFR` is an element of the error vector and `SA_DNULL_SIV` is an element of the shape vector.

The characters used by convention for `X` are the following.

- The continuous portion of the target vector: `T`.
- Floating point step portion of the target vector: `FST`.
- Integer step portion of the target vector: `IST`.
- Pointer target vector: `PTR`.
- Error vector: `E`. There is a one-to-one correspondence between elements of the P vector and the error vector so the elements of the P vector are referenced using the same macros as for the error vector.
- Shape vector: `S`.
- Command vectors: `CMD`. There is a one-to-one correspondence between elements of the floating point command vector and the integer command vector so the elements of both vectors are labeled with these characters.
- Communication vector: `C`.
- A parameter data block order index: `B`.

In each vector, each category is assigned one or more blocks of space that the category can use. The algorithm author uses a predefined macro to define a vector element index in one of these blocks. For instance,

`#define TA_DNULL_ZXP CTA_SHAPE(3)`

assigns element 3 of the block in the continuous portion of the target vector (indicated by `CT`) assigned to the shape category to the symbol `TA_DNULL_ZXP`. `TA_DNULL_ZXP` can then be used as an index into the target vector array. For example,

```
    x = targets[TA_DNULL_ZXP - 1];
```

Note that by convention, the vector element numbers are 1 based (rather than 0 based) offsets so that in C code 1 must be subtracted to obtain the correct array index.

The assignment of space in the various vectors to the various categories is done in the master file for each real time cpu (e.g. `cpu1_master.h`). At the top of this file is a detailed description of the allocation of space in the vectors on that cpu. Following the comments giving this description is the code that defines the layout of the space in the vector. This code includes definitions of the various macros that must be used to define vector element numbers (look for `SUBSECTION: vector element number macros`).

The vector element number macros have the following name format:

`\ntt{VectorCpu\_CategoryBlock(block\_element\_number)}`

Here:

- `Vector` is an abbreviation for a specific portion of a particular vector.
  - `CT`: continuous portion of the target vector
  - `IST`: integer step portion of the target vector
  - `FST`: float step portion of the target vector
  - `PTR`: pointer target vector.
  - `PIDGAIN`: portion of the continuous portion of the target vector assigned to hold the gain values for the PID calculation
  - `SAT`: portion of the float step portion of the target vector assigned to hold the saturation levels for the "standard outputs".
  - `E`: error vector (note: the pidtau vector element numbers are the same as error vector element numbers so the error vector element number macro is used when referring to the pidtau vector. Similarly for the P vector).
  - `S`: shape vector
  - `CMD`: command vector
  - `FCN`: function vector
  - `COM`: communication vector.
- `Cpu` is the letter designating a particular virtual cpu for the given category (A, B, C . . .)
- `Category`: the name of the category (e.g. `SHAPE`, `IP`, `FSUPPLY`, `GAS`, `DENSITY`, `JPROFILE`, `RF`, `BEAM`)
- `Block`: the number of a particular block if the category is assigned more than one block in the given "Vector" portion.
- `block_element_number`: the index (1 based) in the particular subblock of the vector assigned to the specified category.

2.10.4 Generating data entry numbers

There is a standard series of macros that generate data entry numbers. The data entry numbers specify one of the following:

- An element of the target vector on a particular virtual cpu of a specific category.

- An element of the pointer target vector on a particular virtual cpu of a specific category.
- An element of the pidtau vector on a particular virtual cpu of a specific category.
- A parameter data block.

Data entry number macro names have the format:

```
\ntt{DeCpu\_Category(vector element number)}
```

Here,

- `De`: the data entry type
  - `T`: target vector
  - `POINTER`: pointer target vector
  - `PIDTAU`: pidtau vector
  - `P`: parameter data
- `Cpu`: the letter designating the virtual cpu of the specified category. (A, B, C . . .)
- `Category`: the category
- `vector element number`: the macro for the vector element as discussed above.

For example, a data entry number might be given by:

```
\ntt{TA\_SHAPE(TA\_DNULL\_ZXP)}
```

which generates the data entry number of the element of the target vector given by `TA_DNULL_ZXP` on the cpu A of the shape category.

2.10.5 The alg_vectors routine

As discussed previously (Secs. 2.10.1, 2.10.2), the data held by the waveform server is of two types:

1. Raw data provided by the PCS user (waveforms, static data).
2. Processed data to be provided to the real time cpu (target vector elements, parameter data blocks, pid lookup tables etc.)

The processed data is produced from the raw data by computations specified by the algorithm author. The routine that controls these computations is the `alg_vectors` routine. (Here `alg` refers to the identifier for the algorithm and `_vectors` is the conventional suffix for the function name).

Each piece of raw data has associated with it a list (see `targets` in Sec. 4.7) of "data entry numbers" (Sec. 2.10.4) that indicates which pieces of processed data must be recomputed when that piece of raw data is changed. Each piece of processed data is identified by a unique data entry number. When the raw data are changed the `alg_vectors` routine is called once for each piece of processed data that must be recomputed. The `alg_vectors` routine is passed the data entry number of the piece of processed data. The `alg_vectors` routine then uses a `switch` statement to determine which piece of code to execute in order to recompute the processed data specified by the data entry number.

There is also a list (Sec. 4.6.1) of data entry numbers that are not associated with a particular piece of raw data. These represent processed data that is computed just once when the shot phase is initialized. The `alg_vectors` routine must provide code for this processed data also.

The calling format for the `alg_vectors` function is

```
\ntt{void alg\_vectors(int data\_entry, struct shotphase *phase)}
```

The arguments are:

- `data_entry`: the data entry number of the piece of processed data to be computed.
- `phase`: pointer to the descriptor for the shot phase for which the processed data is to be computed.

Note that the `alg_vectors` function computes processed data for the specified shot phase using as input the raw data specified for that shot phase. So, when a waveform or static data is used as raw input data, this data normally comes only from the specified shot phase. The exception to this is that a list of external data items (Sec. 2.10.6) can be specified for an algorithm. These external data items are raw data located in another shot phase.

In general, a piece of processed data is computed, with one or more pieces of raw data as input, using one of a standard set of utility routines. For each of these standard routines there is a C preprocessor macro defined for calling the routine. The macro includes the code to handle the `case` statement for the switch structure and the macro generates the correct code to call the function to do the computation.

The utility routines that are available are documented in Sec. 14. An example is `case_wvmp`. This macro generates the code that creates a target vector element from a waveform by multiplying the waveform vertices by a scale factor and then adding an offset. An example of the use of this macro is the following code segment (taken from the DIII-D algorithm master file `snull_master.h`).

```
                              /* waveform = cm */
case_wvmp(TA_SHAPE(TA_ANLGSTD_RPP),TA_ANLGSTD_RPP,
          CPUA,"RPP",-16.234f,0.1f)
```

In this example, the element of the target vector that is being defined is given by the macro `TA_ANLGSTD_RPP`. The data entry number that corresponds to this target vector element is `TA_SHAPE(TA_ANLGSTD_RPP)`. The data for the specified target vector element are generated by taking the waveform `RPP`, multiplying the waveform vertices by 0.1 and then adding -16.234 to the result. The result is a list of vertices that define the time evolution of the specified target vector element.

In this example the waveform is used by the operator to specify a value in units of cm. The numerical values given by the operator are typically near 160 cm, but the code executed in real time needs values that are on a different scale; the values are near zero and are in units of 10 cm. So, the conversion of the vertices from the waveform to the target vector takes into account the conversion from units that are convenient for the operator to units that are convenient for the real time code.

Many other examples of how to use macros in the `alg_vectors` can be found in the various `alg_master.h` files.

An outline of an `alg_vectors` routine is shown below. As shown by the example, the routine is quite simple. It consists only of a single `switch` statement, its associated `case` statements and the code that goes with each `case` statement. If one of the standard macros is used, the `case` statement and the code for that `case` is generated by the macro. The algorithm author also has the option of providing the `case` statement and its associated code. This is used primarily when there is no standard utility routine to use, such as the computation of processed parameter data.

```
void alg_vectors(int data_entry, struct shotphase *phase)
{
   switch(data_entry)
   {
      case_some_utility_routine(
                 /* the arguments for this macro */
                              )
      /* the macro generates all necessary code */

      case SOME_DATA_ENTRY_NUMBER:
         /* code to compute this processed data */
         break;
   }
   return;
}
```

When writing the `alg_vectors` function it is important to keep in mind how the function is called and the difference between raw and processed data. On each call to the `alg_vectors` function, the processed data labeled by a particular data entry number is computed. This processed data can be computed using as input any other data available in the waveform server, including other processed data. However, there is no guaranteed order in which the `alg_vectors` function will be called for the various data entry numbers. So, if any processed data is required during a call to the `alg_vectors` function, it is necessary to obtain that processed data by recomputing it.

2.10.6 External data items

For the most part, processed data (Sec. 2.10.2) are produced from raw data that are contained in the same shot phase (and thus the same control algorithm) that contains the processed data. However, sometimes it may be necessary to use raw data from another phase (which could be in a different category) when producing processed data. An example would be data that specifies some information about the PCS or tokamak setup that could be common to all algorithms. These data might be held in a category/phase combination that is specifically intended to hold this type of general purpose data (e.g. the "data acquisition" or "system" categories).

Raw data items that are required to produce processed data but which are contained in another phase are called "external data items."

For each algorithm there can be a list of descriptors of external data items that are used by the algorithm. Each descriptor specifies the location of the raw data item and provides a list of data entry numbers of processed data which must be recomputed when the external raw data item is changed. The way to set up the list of external data item descriptors in an algorithm master file is described in Sec. 4.8.

The location of an external data item is specified by providing four strings in the descriptor: the data item name, the category and algorithm identifiers and the phase name. Any of these strings can contain one or more asterisks as a multicharacter wildcard and one or more question marks as a single character wildcard. If one of these strings is irrelevant (like the phase name), the string can contain only an asterisk so that anything will match it.

When the operator changes the data for the external data item, the `_vectors` function is called for each data entry number in the `targets` list in the external data item descriptor. Providing this list of data entry numbers is the purpose for the list of descriptors. It is the job of the code in the `_vectors` function to obtain the necessary external data and use it to produce processed data.

The code in the `_vectors` function that uses the external data calls a function to obtain the data. Normally, external data would come from a parameter data block in another phase. Access to a parameter data block in another phase is obtained by calling the function `get_external_param_block_ptr` (described in Sec. 14.3.17). The same category and algorithm identifiers and phase name that are in the external data item descriptor are provided as arguments to this function. In addition, the parameter data block name is provided.

Conceivably, external data could also be the vertices on a waveform. However, because the time associated with each waveform vertex is referenced to the start of the phase, it isn't clear how waveform vertices from one phase could be useful in calculating processed data for a different phase. The method to use to access the vertices from a waveform is described in Sec. 2.10.8.

2.10.7 Storage of processed data in the waveform server

Processed data is of two types.

1. Parameter data blocks containing data that were computed from the raw data provided by the operator.

2. Data that specifies the time evolution of the value in a particular element of the target vector, pointer target vector or pidtau vector.

This section provides background about the storage of these data in the waveform server that might be required by an algorithm author.

2.10.7.1 Parameter data storage

Processed parameter data blocks and raw parameter data blocks are stored together in one block of memory for each phase. The location of this memory block is stored in the phase descriptor at `phase`. (The pointer to the phase descriptor is an argument to the `alg_vectors` function, Sec. 2.10.5). The organization of the memory block used for storage of the parameter data is maintained by the infrastructure. The algorithm author uses a set of standard functions to access the data in a particular block (see Sec. 2.11.4 and 14.3).

To aid the algorithm author in diagnosing problems with parameter data there are several options under the `diagnostics` menu on the user interface.

- `phase parameter data summary`: this menu item brings up a window that contains a list of every parameter data block contained in the phase presently selected by the user interface. Included in the list are the block names, the flags assigned to each block and the size of the data storage area in the block. This display is useful for checking the options configured for each parameter data block.

- `phase parameter data (IO queue)`: this menu item brings up a window containing a list of just the names of the parameter data blocks. One of the blocks can be selected by clicking on its name and then when the `show` button is clicked a new window displays the values in the parameter data block. The data are interpreted using the data object descriptor for the parameter data block. The descriptor contains only information about the data types and quantity so the data values are visible but not much information is available about the data organization from this window.

The parameter data blocks displayed by this menu choice are those stored in the portion of the waveform server that contains the most recent data provided by the user interface. So, this list contains only raw parameter data blocks.

- `phase parameter data (work queue)`: this menu item is the same as `phase parameter data (IO queue)` except that the data come from the portion of the waveform server that is responsible for calling the `alg_vectors` function to compute processed data. So, both raw and processed data parameter blocks are available with this menu item.

2.10.7.2 Vector element time evolution storage

The time evolution of a data vector element is stored in a "change list." Each entry in a change list specifies that at a particular time during the discharge the data in a particular vector element should change to a new value. The change list entry contains the information required to obtain the new value for the vector element.

For instance, for a integer step target vector element, the change list entry contains the new value to be written to the target vector element. For a continuous target vector element, the change list contains the information about a new vertex: the Y axis intercept and slope of the line that defines the time evolution of the value in the continuous target vector element.

Code in the `alg_vectors` function that computes values for vector elements takes the vertices from one or more waveforms and any other required data and computes entries in the change list.

The PCS user interface `diagnostics` menu entry `phase change list` allows the algorithm author to view the content of the change list for the shot phase presently selected by the user interface as it is stored in the waveform server. By viewing the change list the algorithm author can determine if the algorithm code is computing the values for the vector elements correctly. The `phase change list` window displays the change list entries sorted by time (with respect to the start of the shot phase), vector element type and vector element number. The data stored for each change list entry is also shown.

Note that the change list contains only the number of the element in the vector that is to be changed. In order to make the display in the `phase change list` window more useful for diagnostics, the information in the algorithm's vector element information arrays (Section 4.5) is used to label each of the change list entries in the display. This is one of the primary purposes of providing this vector element information (along with providing information for archival of the vector element content) and is the reason that an entry in the vector element information array should be provided even if the vector element will not be archived.

2.10.8 Producing target vector data from waveform vertices

There are several standard macros described in Sec. 14.1 that are useful for generating the data for a target vector element from the vertices on one or more waveforms. An example is the macro `case_wvmp` that generates a function call to create the content of a continuous target vector element from a waveform by scaling and offsetting the waveform vertices.

If a standard function is not available to generate target vector element data in the desired way, it is necessary to write a customized function. The way to implement this function is described in this section.

There are four basic steps required in this customized function.

1. Access the raw data required. Since the function we are writing generates data for a target vector element, normally the vector element value will evolve with time during a discharge. So, we will need to be able to access the vertices from a waveform since a waveform is how the operator provides raw data to specify a time evolution.

2. If more than one waveform is to be combined, create a set of vertices for each waveform that is on a common timebase.

3. Compute the data describing the time evolution of the target vector element value. This consists of an an array of time/value pairs.

4. Generate the change list entries describing the time evolution of the target vector element.

The standard macros (Sec. 14.1) provide good examples of what is required. These macros are all defined in the infrastructure file `serverdefs.h`. By referring to the macro definition the function called can be determined and then located in the infrastructure file `targetvectors.c`.

For instance, the macro `case_wv2mp` generates code to call the function `wv2mp` (Sec. 14.2.10). This function contains a good, simple example of the four steps listed above. The steps in this function are followed in the description here. The utility functions discussed here are described in detail in Sec. 14.2.

The function `wv2mp` takes the product of two waveforms and scales and offsets it to create the time evolution of a target vector element value. So, the call to the function identifies the two waveforms by their names and provides the offset and scale factor values. Also specified are the indices (with respect to 1) of the target vector element and the virtual CPU on which the target vector is located. The phase descriptor specifies the shot phase for which the target vector data are computed and `target_type` specifies the type of target vector element for which data is being computed, either a continuous target vector element or a step target vector element.

```
void wv2mp(char *waveform1, char *waveform2,
           float offset, float factor,
           int tindex, int cpu_index, struct shotphase *phase,
           int target_type)
{
   int i,windex1,windex2,
       wpindex1, wpindex2,
       xcount,numvertex1,numvertex2;
   struct vertex *vertices,*vertices1,*vertices2;
   float *xvalues,*yvalues1,*yvalues2;
```

It is necessary, next, to obtain copies of the vertices defined for the two waveforms. First, though, the waveform names must be translated into indices into the lists of waveforms using the function `find_waveform_indices` (Sec. 14.2.3). In the waveform server there is one long array of descriptors of all waveforms used by all algorithms. This function returns an index into this array (`windex`). This array of waveform descriptors contains only data about the waveform that is fixed by the algorithm source code. For each shot phase, there is also a list of information about each of the waveforms used by the algorithm assigned to that phase. This information contains the data that varies depending on the shot setup. The function `find_waveform_indices` also returns an index into this phase specific list (`wpindex`).

```
   find_waveform_indices(waveform1,phase,
                         "waveserver (wv2mp)",
                          &wpindex1, &windex1);
   find_waveform_indices(waveform2,phase,
                         "waveserver (wv2mp)",
                          &wpindex2, &windex2);
```

The number of vertices assigned to the waveform is given by `phase->wavelist[wpindex].numvertex`. This value should be checked first to make certain that the operator didn't accidentally remove all vertices from the waveform. If either waveform has no vertices, nothing further can be done. An entry will automatically be added to the raw data problem list (Sec. 2.14.1) to force the operator to add vertices to the waveform.

```
   if(phase->wavelist[wpindex1].numvertex == 0)
      return;
   if(phase->wavelist[wpindex2].numvertex == 0)
      return;
```

Then, a copy of the vertices for the waveform is obtained using the function `fetch_vertices` (Sec. 14.2.1). This function returns a pointer to an array of structures of type `vertex`. This memory must be unallocated before exiting the function (call `free()`).

```
   vertices1 = fetch_vertices(&waveforms[windex1],
                              &phase->wavelist[wpindex1],
                              &numvertex1,1);
   vertices2 = fetch_vertices(&waveforms[windex2],
                              &phase->wavelist[wpindex2],
                              &numvertex2,1);
```

The structure returned is defined as follows.

```
struct vertex {float x; float y;}
```

If the waveform is of the “continuous” type, the array of vertices returned exactly duplicates the vertices for the waveform. If the waveform is of the “step” type, `fetch_vertices` will optionally return an extra vertex at a time one phase clock tick before each waveform vertex. This makes an array of vertices that simulates the vertices for a continuous type waveform. This is required if continuous and step type waveforms are combined to create the target vector element values.

At this point, because multiple waveforms will be combined (in this example), the waveform vertices must be interpolated onto a common time axis. That is, at any time value where either waveform has a vertex, the correct Y axis value must be determined for both waveforms.

So, first a single list of all X axis values for both waveforms is created.

```
   xvalues = (float *)malloc( (numvertex1 + numvertex2) * sizeof(float));
   for(i = 0; i < numvertex1; i++)
      xvalues[i] = vertices1[i].x;
   for(i = 0; i < numvertex2; i++)
      xvalues[i + numvertex1] = vertices2[i].x;
```

Next, the function `sort_xvalues` (Sec. 14.2.5) is used to create a sorted, common list of X axis values with duplicates removed.

```
   sort_xvalues(xvalues, numvertex1 + numvertex2, &xcount);
```

Finally, the function `wv` (Sec. 14.2.6) is called to evaluate each of the waveforms on the common X axis value list.

```
   yvalues1 = wv(vertices1,numvertex1,xvalues,xcount);
   yvalues2 = wv(vertices2,numvertex2,xvalues,xcount);
```

At this point the first `xcount` values in the array `xvalues` contains the common timebase values. The arrays `yvalues1` and `yvalues2` contain the Y axis values of each waveform that correspond to each X axis value on the common timebase.

Now whatever custom computation is necessary can be done to produce a single array of vertices. In this example, the Y values of the vertices are multiplied and the result is scaled and offset.

```
   vertices = (struct vertex *)malloc(xcount*sizeof(struct vertex));
   for(i = 0; i < xcount; i++){
      vertices[i].x = xvalues[i];
      vertices[i].y = (yvalues1[i] * yvalues2[i]) * factor + offset;
   }
```

Finally, the final array of vertices is copied into the phase change list in the appropriate format. If data for a continuous target vector element is being created, the function `wvcontinuous` (Sec. 14.2.7) is used to create change list entries. If data for a step target vector element is being created, the function `wvstep` (Sec. 14.2.8) is used.

```
   if(target_type == CT_CHANGE){
      if(wvcontinuous(vertices,xcount,tindex,cpu_index,phase) != 0){
 msg_log(PCSMSG,"Waveform server exiting in wv2mp.\n");
 pcs_abort();
      }
   }
   else
   {
      if( (waveforms[windex1].step == 1) &&
          (waveforms[windex2].step == 1) )
         wvstep(vertices,NULL,xcount,1,tindex,cpu_index,phase,target_type,0);
      else
         msg_log(PCSMSG,"Waveserver (wv2mp): PROGRAMMING ERROR!\n"
                        "A step target cannot be made"
                        " from the waveforms %s and %s.\n",
                        waveform1,waveform2);
         pcs_abort();
   }
```

Note that there are checks for error conditions. If there is a failure in the function `wvcontinuous` the function will have printed an error message. The function `pcs_abort` is used to cause a core dump of the waveform server process for debugging as this is a problem that must be corrected by the algorithm author.

Note also that it is possible to create a continuous target vector element from a step-type waveform by obtaining the extra vertices from `fetch_vertices` as discussed above. However, it is not possible to create a step target vector element from a continuous-type waveform. A check is made here for this type of problem which must be corrected by the algorithm author.

2.10.9 The alg_vertices routine

When an algorithm is chosen for a shot phase, all of the waveforms that are part of that algorithm are assigned a single vertex by default at $t = 0$ (with respect to the start of the shot phase) and Y value equal to 0. The algorithm author can override this default vertex by including code in an `alg_vertices` routine. This routine can specify a default set of vertices for any of the waveforms that are defined for that algorithm. Any waveforms not referenced in the `alg_vertices` routine retain their default vertex at $t = 0$.

An outline of the `alg_vertices` function is as follows.

```
void alg_vertices(struct shotphase *phase)
{
   struct vertex newv[3];
   int numvertex;

   newv[0].x = 0.0;
   newv[0].y = 0.0;
   newv[1].x = 1.1;
   newv[1].y = 22.5;
   newv[2].x = 5.4;
   newv[2].y = -34.5;
   numvertex = 3;
   set_vertices("mywaveform",phase,newv,numvertex);
}
```

In this example, the waveform with name `mywaveform` is assigned three vertices. This set of vertices replaces any vertices previously assigned to the waveform. The infrastructure function `set_vertices` (Sec. 14.6.5) is used to change the vertices for the waveform. The function is passed the name of the waveform, the pointer to the shot phase which uses the algorithm (which is an argument to the `alg_vertices` function), an array of structures that holds a list of the new vertices and an integer giving the count of the number of vertices.

2.10.10 Global and static variables in the waveform server

In Sec. 2.5.5 there is a description of the problems that can arise with the use of global and static variables within application-specific code on the real time processors. The principal problem is that the same application code is common to every phase using a given algorithm. So, phases using the same algorithm will also use the same global or static storage space, with each phase overwriting the data stored by other phases.

This problem exists in the waveform server as well. If an algorithm requires storage space that is specific to a particular phase, parameter data blocks (Sec. 2.11) should be used for this storage.

2.11 Parameter data

The parameter data facility is used to manage blocks of memory for storage of arbitrary data by the algorithm. These "parameter data blocks" are created by the algorithm's code in the waveform server. Any parameter data block can also be made available in the host_cpu process or on the real time processor. So, the algorithm uses the parameter data facility to position any arbitrary data in the location where it is needed.

The parameter data facility (see Sec. 2.5.3 for an introduction) is very flexible in that it will handle any type of data structure, but this flexibility means that the amount of generic code that can be written to handle it is reduced. Most of the code for handling parameter data must be written by the application programmer once the structure of the data is defined. (The exception is that there are support routines for certain generic types of data structures.) This section describes the methods for managing parameter data within the PCS.

2.11.1 Uses for parameter data

The parameter data facility is used to manage the data required by an algorithm and to copy the data to the processes where it is required. The algorithm code initially creates and organizes the parameter data in the waveform server process. Parameter data "blocks" are created, each of which is filled with the appropriate data, is given a "name" to identify it, and is assigned a set of flag values to indicate which one or more of the uses listed below applies to the particular block.

The primary uses for parameter data are the following.

1. Data that are held between shots in the address space of the waveform server (Secs. 2.11.3.1 and 2.11.4). This provides general purpose space for an algorithm to store data.

2. An associated group of parameter data blocks is used to create a static data item (Secs. 2.10.1.2 and 2.11.3.2). These blocks contain raw data (Sec. 2.10.1 and Sec. 2.11.2).

3. Data that are provided to the host_cpu process by the waveform server for use during the setup for a shot and during the cleanup after a shot. See Sec. 2.11.3.3.

4. Data that are provided to the real time CPU by the waveform server for use by the real time code. See Sec. 2.11.3.3.

5. Scratch storage space that is provided for use by the control algorithm on the real time processor (Sec. 2.11.10). Scratch parameter data blocks are a special form of parameter data block. Prior to the shot all scratch parameter data blocks are initialized with zeros.

The PCS infrastructure code will automatically transport selected parameter data blocks to the host_cpu process and the real time processor during setup for a shot.

2.11.2 Raw and processed parameter data

The parameter data facility provides a generic method to manage blocks of data. The role of a given parameter data block in the PCS control algorithm can be one or more of the choices listed in the previous section (2.11.1). The methods for creating and accessing a parameter data block are the same independent of the role of that data block. So, some care by the algorithm author is required so that the the exact role of a given parameter data block can be easily understood.

The features that identify the role of a parameter data block are the block name (Sec. 2.11.4.3), the options for the block (Sec. 2.11.4.4), and the virtual cpu(s), if any, where the block will end up. For instance, a block name that begins with the name of a static data item (Sec. 2.10.1.2) should always be part of an associated group of blocks that makes up the static data item. Some of the options indicate whether the parameter data block is a scratch block, whether the block will be archived with the PCS setup, or whether the block can be overwritten during a restore of a PCS setup. A block with no options is one that is simply used to store data within the context of the waveform server. The virtual cpu(s) specify that the block will be copied to a real time processor or a `host_cpu` process.

Parameter data blocks can contain either raw or processed data. Raw data are the data provided by the PCS operator. Raw data blocks are almost always part of a static data item. The algorithm author can provide an “editor” within the user interface to allow the PCS operator to specify values to be stored in the raw parameter data blocks.

Processed data are derived from the raw data from a waveform or a parameter data block that is part of a static data item. Processed parameter data are created by the `alg_vectors` function (Sec. 2.10.5) and are stored in parameter data blocks with arbitrary names.

2.11.3 Quick start outlines for parameter data

A list of potential uses for parameter data blocks is given in Sec. 2.11.1. This section provides a short outline of the steps required when parameter data blocks are used in each of these ways. The code described here would normally be called from either the `alg_parameters` function (Sec. 2.11.4.1) or the `alg_vectors` function (Sec. 2.10.5).

2.11.3.1 Outline for parameter data block usage 1

The simplest application of a parameter data block is for storage of arbitrary data within the waveform server. In this application, a single parameter data block is created and loaded with the desired data.

- First, create the data to be stored in the parameter data block. Often, the data are in memory obtained with `malloc`. This memory can be freed after the data are placed in the parameter data block.

- Next, create the descriptor for the parameter data block (Sec. 2.11.4.5). The descriptor is usually placed in the `CONTROLDEF` section of the algorithm master file (Sec. 4.3). Here is an example corresponding to the example block creation code below.

```
#define GEN_shapestr \
SDSTART(struct shapestr {              ,D_shapestr                   )\
DGEN    (  int   time;                 ,INTTYPE,1                    )\
DGEN    (  float zp;                   ,FLOATTYPE,1                  )\
DGEN    (  int   ix[2];                ,INTTYPE,2                    )\
DGEN    (  float r[MAX_BOUNDARY_POINTS]; ,FLOATTYPE,MAX_BOUNDARY_POINTS  )\
DGEN    (  int npt;                    ,INTTYPE,1                    )\
SDEND   (                };                                          )
GEN_shapestr
```

  Here, `GEN_shapestr` generates the C language code that declares the structure of type `shapestr`.

- Choose the name for the parameter data block (Sec. 2.11.4.3). Because the block is not part of a static data item, any name can be used.

- Store the data in the parameter data block using one of the functions described in Sec. 2.11.4.2. Here is an example.

```
    {
GEN_shapestr

D_shapestr[0].size = num_target_shapes;
        putblock(phase, "target shapes data",
                 NULL,   /* no options */
                 NO_BLOCK_ORDER,
                 D_shapestr,
                 (char *)target_shapes);
    }
```

  In this example, the parameter data block contains an array of structures of type `shapestr`. The number of elements in the array is `num_target_shapes`. The descriptor is generated by the macro `GEN_shapestr`. Because the parameter block is simply used to store data within the waveform server, the options argument (Sec. 2.11.4.4) is NULL and the block order index (`NO_BLOCK_ORDER`) (Sec. 2.11.4.7) is not relevant.

- The data are always copied into the parameter data block from the original location. So, any memory that was allocated to hold the data can be freed. For instance, in the example above, `free(target_shapes)` would be called if necessary.

2.11.3.2 Outline for parameter data block usage 2

When it is necessary to allow the PCS operator to edit a set of static data values a static data item is required. A static data item is similar to a waveform in that it is used to provide raw data to the PCS. However, the format of the data is not as strictly defined as the data for a waveform. The algorithm author has wide flexibility to design a set of parameter data blocks to implement a static data item.

An outline of the procedure for creating a static data item is given in Sec. 2.10.1.2.

2.11.3.3 Outline for parameter data block usages 3 and 4

Any given parameter data block can be present in the host_cpu process or on the real time processor or both. It is only necessary to specify the correct virtual cpu letters in the options argument (see Sec. 2.11.4.4).

For instance, to have the parameter data block copied to the host_cpu process for virtual CPU A of the category to which the algorithm is assigned, specify the following option: "HOSTCPUS=A". To have the block available on the real time processor, virtual CPU A, specify the following option: "RTCPUS=A". In addition, a positive block order index must be given. To copy the block to both the real time processor and the host_cpu process, specify both options: "HOSTCPUS=A—RTCPUS=A". If the algorithm uses multiple virtual CPUs, then specify multiple letters: "RTCPUS=AC".

A parameter data block that is copied to the host_cpu process or the real time processor can contain either raw or processed data. A block containing raw data would be a component of a static data item (Sec. 2.10.1.2). Any parameter data block containing processed data can be copied to the host_cpu process or the real time processor. Simply specify the appropriate virtual cpu letter(s) on the appropiate keyword of the options argument, as discussed above, when the function that creates the parameter data block (Sec. 2.11.4.2) is called. Recall that processed parameter data blocks are created by code in the `alg_vectors` function (Sec. 2.10.5).

2.11.4 Parameter data in the waveform server

The waveform server is the central storage location for the parameter data (along with all of the other PCS setup data). Each feedback control algorithm has a (possibly unique) definition of the structure of its set of parameter data blocks.

Each shot phase that has assigned to it an algorithm that uses parameter data has its own set of parameter data blocks. Thus, if the shot phase in a given category is changed during a shot, even if the algorithm does not change, the set of parameter data that is in use for that category changes. The structure of the parameter data is defined by the algorithm that is assigned to the shot phase.

However, a parameter data block can be created that is shared by other phases by specifying the SHARED keyword in the options argument of the function that creates the block (see Sec. 2.11.4.8).

2.11.4.1 Initializing parameter data

The algorithm author must provide a parameter data initialization routine that executes in the waveform server process. This routine is called when a shot phase is created or a new algorithm is assigned to a shot phase. The routine creates and initializes the set of "raw" parameter data blocks for that shot phase.

The raw parameter data blocks normally correspond to those containing data that are edited using the user interface. These raw data are used by the `alg_vectors` function to create the "processed" data that are passed to the real time processor. After the parameter data initialization function is called, the infrastructure code in the waveform server calls the `alg_vectors` function to create the processed data. Normally the raw data blocks are created only in the parameter data initialization function and the processed data blocks are created only in the `alg_vectors` function.

The initialization routine calling format is:

```
\ntt{void alg\_parameters(struct shotphase *phase)}
```

The naming convention for the routine is shown in this format; `alg` is the identifier for the algorithm. `phase` is a pointer to the descriptor of the shot phase being initialized.

An outline of the procedure for the parameter data initialization routine is as follows.

1. Before initializing the parameter data blocks for a particular static data item, call the function `check_access_control` (Sec. 14.6.39) to be certain that the data item really should be initialized.

   ```
   if(check_access_control("my static data item",phase,AC_INITIALIZE))
   {
   /* Initialize the parameter data blocks for the data item. */
   }
   ```

   See Sec. 2.17 for more information.

2. Create the actual data for each block, perhaps by filling temporary memory allocated for the purpose with `malloc`.

3. Call one of the infrastructure support routines (described in Sec. 2.11.4.2 and, in detail, in Sec. 14.3) to create each of the parameter data blocks.

4. If necessary, free the temporary memory previously allocated to hold the data for each block. The functions that create a parameter data block copy the data so the temporary memory is no longer needed.

2.11.4.2 Creating parameter data blocks

A general purpose parameter data block creation routine is provided, and various more specific routines are also available. Here is a brief list of the routines. See Sec. 14.3 for detailed descriptions of the routines.

- The general purpose parameter data block creation routine is `putblock` (Sec. 14.3.1). This routine simply stores away a block of bytes with arbitrary structure. The algorithm author must manage the organization of the data in the block. In addition, the author must provide a data object descriptor (Sec. 2.11.4.5) that specifies what types of data are in the block for use in archiving the data and passing it to other processes.

- The following routines create a block containing an array of a single data type. These routines simplify somewhat the creation of the parameter data block by creating the data object descriptor.

  - `putblock_chars` (Sec. 14.3.2) creates a block containing an array of data of type `char`.
  - `putblock_shorts` (Sec. 14.3.3) creates a block containing an array of data of type `short`.
  - `putblock_ints` (Sec. 14.3.4) creates a block containing an array of data of type `int`.
  - `putblock_longs` (Sec. 14.3.5) creates a block containing an array of data of type `long`.
  - `putblock_floats` (Sec. 14.3.6) creates a block containing an array of data of type `float`.
  - `putblock_doubles` (Sec. 14.3.7) creates a block containing an array of data of type `double`.
  - `putblock_strings` (Sec. 14.3.8) creates a block containing an array of data of type `char`.

- The following routines create multiple blocks which allow future expansion of the data. One block contains the data, another contains labels, and yet another contains substitution values used as default values when the block has expanded. These routines are used with standard static data types that are recognized by the generic editor. See (Sec. 2.12).

  - `putblock_labeled_int_array` (Sec. 14.3.9) creates blocks associated with a `labeled int array`.
  - `putblock_labeled_float_array` (Sec. 14.3.10) creates blocks associated with a `labeled float array`.

  - `putblock_labeled_structure` (Sec. 14.3.11) creates blocks associated with a `labeled structure`.

  Note that the names of these three blocks each have a suffix appended to the name given in the function call. The labels block has the suffix " :labels", the data block has the suffix " :data", and the substitute data block has a suffix of " :substitute data". Remember that when retrieving the `data` block in a waveform server function like the "_vectors" function, the suffix " :data" is required.

- The following routine creates multiple blocks associated with matrices. This is the standard assumed by the matrix editor. Rows and columns can be added to the matrices in the future.

  - `putblock_labeled_matrices` (Sec. 14.3.12) creates blocks associated with `labeled matrices`.

  This function creates five or six blocks each with a suffix appended to the name given in the function call. These suffixes are " data" for the floating point data, " dimensions" which is a four element int array (see below), " row names", " col names", and " names" for the names of the matrices. The last block with a suffix of " substitute data" is optionally created only if substitute data are provided in the function call. Note that there is no colon added to the suffixes.

  The dimensions block contains the following:

  - the number of matrices
  - the number of matrix rows
  - the number of matrix columns
  - the maximum length of the row names, column names, and matrix names

When a parameter data block is created several values are specified.

- A character string is used to give the data block a name for later identification. See Sec. 2.11.4.3 for more details.

- An options string is provided that specifies the uses for that particular parameter block. For instance, whether the block should be copied to the real time processor, whether the data should be archived and/or restored with the PCS setup, etc. These parameter data block options are described in Sec. 2.11.4.4.

- A "block order" index is provided for parameter data blocks that contain data that will be copied to the real time processor. This index specifies the order of storage on the real time processor and is also used as an identifier for locating the block on the real time processor. (See Sec. 2.11.4.7.)

- For the most general parameter block creation function, `putblock`, a data object descriptor (Sec. 2.11.4.5) must be provided to specify the structure of the data in the block.
- A pointer to the data to be copied into the parameter data block is required, unless a scratch parameter data block (Sec. 2.11.10) is being created in which case a null pointer is passed instead.

2.11.4.3 The parameter data block name

Each parameter data block is given a character string name that is used to identify the block in all cases except on the real time processor. In principle, the block name can be any string, but for parameter data blocks that might be transferred to the user interface or which will be archived and/or restored with the PCS setup, the naming convention described in the remainder of this section must be followed.

To create a "static data item" (Sec. 2.10.1.2) composed of a group of "associated" parameter data blocks, a specific naming convention is required. This naming convention is the mechanism to make a group of blocks "associated."

For instance, a static data item containing an array of matrices might use one block to hold the matrix data, another for the matrix column names, another for the matrix row names etc. These parameter data blocks become associated by using the same prefix for the name of each block. For example, "`matrix A :data`", "`matrix A :row names`", "`matrix A :col names`". These three block names would be identified as being associated because they all start with the string "`matrix A`". Note that by convention the prefix and suffix of the block name are separated by " :", but there is nothing in the infrastructure that requires this.

For a static data item, the prefix of the parameter data block name must be the same as the data item name (Sec. 2.10.1.2).

For the uses of associated groups of parameter data blocks see Sec. 2.10.1.2, Sec. 2.11.5.5 and Sec. 2.11.5.2.

2.11.4.4 Parameter data block options

All parameter data block creation routines (Sec. 2.11.4.2) take a string argument that provides options that define how the parameter data block is used. The options argument is a string which gives keywords. Some keywords require a value while others do not. To specify a value on a keyword, simply use an equals sign:
”keyword=value”

Separate multiple keywords with a vertical bar:
”keyword1—keyword2=value—keyword3”

If the options argument is NULL, then the block is simply temporary storage within the waveform server process. It is not archived and will not be copied to the real time processor.

Options allowed:

- `USER`: When parameter data are requested by the user interface, this block should be provided. If this option is not set, this is an indication that the data in the parameter data block are not available to be edited by the user.
- `ARCHIVE`: Indicate that the parameter data block should be included in the PCS setup that is archived for each shot.
- `RTCPUS=<one or more virtual cpu letters>`: Each letter given indicates that the parameter data block should be copied to one of the real time CPUs. The letters A, B, etc. specify the virtual CPU numbers for the category, e.g., "RTCPUS=AC" specifies virtual CPU A and virtual CPU C. The keyword `ALL` can be used to specify all virtual CPUs used by the category.
- `HOSTCPUS=<one or more virtual cpu letters>`: Each letter given indicates that the parameter data block should be copied to one of the host_cpu processes. The letters A, B, etc. specify the virtual CPU numbers for the category, e.g., "RTCPUS=AC" specifies virtual CPU A and virtual CPU C. The keyword `ALL` can be used to specify all virtual CPUs used by the category.
- `NORESTORE`: Indicates that even if a parameter data block with the same name as the block being created exists in an archived PCS setup, the block should not be restored when the archived setup is restored. This option controls the restore of individual parameter data blocks. So it can be used to control a parameter data block that is part of a static data item. Even if the remainder of the parameter data blocks belonging to the static data item are to be restored, this flag can prevent an individual block from ever being restored. See Sec. 2.11.5.2 for more information.
- `SCRATCH`: Indicates that the block is a scratch data block (Sec. 2.11.10). The block has no associated data. When the block is created on the real time processor, it is filled with zeros. The real time algorithm code can use the block for any scratch storage purpose. This option has no meaning for a parameter data block that isn't created on a real time processor. Because no data need to be provided to fill the block, the argument to the function that creates the parameter data block (Sec. 2.11.4.2) that usually points to the data is specified as a null pointer. See Sec. 2.11.10 for more information.
- `EXTERNALDATA`: Use this option to indicate that the data in the parameter data block should be stored in a "parameter data block external file." See Sec. 2.17.4 for a discussion of this feature.

- `ALIGNMENT=n`: Use this option to specify the alignment of the parameter data block on the real time processor. See Sec. 2.11.4.6 for a discussion of this feature.
- `REGION=n`: Use this option to specify the memory region where the parameter data block should be placed on the real time processor. See Sec. 13.2 for a discussion of this feature.
- `SHARED[=value]`: Use this option to indicate that the parameter data block should be shared between phases in the waveform server. This keyword can have three possible values:
  - `SYSTEM`: The block is shared system wide. Any phase in any category may share the block.
  - `CATEGORY`: The block is shared only in the category. Any phase in the category can share the block.
  - `ALGORITHM`: The block is shared only by the algorithm. Any phase with the same algorithm can share the block.

  If this options keyword has no value, then the default is `SYSTEM`. See Sec. 2.11.4.8 for a discussion of this feature.
- `NOOVERWRITE`: Use this option to indicate that a shared parameter data block should not be overwritten if it already exists. See Sec. 2.11.4.8 for a more details about this option.
- `NODELETE`: Use this option to indicate that a shared parameter data block should never be deleted. See Sec. 2.11.4.8 for a more details about this option.
- `RTNOSHARE`: Use this option to indicate that a shared parameter data block (which is shared in the waveform server) should NOT be shared on the real time processor. See Sec. 2.11.4.8 for a more details about this option.

2.11.4.5 The parameter data block descriptor

A parameter data block can contain mixtures of any type of data. As long as the data are used only within the waveform server process, the algorithm author would normally arrange for the compiler to handle all of the details of the layout of the data block. However, there are several situations in which the data will be transferred to another process or perhaps to another computer.

- When the data are archived with the PCS setup.
- When the data are copied to the host_cpu process or to the real time processor for use during a shot.

- When the data are transferred to the user interface.

In these cases it will be necessary to know the structure of the data block; i.e. how many of each data type and in what order. This allows the PCS infrastructure code to handle differences in data representation when the data are transferred to another processor architecture. Therefore, when creating a parameter data block, it is necessary to provide a description of the data in the block.

The structure of the parameter data block is defined for this purpose by a "data object descriptor." The content of the parameter data block is the "data object" that is detailed by the descriptor.

In this section a description is given of how to implement the C language code for a data object descriptor. This code both defines the data object to the compiler and provides the capability to generate a descriptor for the data object. By combining the definitions of both the data object and the descriptor in the same code, the definitions are easily maintainable.

Only the minimum information is given here, but if the examples provided here are emulated it is straightforward to write the necessary code. More detail on the implementation of data object descriptors in the PCS is given in Sec. A.

A data object descriptor is an array of type `STRUCT_DESCRIPTORS`. Data object descriptors are implemented by creating a macro that generates the descriptor for a particular data structure. Then it is only necessary to invoke this macro each time the descriptor is required. The macro should be defined in the `CONTROLDEF` section of the algorithm master file (see Sec. 4.3).

Here are several examples that illustrate the method to generate a data object descriptor. The first, and simplest, example illustrates most of the basic method.

1. The simplest example of a data object is an array of values of a single type. In this case, the code to generate the descriptor is as follows.

```
/*
Define the macro that will generate a data object descriptor.
This code goes in the CONTROLDEF section of the algorithm
master file.
*/
#define GEN_example_variable \
DSTART(                                            D_example_variable)\
DGEN  (          int example_variable[2];            ,INTTYPE, 2)\
DEND
/*
Optionally, invoke the macro to define the variable.
Note that this creates a global variable named example_variable.
In this example of a simple variable, this is not required
and is probably not desired.
```

```
*/
GEN_example_variable
```

Here, a definition is given for the macro `GEN_example_variable`. This macro is usually used in two situations.

- Immediately after the definition of the `GEN_example_variable` macro, the macro is invoked. This first use of the macro generates C language code to define the data object which is a one dimensional array of type `int` with 2 elements.

  ```
  int example_variable[2];
  ```

  This variable will be global in all three programs: the waveform server, the host_cpu process, and the real time process. For this simple case, this step is optional and may not be desired at all.

  When the macro is utilized within the `CONTROLDEF` section of the algorithm master file, its function is to create the variable. This behavior is determined by the definitions of the macros `DSTART`, `DGEN`, and `DEND`.

- In sections of the algorithm master file that include C language instructions, rather than only definitions, the macros `DSTART`, `DGEN`, and `DEND` are redefined. In this case they cause the code necessary to define a data object descriptor to be generated when the `GEN_example_variable` macro is invoked. In the context where `GEN_example_variable` is invoked, `D_example_variable` is defined as the data object descriptor and can be used as a function argument for any function requiring a pointer to a data object descriptor. The code generated by the macro looks something like this.

  ```
  STRUCT_DESCRIPTOR D_example_variable[] =
     {/* necessary initializers for the data object descriptor */};
  ```

  Here is an example of the use of the `GEN_` macro to generate a descriptor.

  ```
  void example_function_1()
  {
     GEN_example_variable
     .
     .
     .
     putblock(...,D_example_variable,...);
  }
  ```

The important features of the code that defines the `GEN_example_variable` macro are the following.

- The macro name, by convention, always has the prefix `GEN_` (for "generate"), and the suffix is the name of the variable or data type for which the descriptor is being defined. (The definition of a descriptor for a particular data type is the usual case that will be encountered. This first example shows the descriptor for an individual variable for simplicity.)
- The first line must be the `DSTART` macro ("descriptor start"). Its only argument is the name to be assigned to the data object descriptor array. By convention, this name always has the prefix `D_` (for "descriptor"), and the suffix is the name of the variable or data type for which the descriptor is being defined.
- The following one or more lines each is an invocation of the `DGEN` (for "descriptor generate") macro. This macro has 3 arguments.
  (a) The C language text that is used to define a variable.
  (b) A macro giving the type of the variable. The possible macros are `CHARTYPE`, `SHORTTYPE`, `INTTYPE`, `FLOATTYPE`, `LONGTYPE`, `DOUBLETYPE`, `POINTERTYPE`, `NESTEDTYPE`. Each of these matches the corresponding C language data type except `NESTEDTYPE` which is discussed below in another example.
  (c) If the variable is an array, the total number of elements in the array. If the variable is not an array, the third argument should be 1.
- The last line in the macro definition must be the `DEND` ("descriptor end") macro, which has no arguments.
- Recall that when defining a macro that includes multiple lines, the last character on each line should be a backslash. There cannot be any spaces or other characters on the line after the backslash.
- Take note of the layout of the code defining the `GEN_example_variable` macro. Because the actual text giving the definition of the data object is buried within this macro definition, it is important for legibility that the data object definition stand out as much as possible from the surrounding text that is there to handle the descriptor conventions. So, the names of the `DSTART`, `DGEN`, and `DEND` macros are justified to the left, a large space is left between these macro names and the variable definition text, and another large space is left between the variable definition text and the other macro arguments (including the comma separating the variable definition text from the macro arguments which follow it).

2. The descriptor for a `typedef` is very similar to the first example. This is an example of a descriptor for a particular data type. Just for variety, in this example the data object contains values of type `float`.

```
/*
Define the macro that will generate a data object descriptor.
```

```
This code goes in the CONTROLDEF section of the algorithm
master file.
*/
#define GEN_ftest \
DSTART (                                         D_ftest       ) \
DGEN   (             typedef float ftest[5];     ,FLOATTYPE,   5) \
DEND
/*
Invoke the macro to define the data type.
*/
GEN_ftest
```

Note that this example, unlike the first example above, requires the GEN_ftest line which will actually define the `typedef`.

3. The definition of a descriptor for a C language structure is slightly different because it is necessary to indicate when a data object contains a structure. This is because structures in C often have padding between individual components of the structure in order to maintain the proper alignment in the processor memory of the various data types used in the structure. In order to convert data formats properly when transferring data between processors with differing architecture, it is necessary to know about this padding. Here is an example of the code for a descriptor for a C language structure.

```
#define GEN_size_test0 \
SDSTART(           struct size_test0 {              ,D_size_test0  ) \
DGEN   (               int i;                       ,INTTYPE,     1) \
DGEN   (               float f1;                    ,FLOATTYPE,   1) \
SDEND  (                             };                            )
GEN_size_test0
```

Note that this example, unlike the first example above, requires the GEN_size_test0 line which will actually define the structure.

Note the following features in this code for a descriptor of a C language structure.

- The first macro used is `SDSTART` ("structure descriptor start") rather than `DSTART` as in the previous examples.
- The macro on the last line is `SDEND` ("structure descriptor end").
- The first argument for the `SDSTART` macro is the text for the beginning of a standard C language structure definition including the first brace.
- The `SDEND` macro has an argument which is the final brace and semicolon of the text for the C language structure definition.

- Each of the individual components of the structure are defined, as in the examples above, with the `DGEN` macro with, as arguments, the text for the variable definition, the macro for the variable type and the number of elements in the array.
- The suffix on the descriptor generation macro name (`GEN_size_test0`) and descriptor variable name (`D_size_test0`) is the structure type because the descriptor can be used for any instance of this type of structure.

4. In some cases one data object will contain another data object. For example, a structure of one type containing an array of structures of another type. The data object contained within another data object can be described by a "nested" descriptor. The descriptor for the nested data object is defined separately and referenced within the larger data object as in the example shown here.

```
#define GEN_size_test2 \
SDSTART(           struct size_test2 {       ,D_size_test2        ) \
DGEN   (                float f[3];          ,FLOATTYPE,         3) \
DGEN   (                double d;            ,DOUBLETYPE,        1) \
NDGEN  (                struct size_test0 nested[3];                \
                                             ,NESTEDTYPE,3,D_size_test0) \
DGEN   (                int *p;              ,POINTERTYPE,       1) \
DGEN   (                float f1;            ,FLOATTYPE,         1) \
SDEND  (                            };                            )
```

- The nested data object is defined using the macro `NDGEN` ("nested descriptor generate") and its type specified within the macro is `NESTEDTYPE`.
- The first three arguments of the `NDGEN` macro are the same as the arguments in the `DGEN` macro.
- The fourth argument of `NDGEN` is the name of the variable (`D_size_test0`) assigned to the descriptor that is pointed to as the nested descriptor.
- `D_size_test0` is turned into a pointer to the descriptor for the structure type `size_test0`. Therefore, in functions where the `GEN_size_test2` macro is invoked to generate a descriptor, the variable `D_size_test0` must have already been defined by invoking the `GEN_size_test0` macro, and, in order for the compiler to generate the necessary pointer the variable `D_size_test0` must have type `static`. So, unless `GEN_size_test0` has been invoked so that `D_size_test0` is a global variable, the type modifier `static` must be used when invoking `GEN_size_test0`, as shown in this example.

```
void example_function_2()
{
   static GEN_size_test0
   GEN_size_test2
   .
   .
   .
}
```

- Descriptor nesting is specifically designed to handle the case of a C language data type for which the variable definition and descriptor definition are given elsewhere. That is, a C language structure or a C language `typedef`. A nested descriptor can only point to a descriptor for one of these two types of data object.

5. Finally, here is one more example of a structure definition, incorporating definitions from the examples above. This structure contains multiple nested descriptors, one of which is an array of structures of type `size_test2`, with the other a descriptor for a data type that was defined using a `typedef` statement. Note that there are 2 levels of descriptor nesting in this example because the structure of type `size_test2` has nested within it a structure of type `size_test0`. There is no limit to the number of levels of nesting of descriptors.

```
#define GEN_size_test3 \
SDSTART(         struct size_test3 {                   ,D_size_test3  )   \
DGEN   (              float f;                         ,FLOATTYPE,   1)   \
DGEN   (              double d;                        ,DOUBLETYPE,  1)   \
DGEN   (              int i;                           ,INTTYPE,     1)   \
NDGEN  (              struct size_test2 nested[4];                        \
                                            ,NESTEDTYPE,4,D_size_test2)   \
DGEN   (              int *p;                          ,POINTERTYPE, 1)   \
NDGEN  (              ftest ft[2];                                        \
                                            ,NESTEDTYPE,2,D_typedef_test1)\
DGEN   (              float f1;                        ,FLOATTYPE,   1)   \
SDEND  (                           };                                 )
GEN_size_test3
```

6. In the examples above showing how to create a descriptor definition for a C language structure, each descriptor is for a single structure. Or, in other words, an array of structures with a single element. The number of array elements can be altered, if desired, each time the descriptor is used. This is illustrated in the following example. Note that what is altered is the `size` value in the first element of the array of structures of type `STRUCT DESCRIPTORS` (see Sec. A). This first entry in the descriptor is defined by the macro `SDSTART` to indicate the start of an array of structures.

```
void example_function_3()
{
   GEN_size_test0
   .
   .
   .
   /* Array of structures with
      2 elements. */
   D_size_test0->size = 2;
   putblock(...,D_size_test0,...);
   .
   .
   .
   /* Array of structures with
      10 elements. */
   D_size_test0.size = 10;
   putblock(...,D_size_test0,...);
}
void example_function_4()
{
   GEN_size_test0
   .
   .
   .
   /* Array of structures with
      5 elements. */
   D_size_test0->size = 5;
   putblock(...,D_size_test0,...);
}
```

There is a potential bug to watch out for when altering the `size` value in a descriptor. Recall the example above describing the usage of nested descriptors. In that case it was necessary to declare the nested descriptor to be type `static`. If, however, the `size` value of the nested descriptor is altered, be careful to set it to the correct value each time the descriptor is required because the `size` value will carry over to the next call to the function. This subtle problem is illustrated in the following example.

```
void example_function_4()
{
   /* Recall that the structure of type size_test0 is defined
   in a previous example to be nested within the structure of
```

```
      type size_test2. */

      static GEN_size_test0
      GEN_size_test2
      .
      .
      .
      /* Create a parameter data block containing a structure
      of type size_test2. */

      /* Be careful to set the size in the descriptor for
      size_test0 because that size value is altered below
      and that new size will still be there the next
      time this function is called. */

      D_size_test0->size = 1;
      putblock(...,D_size_test2,...);
      .
      .
      .
      /* Create a parameter data block containing a 5 element
      array of structures of type size_test0. */

      D_size_test0->size = 5;
      putblock(...,D_size_test0,...);
   }
```

7. Descriptors for arrays of the basic C language data types can be created as in the previous example using descriptor generation macros that have been predefined. The available descriptor generation macros are `GEN_char`, `GEN_short`, `GEN_int`, `GEN_long`, `GEN_float`, `GEN_double`, and `GEN_pointer`. Here is an example.

```
   void example_function_5()
   {
      GEN_int
      .
      .
      .
      D_int.size = 53;
      putblock(...,D_int,...);
   }
```

2.11.4.6 Specifying the alignment of a parameter data block

It may be necessary to control the storage address alignment for a parameter data block on the real time processor. For instance, any given data object will have a required storage address alignment that is determined by the type of data in the object and the type of processor. Typically, 4 byte alignment is required for data of type `float` and most processors require 8 byte alignment for data of type `double`. The data object descriptor (Sec. 2.11.4.5) provided when a parameter data block is created (Sec. 2.11.4.2) is used to determine the minimum storage alignment requirement for the data in the block. The storage address on the real time processor will have at least this minimum alignment.

A particular byte alignment larger than the minimum alignment might be required. This might be the result of a requirement imposed by a particular function for which the data will be an argument. For instance, custom written linear algebra routines requiring 8 or 16 byte alignment for an array of floats which otherwise would normally be 4 byte aligned.

To specify the byte alignment of the parameter data block on the real time processor use one of the following values. (This is only relevant for parameter data blocks that are to be loaded into the memory of a real time processor. For other parameter data blocks use the value `BLOCK_ALIGN_0BYTES`).

- `BLOCK_ALIGN_0BYTES`: No special alignment is required so the normally required minimum alignment of the data object in the parameter data block should be used. This is also the appropriate value to be specified if the data block will not be loaded into real time memory.

- `BLOCK_ALIGN_2BYTES`: 2 byte alignment (appropriate for data of type `short int`). This value only has an effect if the normally required alignment of the data object in the parameter block is smaller than 2 bytes.

- `BLOCK_ALIGN_4BYTES`: 4 byte alignment (appropriate for data of type `float` or `int`). This value only has an effect if the normally required alignment of the data object in the parameter block is smaller than 4 bytes.

- `BLOCK_ALIGN_8BYTES`: 8 byte alignment (appropriate for data of type `double`). This value only has an effect if the normally required alignment of the data object in the parameter block is smaller than 8 bytes.

- `BLOCK_ALIGN_16BYTES`: 16 byte alignment (appropriate for routines on the i860 processor that use a single instruction to load 4 floating point registers simultaneously). This value only has an effect if the normally required alignment of the data object in the parameter block is smaller than 16 bytes.

- `BLOCK_ALIGN_32BYTES`: 32 byte alignment (appropriate for alignment to a cache line on some machines like the i860 processor). This value only has an effect if the normally required alignment of the data object in the parameter block is smaller than 32 bytes.

- `BLOCK_ALIGN_64BYTES`: 64 byte alignment (appropriate for alignment to a cache line on some machines). This value only has an effect if the normally required alignment of the data object in the parameter block is smaller than 64 bytes.

2.11.4.7 Parameter block storage order

The parameter data block creation routine takes an integer argument, the block order index, that is specified using a macro as described below. This argument is only used for parameter data blocks that are to be copied to a real time processor and may be set to `NO_BLOCK_ORDER` (or 0) otherwise. The block order index is used to identify a given parameter data block on the real time processor so each parameter data block must have a block order index that is unique in the group of blocks to be copied to a given virtual CPU.

A macro for the block order index is declared in the `CONTROLDEFS` section of the algorithm master file. The macro name has the form `Bvcpu_algorithm_block` where `vcpu` is the letter index indicating the virtual CPU of the category (e.g. A, B, :.), `algorithm` is the identifier for the control algorithm and `block` is a name for the parameter data block. The values for the block order index macros should be assigned sequential integers beginning with 1.

For example, if the algorithm `example` has 2 blocks on CPU A (`my_block` and `their_block`) and 1 block on CPU B (`other_block`) the macro definitions would be the following.

```
#define BA_EXAMPLE_MY_BLOCK 1
#define BA_EXAMPLE_THEIR_BLOCK 2
#define BB_EXAMPLE_OTHER_BLOCK 1
```

The block order index is used for two purposes.

- The block order index values for all of the parameter data blocks to be stored on a given real time processor for a given shot phase are sorted in numerical order and used to determine the storage order for the blocks in the area of memory allocated on the real time processor to parameter data.
- Code on the real time processor uses the block order index as the index into an array of pointers to obtain the pointer to the parameter data block (Sec. 2.11.7.1).

2.11.4.8 Sharing a parameter data block

As described in Sec. 2.11.4, parameter data are assocated with a shot phase. If the phase is deleted, then the parameter data is deleted along with the phase. The content of a parameter data block will usually get reset to the defaults for that block when a new phase is created or an existing phase is reinitialized.

However, if a parameter data block is created with the `SHARED` keyword in the options argument (Sec. 2.11.4.4), then a "system" copy of that block is created external to the copy created in the phase. The system copy of the block has a prefix inserted before the name.

The prefix indicates whether it is shared only by the algorithm, only by the category, or is shared system wide.

When the parameter data block is retrieved using a getblock function (Sec. 2.11.4.9), the system copy of the block descriptors (Sec. 2.11.4.5) and data are returned rather than the copy specific to the phase. This allows easy access to data in other phases, even if the phases are in other categories.

This is especially true if the block is associated with a static data item. Two phases using the same algorithm would share the data in a static data item. A change to the item's data from either phase would cause a change to the other phase.

If the parameter data block is shared system wide, then the static data item may appear in multiple algorithms in multiple categories. Again, a change to the data in one static data item would cause a change to the data for all the other data items whether the change comes from a static data item editor in the user interface or from the restore of the static data item to any of the phases.

Shared parameter data blocks are useful if multiple algorithms need access to the same data. If a shared block is associated with a static data item, then the item can be found in all algorithms that need it. This allows the user to see the item in the algorithm being viewed, rather than having to go to a different phase and/or category to view the item.

In addition to being shared in the waveform server, a shared block is shared on the real time processor by multiple phases. If changes are made to the block in real time, then those changes would be available to any phases that run later during the discharge. One copy of the block in memory is pointed to by each phase that shares the block.

Some notes about shared parameter data blocks.

1. When the shared block is created, usually in the `alg_parameters` routine, it is almost a requirement that the `NOOVERWRITE` option be specified. This option indicates that the block's data should only be written if the system copy of the block does not exist. Otherwise, without this option, the data would be overwritten with default values whenever the `alg_parameters` routine is called for one of the algorithms that use the shared block, e.g., when a new phase is created.

2. The system copy of a shared block is deleted only when all phases that share the block are initialized in a restore, get deleted, or get re-initialized to their default values. Only in these cases will the system copy not exist and a new system copy be created the next time an `alg_parameters` routine is called. To avoid this deletion specify the `NODELETE` keyword when the shared block is created. With this option the system copy of the block will never be deleted.

3. A shared block, by default, is shared on the real time processor. Any phase using the block will have the same pointer to the memory that contains the block's data. To keep the block from being shared on the real time processor, specify the `RTNOSHARE` option when the block is created. This is a phase specific option so it can be shared

in one algorithm and not shared in another. However, the block will still be shared in the waveform server.

4. There are three possible types of shared blocks (see Sec. 2.11.4.4). An algorithm specific shared block with the same name as a category specific shared block (or a system wide shared block) do not point to the same block. A prefix is added to the name for the system copy of the block so these three types can be distinguished from each other.

5. Shared parameter data blocks require the newer versions of the parameter data functions, i.e. `putblock` and `getblock`. `put_param_block` and `get_param_block` cannot be used.

2.11.4.9 Retrieving parameter block data

The `alg_vectors` function typically might want to combine one or more parameter data blocks and perhaps one or more waveforms, to produce processed target vector elements or parameter data blocks. To do this the `alg_vectors` function uses one of the following routines to access a parameter data block (typically, but not necessarily, containing "raw" parameter data from the user interface).

- `getblock`: (Sec. 14.3.14) This routine provides access to the parameter data and, in addition, many details about the parameter data block such as the flags, alignment and block order specification. Typically this much detail isn't needed and one of the 2 routines listed next would be used.

- `getblock_data_ptr`: (Sec. 14.3.15) Given the name of the parameter data block, this routine returns a pointer to the data in the block.

- `getblock_data_copy`: (Sec. 14.3.16) Given the name of the parameter data block, this routine returns a copy of the data from the parameter data block. The memory for the copy of the data is obtained using `malloc`. When the program no longer needs the copy of the data, the memory should be freed using `free`.

It could also be necessary to access parameter data that is located in another phase. This would typically be a parameter data block that is part of an external data item declared by the algorithm (Sec. 2.10.6). To access parameter data in another phase the function `get_external_param_block_ptr` (Sec. 14.3.17) is used.

IMPORTANT NOTE:
The routines `getblock` and `getblock_data_ptr` both return a pointer to the location in the waveform server where the data are stored for a specified parameter data block in a specified shot phase. When any parameter data block for the same phase is replaced, the locations of all parameter data blocks for that phase are likely to change. This will invalidate any pointers already obtained to parameter data for that phase. This feature (admittedly

not a desirable one) introduces a relatively easy way to introduce a bug that will cause the waveform server to crash.

To systematically avoid this possible problem, the following programming practice should be used in a routine that uses parameter data and creates or replaces one or more parameter data blocks.

1. First, obtain all necessary parameter data block pointers.
2. Then create all new parameter data in temporary memory.
3. Do all creation or replacement of parameter data blocks.
4. Free the temporary memory created in step 2.

By following this method, invalid pointers will not be used accidentally.

2.11.4.10 Replacing parameter block data

When the `alg_vectors` function computes processed data in response to the receipt of new raw data and stores the processed data into a parameter data block, usually the parameter data block will already exist. In this case the existing parameter data block will be completely replaced using the new parameter data provided. When the data in a parameter block are replaced, it is normally desirable that the various flag, alignment and block order values remain the same.

There are several ways to handle replacing a parameter data block.

- The best programming practice is to create a given parameter data block in only one place in the algorithm code (see also Sec. 2.11.4.1). In this way the flag, alignment and block order values need only be known in one function. Then, the processes of creating or replacing a parameter data block are the same. One of the parameter data block creation routines described in Sec. 2.11.4.2 is used.

- If it is necessary to replace a parameter data block without handling the flag, alignment and block order values the function `replace_param_block` (Sec. D.8.17) can be used. This function takes as arguments only the block name, block data and the array of block descriptors. Other information normally provided to create a parameter data block is taken from the existing block that is being replaced. Note that these routines will do nothing if the specified parameter data block doesn't actually exist already.

- Companion functions can be used to replace a specific type of block. These functions take a count of values rather than the array of block descriptors. These functions are
  - `replace_param_block_chars` (Sec. D.8.18)
  - `replace_param_block_shorts` (Sec. D.8.19)

  - `replace_param_block_ints` (Sec. D.8.20)
  - `replace_param_block_longs` (Sec. D.8.21)
  - `replace_param_block_floats` (Sec. D.8.22)
  - `replace_param_block_doubles` (Sec. D.8.23)
  - `replace_param_block_strings` (Sec. D.8.24)

- Finally, if it is necessary to replace an existing parameter data block while changing only some of the flag, alignment or block order values, use `getblock` to obtain the parameter data along with any of the flag, alignment or block order values. Then those values can be used with one of the `putblock*` functions (Sec. 14.3) to replace the parameter data block.

2.11.5 Parameter data copy routines

The central storage location for the parameter data is the waveform server process so it is often necessary to copy the data to other processes, either for use during execution of a discharge or for editing by the user interface. Also, facilities are needed to add the parameter data to the shot setup archive and to restore parameter data from the setup archive. There is provision for each algorithm to specify a set of functions unique to that algorithm that provides these parameter data copy functions. The names of these functions are specified in the algorithm descriptor, described in Sec. 4.6.1.

Normally, though, special functions are not required for an algorithm. Generic functions provided by the PCS are used instead. The names of the generic functions are provided in the discussion of the algorithm descriptor (Sec. 4.6.1), in the subsections which follow, and in the following list.

- To provide data for archiving with the PCS: `pcs_forarchive`.
- To receive data when restoring an archived PCS setup: `pcs_fromarchive`.
- To copy data to the real time processor and the host_cpu process: `pcs_forhost`.
- To provide data for editing to the user interface: `pcs_foruser`.
- To receive the data returned by the user interface: `pcs_fromuser`.
- To return the number of bytes to be copied to the real time processor: `pcs_paramsize`.

For algorithms that require custom parameter data copy functions, there is a discussion in the following subsections of the purpose for each of the parameter data copy functions. The most common example of a situation in which a custom routine is necessary is an assistant to the routine that receives the data when a PCS setup is restored. Sometimes a custom function is used to adapt an older parameter data format to a newer format used by an updated version of a control algorithm.

2.11.5.1 Data to be archived

For each algorithm there must be a function that returns the parameter data sorted properly for archival with the PCS setup. The name of this routine is specified in the descriptor for the algorithm (see Sec. 4.6.1).

Normally the standard routine `pcs_forarchive` (Sec. 14.3.32) is specified.

In general the parameter data are a mixture of various data types. In order to allow the PCS setup to be transported to processors with different data formats, the PCS data must be separated into arrays of the various data types (e.g. float, int etc). Then the data archive routines will be able to automatically determine how to adjust the data formats for the target processor. Thus, the routine that returns the parameter data blocks formatted for archival must separate the various data types from the parameter blocks into arrays with a single type. This is one of the purposes for providing the parameter data block descriptor (Sec. 2.11.4.5). The `pcs_forarchive` function uses the descriptor to determine how to separate the data types from within the parameter data blocks.

The only parameter data blocks that are archived are those that have specified `"ARCHIVE"` in the options argument when the block was created (Sec. 2.11.4.2). The normal procedure would be to archive only "raw" parameter data blocks, that is, those that are edited by the user. Parameter data blocks used only as local storage in the waveform server or that are processed data computed for the purpose of use in real time normally wouldn't need to be archived because the data in those blocks are derived from the raw data.

2.11.5.2 Restoring data

For each algorithm there must be a function that receives parameter data that has been restored from the archived PCS setup, and stores it in the parameter data storage location for the specified shot phase. The name of this routine is specified in the descriptor for the algorithm (see Sec. 4.6.1).

Normally the standard routine `pcs_fromarchive` (Sec. 14.3.33) is specified. This routine receives the data sent from the setup restoration functions and determines the names of the parameter data blocks that exist. Then, any parameter data blocks that already exist in the parameter data storage area for the specified shot phase are replaced with the blocks provided by the setup restoration functions. If blocks with names that do not already exist for the specified phase are in the archive, they are ignored. This allows some flexibility in altering a control algorithm (by changing the names or function of parameter data blocks) while still allowing compatibility with the setup archived for older discharges using the previous algorithm version.

Parameter data blocks that exist in the waveform server which were created with the `NORESTORE` option are not restored. Sometimes it is desirable to archive some data only to save a record of the data that were actually used. But, these data might not need to be used in the future so they wouldn't normally be restored. An example would be a calibration factor for a diagnostic. It would be desirable to save the value actually used on the discharge for future reference, but for new discharges the current value of the calibration factor would always want to be used. (Note: presently there isn't actually a way to override the no-restore option in order to view the archived data. This is a feature that needs to be added.)

2.11.5.3 Maintaining backward compatibility

Sometimes it is desirable to change the name of a parameter data block, especially a block associated with a static data item. For example, a static data item created for a power supply may need to be renamed if another static data item is needed when another power supply is added. In order to restore data for the older static data item, the older name in an archive must be converted to the newer name.

One way of handling this name change is to add a `NAME_CHANGES` entry as discussed in section 4.9. In most cases, this will handle the name change without any additional programming.

However, along with the name change, the structure or size of the data may have changed as well. If the data item was always a standard static data item (Sec. 2.12), then the change to the structure is handled automatically as discussed in Sec. 2.12.1.

But what if the parameter data block was not always part of a standard static data item? Then more programming is required to restore this block from old archives. A custom restore routine is required in this case. This routine must avoid the overwriting of the existing block since this will change the structure back to the older format. There are two ways to handle the restore.

1. If the name of the block hasn't changed, then the standard restore function, `pcs_fromarchive`, must be told not to overwrite the existing block. This is done by setting values in the `fromarchive_info` structure which is passed to `pcs_fromarchive`. The values to set are the names of the parameter data blocks that should not be overwritten.

   For example:

```
void alg_fromarchive(
    struct shotphase *phase,
    char *asciidata,    int asciicount,
    short *shortdata,   int shortcount,
    int *intdata,       int intcount,
    float *floatdata,   int floatcount,
    double *doubledata, int doublecount,
    struct fromarchive_struct *fromarchive_info)
{
char *LEGACY_GROUPS[] = { "calibration data"};
int zero,*itemp,inum;
```

```
    fromarchive_info->legacy_groups_count = 1;
    fromarchive_info->legacy_groups = LEGACY_GROUPS;
/*
Older archives have the calibration data stored with a different
structure than the current structure.  The count of structures
was also stored as a single integer.  Create an archive block
with a count of zero.  If this block is overwritten, then
we know that an older version of the block was restored.
*/
    zero = 0;
    putblock_ints(phase,"calibration data :count_a",NULL,
                  NO_BLOCK_ORDER,1,&zero);
/*
Follow the standard procedure to restore parameter data blocks.
*/
    pcs_fromarchive(
                   phase,
                   asciidata, asciicount,
                   shortdata, shortcount,
                   intdata,   intcount,
                   floatdata, floatcount,
                   doubledata, doublecount,
                   fromarchive_info);
/*
If an older version of the structure was restored, then
transfer the data from the old structure to the new one.
*/
    if (strcasecmp(fromarchive_info->new_name, "calibration data") == 0)
    {
        itemp = (int *) getblock_data_ptr(phase,"calibration data :count_a");
        inum = itemp[0];
        if (inum > 0)
        {
            old_data = (struct old_struct *)
                       getblock_data_ptr(phase,"calibration data :data_a");
            new_data = (struct new_struct *)
                       getblock_data_ptr(phase,"calibration data :data");
            /* move data from old structure to new structure */
        }
    }
```

The `pcs_fromarchive` routine will look at the names provided and, instead of overwriting these blocks, the blocks will be restored to new blocks that have "_a" appended to their names. Then it is left to the algorithm's fromarchive function to compare the size of the existing block with the restored block and transfer the data from the restored block to the existing block.

2. If the name of a parameter data block was changed, then a second method is required. In this case, the parameter data block with the old name must exist in the waveform server in order for the block from the archive to be restored. This requires that a dummy block be created. The dummy block would have a format that is identifiable as not actually containing data from the old storage format. In a custom restore routine, `pcs_fromarchive` is first called. Then, a check is made to see if the dummy block was changed by the restore procedure. If so, the old data are copied from the dummy block with the old format into the new format block. Then the dummy block is replaced with its recognizable format again.

2.11.5.4 For the real time CPU

For each algorithm there must be a function that returns the parameter data to be loaded into the real time CPU's memory. This routine is specified in the descriptor for the algorithm (see Sec. 4.6.1).

Normally the standard routine `pcs_forhost` (Sec. 14.3.29) is specified. This routine copies the parameter data blocks that have indicated in their usage flag value that the data should be made available on one or more the of the real time processors or in one or more of the host_cpu processes. If some special customization is required a different function can be specified for the algorithm.

See Sec. 2.11.7 for information on how the algorithm code can locate the parameter data on the real time processor.

2.11.5.5 For the user interface

The algorithm author can give the PCS operator the option of modifying one or more blocks of parameter data. For this purpose a custom user interface using the X windows capability of IDL must be written. Normally the block structure of the parameter data is hidden from the operator. The operator sees an interface that allows modification of data displayed in a manner that relates well to the control algorithm.

The IDL user interface code can request from the waveform server the current values of the parameter data. The user then makes any required changes and then the IDL code sends the new parameter data blocks back to the waveform server.

For each algorithm there must be a function that returns the parameter data that is to be sent to the user interface. The name of this routine is specified in the descriptor for the algorithm (see Sec. 4.6.1).

Normally the standard routine `pcs_foruser` (Sec. 14.3.30) is specified. This routine copies the parameter data blocks that have indicated in their usage flag value that the data should be made available to the user interface. If some special customization is required a different function can be specified for the algorithm.

(Coming: the `pcs_foruser` routine will be modified to return only a group of associated parameter data blocks. This will be the preferred method for the user interface to access parameter data blocks for editing.)

See Sec. 2.11.8 for a description of how to create the user interface editor.

See Sec. 2.10.1.2 for information on how to connect the proper user interface routine and a group of associated parameter data blocks.

For certain types of static data, support code already exists including the user interface routine. See Sec. 2.12 for descriptions of these standard static data types.

2.11.5.6 From the user interface

For each algorithm there must be a function that receives parameter data from the user interface and stores it in the parameter data storage location for the specified shot phase. The name of this routine is specified in the descriptor for the algorithm (see Sec. 4.6.1).

Normally the standard routine `pcs_fromuser` (Sec. 14.3.31) is specified. This routine receives the data sent from the user interface and determines the names of the parameter data blocks that were received. Then, any parameter data blocks that already exist in the parameter data storage area for the specified shot phase are replaced with the blocks provided by the user interface. If the user interface provides blocks with names that do not already exist for the specified phase, they are ignored. (Normally this would mean that there is a programming error.)

2.11.5.7 Parameter data size routine

For each algorithm there must be a function that returns the total number of bytes required on each of the real time processors for storage of parameter data. The name of this routine is specified in the descriptor for the algorithm (see Sec. 4.6.1).

Normally the standard routine `pcs_paramsize` (Sec. 14.3.34) is specified.

2.11.6 Data computation routine

As described in Sec. 2.11.1, the data stored in the waveform server's address space can be a mixture of data blocks to be used in real time, to be modified by the user interface, and to be archived/restored. Any, all or none of the data could fall into each category.

A given parameter data block could be a combination of waveforms and other parameter data. So, it is possible that when any of these "raw" input data sets change, the parameter data could need to be recomputed. The algorithm author would provide a custom routine to do this computation. The routine would be called by the algorithm's `alg_vectors` routine (Sec. 2.10.5).

The call format for this computation routine is up to the algorithm author since it is called directly from the `alg_vectors` routine which is also written by the algorithm author.

2.11.7 Accessing parameter data in real time

This section describes two methods that code on the real time CPU can use to obtain a pointer to a particular parameter data block. They differ in their ease of programming and speed of execution in real time.

1. Read the pointer from a list of parameter data block pointers. This is probably the easiest method to program. It will be slightly slower in real time execution. This method can also be used for preshot initialization or postshot cleanup routines.

2. Read the pointer from an element in the pointer target vector. Code implemented in the `alg_vectors` routine can cause the parameter data block pointer to be precalculated and the result written to a pointer target vector element. Then, the real time code can simply read the pointer from the pointer target vector. Recall that the `alg_vectors` routine (Sec. 2.10.5) is part of the code executing in the waveform server process for a given algorithm.

   This is the fastest method to obtain the pointer in real time, so it is the method to use to reduce the execution time of real time code. This is also a convenient method to use in order to use a waveform to indicate which element of an array of values in a parameter block should be used at any given time during a discharge.

   However, pointer target vector values are only valid in real time, so this method cannot be used by preshot initialization or postshot cleanup routines.

Either option can be used by the routines executed when a new phase is begun in real time (the phase entry routine and phase start routine) because when these routines are called the content of the pointer target vector elements has been loaded.

2.11.7.1 Parameter block pointer array

An array of pointers to the parameter data blocks for a given shot phase is located at the beginning of the parameter data storage area for that phase. The block order index (Sec. 2.11.4.7) for a given parameter data block is used as an index into this array to obtain the pointer to the parameter data block.

For example, given the pointer to the beginning of the parameter data,

```
void **parameter_data
```

and the block order index macro for an example parameter data block, `BA_MYALG_BLOCK_A`, then

```
parameter_data[BA_MYALG_BLOCK_A-1]
```

is the pointer to the parameter data block. Note that the block order index values start with 1 so that the array index is obtained by subtracting 1.

The possible sources of the pointer to the parameter data for a phase depend on what type of routine is executing.

- In preshot initialization and postshot cleanup routines, a pointer to the descriptor of the shot phase is provided as a function argument. The pointer to the parameter data for the phase is available within the phase descriptor as shown in the following example.

  ```
  void alg_rt_initialize(struct rtshotphase *rtshot,
                         struct rt_heap_misc *rtheap)
  {
     void **parameter_data, *parameter_data_block;

     parameter_data = (void **)rtshot->parameter_data;
     parameter_data_block = parameter_data[BA_MYALG_BLOCK_A-1];
     .
     .
     .
  }
  ```

  This method can be used in any code where the shot phase descriptor is accessible.

- The function that executes in real time receives the pointer to the "real time heap" structure (`rtheap`) as the function argument. The pointer to the parameter data for the phase currently executing is available from several locations accessible through `rtheap`. (This method can only be used in real time. It cannot be used in pre-shot initialization and post-shot cleanup functions because there is no "currently executing" phase).

  Three examples are given below. Of these examples, the first shows the method which is the easiest to use and the fastest to execute. In the examples, the index of the category with identifier `example` on virtual CPU `A` for that category is given by the macro `CATA_EXAMPLE`. This index can be obtained as described in Sec. 2.16.1. The block order index assigned to the desired parameter data block is given by the macro `BA_MYALG_BLOCK_A` as described in Sec. 2.11.4.7. Note that the macro values all are based at 1 so that 1 must be subtracted to obtain the 0 based index required for use with C language arrays.

1. The array `rtheap->current_parameter_data` specifies, for each of the categories executing on the particular real time processor, the pointer to the parameter data storage area for the phase presently in use for that category.

```
void my_algorithm(struct rt_heap_misc *rtheap)
{
   void **parameter_data, *parameter_data_block;

   parameter_data =
      rtheap->current_parameter_data[CATA_EXAMPLE-1];
   parameter_data_block = parameter_data[BA_MYALG_BLOCK_A-1];
   .
   .
   .
}
```

2. The array `rtheap->current_phase` specifies, for each of the categories executing on the particular real time processor, the pointer to the descriptor for the phase presently in use for that category. This descriptor contains the pointer to the parameter data storage area for that phase.

```
void my_algorithm(struct rt_heap_misc *rtheap)
{
   void **parameter_data, *parameter_data_block;

   parameter_data =
    (void **)rtheap->current_phase[CATA_EXAMPLE-1]->parameter_data;
   parameter_data_block = parameter_data[BA_MYALG_BLOCK_A-1];
   .
   .
   .
}
```

3. The array `rtheap->current_phs_seq` contains the pointer to the descriptor for the shot phase sequence currently in use for each of the categories executing code on the real time processor. The shot phase sequence descriptor contains the pointer to the descriptor for the shot phase currently in use for the category. The shot phase descriptor contains the pointer to the parameter data storage area for that phase.

```
void my_algorithm(struct rt_heap_misc *rtheap)
{
   void **parameter_data, *parameter_data_block;
```

```
        parameter_data =
         (void **)rtheap->current_phs_seq[CATA_EXAMPLE-1]->\
                                          current_phase->parameter_data;
        parameter_data_block = parameter_data[BA_MYALG_BLOCK_A-1];
        .
        .
        .
    }
```

- The examples above focused on obtaining parameter data for the category and phase in which the code is executing, or the currently executing phase in another category. It is also possible to obtain pointers to parameter data blocks for any arbitrary phase. The method described here can be used in any function where the `rtheap` pointer is available, in particular in real time code and in the preshot initialization and postshot cleanup functions.

  The `rtheap` structure contains pointers to the lists of descriptors for all of the phases defined for all of the categories. `rtheap->phaseentries` is an array of pointers, one for each category executing on the real time CPU. Each of these points to the array of phase descriptors for a category. `rtheap->phase_count` is an array of integers that specifies how many phases are defined for each category. So, using these two entries in `rtheap` it is possible to search through the list of phases for a given category to find the descriptor for the desired phase. Then, the descriptor contains the pointer to the parameter data for that phase which can be used as described earlier in this section to obtain the pointer to a particular parameter data block.

  The remaining issue is how to identify the desired phase. Probably the best way would be to search for a phase with a given name. However, the phase names are presently not available on the real time processors. The other possible way to select a phase is by finding the phase that uses a particular algorithm. The phase descriptor contains the index of the algorithm used by that phase which can be compared to one of the algorithm index macros that are described in Sec. 2.16.4.

  Here is an example of code that searches for a phase in the category with the identifier `example` that uses the algorithm `myalg` and obtains the pointer to a parameter data block for that phase.

```
for(i = 0; i < rtheap->phase_count[EXAMPLE_CATEGORY_INDEX - 1]; i++){
   if((rtheap->phaseentries[EXAMPLE_CATEGORY_INDEX - 1][i]).algorithm ==
       ALG_EXAMPLE_MYALG){
      parameters =
         (void **)rtheap->phaseentries[EXAMPLE_CATEGORY_INDEX - 1][i].\
              parameter_data;
            /* Here the parameter data block contains an array of floats. */
            my_parameter_data = (float *)parameters[BX_MYALG_DATA-1];
            break;
        }
    }
```

Here `BX_MYALG_DATA` is the block order index for the desired parameter data clock. The use of `X` in the index rather than an indicator of a virtual CPU is a convention that indicates that the block index is the same on all of the virtual CPUs for that category.

2.11.7.2 Parameter data pointers in the pointer target vector

The algorithm author can arrange for a pointer to a parameter data block to be placed into an element of the pointer target vector. In real time this value can be easily accessed. This is the preferred method for locating a parameter data block in real time because it is the fastest method. Also, it is consistent with the general theme of PCS real time code that the target and shape vectors are used to make data available that might need to be communicated between the code for various categories.

For extra flexibility, optionally, this real time pointer can point to a location at a specified offset from the beginning of the parameter data block. This offset can be calculated from the vertices of a waveform. So, the pointer can be made to change as a function of time under the control of the operator. This feature can be used to allow the operator to specify which structure, out of an array of structures, contains the data that should be used at any given time during the shot.

In the `alg_vectors` function, the macro to use is (Sec. 14.1.5):

```
case_ptrt_paramblock_wvmp(int data_entry_number,
                          int target_vector_index,
                          int virtual_cpu, char *waveform_name,
                          STRUCT_DESCRIPTOR *offset,
                          STRUCT_DESCRIPTOR *scale_factor,
                          int block_order_index)
```

This generates values which will be interpreted as offsets into the block of parameter data specified by `block_order_index`. During shot setup the address of the parameter data block is added to these offset values so that they become pointers into the parameter data.

The offset value is calculated from the waveform given by `waveform_name` by converting the waveform vertices to integers by truncation, multiplying these values by `scale_factor` and adding `offset`. The waveform vertex Y values would normally be integers indicating which of an array of values should be indicated by the pointer. The `offset` and `scale_factor` would normally be counts of bytes. The `scale_factor` would be the size of a single element in the array of values and the `offset` would be the number of bytes in the parameter data block preceding the first element in the array. Because the size of a particular data object can vary depending on the processor architecture, the `offset` and `scale_factor` are computed from data object descriptors (Sec. 2.11.4.5) taking into account the type of processor indicated by `virtual_cpu`. If either `offset` or `scale_factor` is given as a null pointer, the corresponding size in bytes set to 0.

So, for an example of an array of matrices, `offset` would be the number of bytes into the parameter data block of the first matrix (typically this would be 0) and `scale_factor` would be the size in bytes of a matrix. The waveform would have values 0, 1, 2 etc. to indicate that the first, second, third etc. matrix should be used.

If `waveform_name` is a null pointer then no waveform is used. Instead, a single pointer target vector value assigned to $t = 0$ (beginning of the shot phase) is generated pointing to the byte `offset` bytes into the parameter data. In this case `scale_factor` is ignored.

To summarize, the steps necessary to have the pointer to a parameter data block available in a pointer target vector element are the following.

1. Define the macro for the pointer target vector element number in the `CONTROLDEFS` section of the master file (Sec. 4.3).

2. Register this pointer target vector element in the `vector_info` array in the `WAVEGLOBALS` section of the master file (Sec. 4.5).

3.
   - If the offset into the parameter data block is specified with a waveform, define the waveform and put the data entry number of the pointer target vector element in the "dependent processed data" array in the waveform definition structure (Sec. 4.7). This is the list of processed data that must be recomputed when the waveform vertices change.
   - If the parameter data block pointer won't be defined by a waveform but instead will be a fixed value (not changing in time), put the data entry number in the array in the `categories_algorithms` structure that lists processed data that doesn't need to be recalculated when raw data changes, but instead must simply be calculated once (Sec. 4.6).

4. Put the `case_ptrt_paramblock_wvmp` code in the `alg_vectors` the function.

5. In real time, access the parameter data block pointer as in the following example where the parameter data block is an array of floats and `PTR_ELEMENT_NUMBER` is the macro for the pointer target vector element number.

```
float *mydata;
float **pointertargets;
```

```
    pointertargets = (float **)rtheap->pointer_target;
    mydata = (float *)pointertargets[PTR_ELEMENT_NUMBER - 1];
```

2.11.8 The parameter data editor in the user interface

It is often desirable to have a user interface for modification of the data in a group of parameter data blocks. If an algorithm uses a parameter data block format that isn't supported by one of the standard block structures (described in Sec. 2.12), each of which comes with a user interface, it is necessary to write custom user interface code. This user interface is referred to here as a "static data editor."

Most of the code in the static data editor would be standard IDL language making use of the available widget routines to make a graphical user interface. The PCS-specific portions of the static data editor code are of two types.

1. A set of low level routines that provide basic functions.
    - Routines that allow the editor to be installed into the user interface and communicate with the infrastructure code that makes up the bulk of the user interface.
    - Routines that allow the static data editor to retrieve parameter data from the waveform server, modify the data, and return the data to the waveform server.
2. A set of upper level routines that implement most of the functions required to create a static data editor. Use of these routines makes implementation of a static data editor reasonably easy because they do most of the necessary work and they hide the details of the lower level functions.

This section outlines the code for a static data editor. Detailed specifications for the static data editor support routines are given in the reference section 7.6.

2.11.8.1 Calling format

Each static data editor is an IDL procedure with the following call format.

```
pro staticdata_editor, ix, dx, data_item
```

All arguments provide inputs. The values that the arguments have on input to the procedure should not be changed. However, the arguments can provide the editor with information it needs to perform its tasks.

A static data editor routine is called when the user selects a data item in the navigation window that has the string "StaticData" as the prefix to its description string. The name of the editor routine to call for a given static data item is specified in the `xunits` element of the data item structure in the algorithm master file (Sec. 4.7).

The procedure arguments are the following.

- `ix`: an obsolete value which should be ignored by static data editor procedures.
- `dx`: a structure that provides the data context of the user interface, that is, variables that are set by code that is part of the user interface but not part of the static data editor. The convention is that routines in the user interface, including static data editor procedures, always name this variable `dx` in order to keep the code understandable. The `data_item` structure contains the information from the data item definition structure in the waveform server (Sec. 4.7). This includes the following.
  - `data_item.name`: a string giving the name of the data item selected by the user. All static data editors should be written to handle a specific type of data, rather than a particular data item. The name can be used as a title or to determine which options are available in the editor.
  - `data_item.description`: begins with the string `StaticData` that identifies this data item as being associated with parameter data blocks.
  - `data_item.xunits` contains the name of the static data editor procedure.

  `data_item.description` and `data_item.xunits` are used by the user interface code to determine which static data editor procedure to call.

  All other elements of the `data_item` structure are available to be used to pass arguments to the static data editor procedure. The meaning of the arguments is unique to each editor. For instance, if the editor has options for its behavior, these arguments can be used to choose among the options for each data item for which the editor will be used. The values for the arguments must be constants that are provided in the data item definition structure (Sec. 4.7). The structure tags available to hold input arguments are the following.
  - `scale`: an array of four floating point values.
  - `yunits`: a string.
  - `restorepoint`: a string.
  - `restorescale`: a single floating point value.
  - `restoreoffset`: a single floating point value.
  - `archivepoint`: a string.

2.11.8.2 Using the standard display library procedures

The infrastructure has several IDL files which help to implement a static data editor. The main library of routines is in the file `displaylib.pro`. This library of procedures and functions handle the bulk of the work required by any static data editor. To use this library the editor must have the proper "hooks", that is, procedures with the proper names that do particular tasks.

All hook procedures in the editor file must have names that start with the file's name, e.g., `staticdata_editor.pro` would have procedures starting with "staticdata_editor_". These procedures are

- `staticdata_editor`: the main procedure which calls the display library procedure `display_init` (Sec. 2.11.8.8). The `display_init` procedure then handles all the steps to create the editor window and, by default, add the standard buttons to this window. It also calls the following "hook" procedures to handle specific tasks. This procedure has the following call format (Sec. 2.11.8.1).

  ```
  pro staticdata_editor, ix, dx, data_item
  ```

- `staticdata_editor_init`: this procedure creates the widgets which display information. This procedure is only called once to create the editor window. Unless the information is read only, the widgets should be created but not loaded with data. This procedure has the following call format (Sec. 2.11.8.3).

  ```
  pro staticdata_editor_init,info,dx,byte_array,widget_ids,user_data
  ```

- `staticdata_editor_load_defaults`: this procedure is provided with the default value of the parameter data blocks from the waveform server. The block(s) must first be extracted from the variable `byte_array` using the `extract_paramdata` function. This data then should be loaded into the widgets. The "uvalue" of each widget is not changed here. This procedure is only called when the user pushes the "load_defaults" button. The call format is (Sec. 2.11.8.4).

  ```
  pro staticdata_editor_load_defaults,info,dx,byte_array
  ```

- `staticdata_editor_load`: this procedure is provided with the current value of the parameter data from the waveform server. The block(s) must first be extracted from the variable `byte_array` using the `extract_paramdata` function. This data then should be loaded into the widgets. It also can set the "uvalue" of each widget to this current value. This allows the standard display procedures to compare whatever value is found in the widget to that which was originally loaded. This procedure is called when the editor is initialized and also when the user pushes the "cancel" button. The call format is (Sec. 2.11.8.5).

  ```
  pro staticdata_editor_load,info,dx,byte_array
  ```

- `staticdata_editor_update`: this procedure is used to send the parameter data back to the waveform server. It needs to extract from the widgets their current values and insert these values into the variable `byte_array` using the `replace_paramdata` function. The "update" procedure is called when the user pushes the "apply" button. The "load" procedure will then be called by the display library to reload the "uvalue" of each widget. The call format is (Sec. 2.11.8.6).

  ```
  pro staticdata_editor_update,info,dx,byte_array,errorback
  ```

2.11.8.3 staticdata_editor_init procedure

The `editor_init` procedure creates the widgets which display the information and optionally, allow the user to change the information. This procedure is only called once to create the editor window. Unless the information is read only, the widgets should be created but not loaded with data.

The arguments to this procedure are

- `info`: the anonymous IDL structure which contains all kinds of information needed by the display library to perform its tasks. This structure can be appended to in this procedure in order to store any settings or data that may be used in other editor procedures. This variable is passed to all procedures and is also the "uvalue" of the base widget.
- `dx`: the input structure argument to the editor (Sec. 2.11.8.1).
- `byte_array`: a byte array containing all the parameter data that has been retrieved from the waveform server. It is provided as an argument in case it is needed for an editor that only displays the parameter data. Usually, it is ignored here.
- `widget_ids`: output argument containing the widget ids of all the widgets created which can be handled in a structured way. This variable is optional and does not need to be set to anything. It is stored as part of the `info` structure after this procedure returns as `info.widget_ids`. See Sec. **??** for more details about how the `display_event` procedure handles different kinds of widgets.
- `user_data`: an output variable consisting of any data that the editor needs to store. This usually becomes a structure containing the original values of the parameter data. This variable is optional and does not need to be set to anything. It is stored as part of the `info` structure after this procedure returns as `info.user_data`.

2.11.8.4 staticdata_editor_load_defaults procedure

The `editor_load_defaults` procedure loads the widgets with the default value of the parameter data retrieved from the waveform server. The arguments to this procedure are

- `info`: the anonymous IDL structure which contains all kinds of information needed by the display library to perform its tasks.
- `dx`: the input structure argument to the editor (Sec. 2.11.8.1).
- `byte_array`: a byte array containing all the parameter data that has been retrieved from the waveform server.

2.11.8.5 staticdata_editor_load procedure

The `editor_load` procedure loads the widgets with the current information from the waveform server. The arguments to this procedure are

- `info`: the anonymous IDL structure which contains all kinds of information needed by the display library to perform its tasks.
- `dx`: the input structure argument to the editor (Sec. 2.11.8.1).
- `byte_array`: a byte array containing all the parameter data that has been retrieved from the waveform server.

2.11.8.6 staticdata_editor_update procedure

The `editor_update` procedure updates the parameter data with the current values of the editor's widgets so it can be sent back to the waveform server. The arguments to this procedure are

- `info`: the anonymous IDL structure which contains all kinds of information needed by the display library to perform its tasks.
- `dx`: the input structure argument to the editor (Sec. 2.11.8.1).
- `byte_array`: a byte array containing all the parameter data that has been retrieved from the waveform server.

2.11.8.7 staticdata_editor example

Here is an example of a simple editor which creates three widgets, one each for three values.

```
;*****************************************************************************
;FILE: staticdata_editor.pro
;Example of a static data editor.
;*****************************************************************************

;****************************************************************************
```

```
;PROCEDURE: staticdata_editor
; Main routine - need one parameter data block which will be saved.
;***************************************************************************
pro staticdata_editor,ix,dx,waveform

names = waveform.name
display_init,dx,"staticdata_editor",' ',err,info,$
    title='Example Editor',names=names,save=1

end

;****************************************************************************
;PROCEDURE: staticdata_editor_init
; Initialize the widgets for the editor
; They are not loaded here.
;****************************************************************************
pro staticdata_editor_init,info,dx,byte_array,widget_ids,user_data

noyes = ['no','yes']
maggains = ['0.5','1.0','2.0']

base = widget_base(info.base,/column)

rowbase = widget_base(base,/row)
maggainid = widget_droplist(rowbase,/frame,value=maggains,$
    title="magnetics gain: ")

savebaseid = widget_droplist(rowbase,/frame,value=noyes,$
    title="save baseline data:")

basecntid = cw_field(base,xsize=18,/frame,/all_events,$
    title='baseline sample period (sec):',/floating)

widget_ids = [maggainid,basecntid,savebaseid]

user_data = {staticdata,$
             maggain: float(0.0),$
             baseline_count: long(0),$
             savebasedata: long(0)}

info = create_struct(info,'maggains',maggains) ;example of adding on
```

```
end

;*****************************************************************************
;PROCEDURE: staticdata_editor_load_defaults
; Loads values into the widgets for the editor
;*****************************************************************************
pro staticdata_editor_load_defaults,info,dx,byte_array

; Note that a structure element cannot be changed if it is an argument.
; Instead a temporary variable must be used.
; That's why info.user_data cannot be the third argument.

err = extract_paramdata(byte_array,info.param_names(0),staticdata,$
      model={staticdata})
if (err ne 0) then return

info.user_data = staticdata

if (info.user_data.maggain eq 0.5) then begin
    value = 0
endif else if (info.user_data.maggain eq 2.0) then begin
    value = 2
endif else begin
    value = 1
endelse
widget_control,info.widget_ids(0),set_droplist_select=value

value = float(double(info.user_data.baseline_count)/1000000.0)
widget_control,info.widget_ids(1),set_value=value

value = info.user_data.savebasedata
widget_control,info.widget_ids(2),set_droplist_select=value

end

;*****************************************************************************
;PROCEDURE: staticdata_editor_load
; Loads the widgets for the editor
; Do the same as "load_defaults", then set the UVALUE of the widgets.
;*****************************************************************************
```

```
pro staticdata_editor_load,info,dx,byte_array

staticdata_editor_load_defaults,info,dx,byte_array

; Now set the uvalue so comparisons can be made to
; see if anything has changed.
widget_ids = info.widget_ids
for i=0,n_elements(widget_ids)-1 do begin
    widget_type = widget_info(widget_ids(i),/name)
    if (widget_type eq "DROPLIST") then begin
        value = widget_info(widget_ids(i),/droplist_select)
    endif else begin
        widget_control,widget_ids(i),get_value=value
    endelse
    widget_control,widget_ids(i),set_uvalue=value
endfor

end

;******************************************************************************
;PROCEDURE: staticdata_editor_update
; Save the parameter data by putting new values into the byte_array
;******************************************************************************
pro staticdata_editor_update,info,dx,byte_array,errorback

value = widget_info(info.widget_ids(0),/droplist_select)
if (value eq 0) then begin
    info.user_data.maggain = 0.5
endif else if (value eq 2) then begin
    info.user_data.maggain = 2.0
endif else begin
    info.user_data.maggain = 1.0
endelse

widget_control,info.widget_ids(1),get_value=value
info.user_data.baseline_count = nint(double(value*1000000.0))

value = widget_info(info.widget_ids(2),/droplist_select)
info.user_data.savebasedata = value

errorback = replace_paramdata(byte_array,info.param_names(0),info.user_data)
```

```
return
end
```

2.11.8.8 display_init procedure

The arguments to the main display library procesure "display_init" are the following.

- `dx`: the input structure argument to the editor (Sec. 2.11.8.1).
- `editor_name`: a string with the main procedure name. This string is used as a prefix for the names of the "hook" procedurers.
- `data_item_name`: always `waveform.name` which is the name of the static data item.
- `err`: an output integer value giving the return error code.
- `info`: an output IDL structure which contains all kinds of information needed by the library to perform its tasks. This structure is anonymous and can be appended to by the editor if other information is needed. This structure is passed around to all procedures and is also the "uvalue" of the base widget. Widget identifiers, initial values of data, etc. can be easily appended so that this information is available in all procedures.

Keywords that can be used in the `display_init` call include

- `names`: the name(s) of the parameter data blocks needed by the editor. Those blocks that can be updated should always come first. This keyword is usually given in order to reduce the amount of parameter data that needs to be retrieved from the waveform server. If not given, then ALL parameter data in the phase will be retrieved.
- `groups`: the name(s) of the parameter data groups. A group name can be used to match ALL parameter data blocks whose names start with the group name. This keyword can be used if the editor does not know all the names of the parameter data block needed. If used, however, the names that are retrieved from the waveform server may need to be sorted so that those that can be updated by the editor appear first in the array of names (the names are given in info.param_names).
- `save`: an integer giving the number of parameter data blocks to actually send back to the waveform server when changes are "applied". The display library will always send the first `save` blocks given by the first `save` names. This keyword helps diminish the size of the messages sent back and forth.
- `noload`: use this keyword if the editor displays information only. In this case, widgets may be filled in the editor's "init" procedure rather than using a "load" procedure.

- `nodefaults`: use this keyword if the editor has no use for the "load_defaults" button on the editor. The "load_defaults" button is used to retrieve the default value of the parameter data from the waveform server.
- `menubar`: use this keyword to add a menubar to the base editor window.
- `read_only`: use this keyword if no changes can be made by the editor, that is, the information displayed is read only.
- `destroy`: use this keyword if the editor should be destroyed when it is closed because the number or size of widgets depends on the data item that is selected. Usually, it is not necessary to destroy the window and it can be simply unmapped, the default.
- `title`: use this keyword to specify a title for the base editor window. The default title is the data item name followed by " Editor".
- `event_handler`: use this keyword to specify the name of the "event" procedure which is invoked when widgets are used. The default procedure is "display_event" which can handle most all widget events in a structured way.
- `check_if_changed_proc`: use this keyword to specify the name of the procedure which will check if any changes have been made. If so, the "apply" and "cancel" buttons become sensitive. The default is "display_check_for_changes" which can handle most all types of checking in a structured way.

(Remainder of this section has yet to be written.)

2.11.8.9 Obtaining the parameter data

Normally, the `data_item.name` argument to the editor procedure gives the name of the group of parameter data blocks that holds the data that will be used by the editor. This group name is the first set of characters in the names of each of the parameter data blocks in the group. The editor would request this group of blocks from the waveform server by specifying the group name. There could also be some required parameter data blocks that are not part of the `data_item` group which would be requested by specifying their full block name. In general, the editor can request any parameter data block from the waveform server.

The routine to use to obtain the parameter data from the waveform server is `param_getdata` (Sec. 7.6.4). This routine allows the required parameter data blocks to be specified by either their group name and/or the complete block name. In the case where a group name is specified, all blocks with names that begin with the characters in the group name are returned.

The procedure `param_getdata` returns an array of bytes that contains all of the requested blocks. A copy of this array should be maintained to represent the "original" data to which the editor returns when the user presses the "cancel" button (Sec. 2.11.8.10). When the user presses the "apply" button (Sec. 2.11.8.10) to send changes to the waveform server (Sec. 2.11.8.11), the new data sent to the waveform server become this "original" copy.

To extract an individual parameter data block from the parameter data byte array, use the routine `extract_paramdata` (Sec. 7.6.2). This routine is designed to return the data in an IDL variable that is structured as required for use by the static data editor code.

After obtaining the parameter data array of bytes from the waveform server and extracting into IDL variables the data from the relevant parameter data blocks, the data editor code is ready to make the data available in the widget interface for editing by the user. As the user makes changes in the data, the editor would normally maintain the current copy of the modified data in IDL variables. Alternatively, each time a parameter data block is modified, the data for that block in a copy of the byte array obtained from the waveform server can be updated using the procedure `replace_paramdata` (Sec. 7.6.5).

One subtlety to note about obtaining data from the waveform server using the `param_getdata` procedure is the source of the data within the waveform server. There are two choices, the input/output handler's copy and the queue handler's copy.

The input/output handler holds the current copy of the raw data provided by the user. Data from this copy are returned immediately when requested. Because the static data editor normally deals only with raw data, this is usually the source of data that a static data editor would use and this is the default for the procedure `param_getdata`.

The queue handler is responsible for taking new "raw" data provided by the user interface and calling the appropriate routines to create the "processed" data required by the algorithm. Each message from a user interface is placed into the queue and the messages are processed sequentially. If a static data editor requires access to processed data, it must request the data from the queue handler. In this way the access to the data is synchronized with the creation of the data. A request for parameter data from the queue handler's copy is placed in the processing queue and serviced when the queue routines reach that message. The data in the specified parameter data blocks that are available at that point in time are returned. Because the data are not returned until the request message is taken from the queue and processed, there could be a noticeable delay in accessing data from the queue handler's copy. Also, because the processed data could change as the various raw data are modified (possibly by multiple users editing data), the editor user interface could require an "update" button that the user can use to refresh the display with the most recent data.

2.11.8.10 Editor interface standards

The designer of the data editor widget interface is free to arrange the interface as desired for most efficient use by the user. There are, however, a few conventions to follow to make the behavior of the interface similar to that of other portions of the user interface.

1. The data on the display visible to the user should be the data that is used by the interface code. For instance, the data in a text widget where the user types some input should be copied by the code when the user pushes an action button such as "apply."

The fact that the user entered the data in the text widget should be enough for the code to find the new data. There shouldn't be some hidden requirement, such as typing a carriage return within the text widget, for the data to become known to the code.

2. There is a standard set of widgets that control the transmission of the data to the waveform server.

   - There is an "apply" button. When this button is pressed, the current set of data displayed by the editor is sent to the waveform server. The "apply" button is only sensitive if the user has made changes. In this case the data shown by the editor are either different from the data originally obtained from the waveform server when the editor window was created, or, if the "apply" button has already been pressed since the editor window was created, different from the data sent to the waveform server the last time the "apply" button was pressed.
   - There is a "cancel" button. When this button is pressed, the current set of data displayed by the editor is replaced by the set of data originally obtained from the waveform server when the editor window was created, or, if the "apply" button has already been pressed since the editor window was created, the data sent to the waveform server the last time the "apply" button was pressed. Thus, this cancels any changes made by the user. This requires that the editor retained a copy of the original data. Like the "apply" button, this button is only sensitive after the user has made changes in the data. Thus, after the cancel button is pressed,it would be desensitized.
   - There is a label widget that either says "changed" or "unchanged" to indicate whether any changes in the data have been made by the user.
   - There is a "close" button that, when pressed, causes the editor window to disappear. If the current set of data displayed by the editor is different from the original set of data (that is, the label widget says "changed"), a popup window should appear to ask the user whether the new data should be "applied" or "canceled."
   - When the user presses the "apply" button, the current set of data displayed by the data editor is sent to the waveform server and becomes the "original" set referred to above.

A standard set of routines to create these widgets and implement this behavior is being written (and a figure showing an example of the widgets will be placed here).

So, the way a user normally interacts with the editor is to make changes in the data and then send the new data to the waveform server with the "apply" button. At this point more changes can be made and the "apply" button pressed again. After making changes and before pressing the "apply" button, the user can remove the changes by pressing the "cancel" button. When the user is done with editing and doesn't want the editor window present anymore, the “close” button is pressed. If the user has made changes and forgets to apply them before closing the window, the changes are protected from loss by the popup window which prompts the user to either save or discard the changes.

2.11.8.11 Returning the updated parameter data

When the user presses the “apply” button (Sec. 2.11.8.10), the new parameter data are sent to the waveform server.

First, the static data editor would take the new data from the IDL variables where it is stored and store the data into the copy of the array of bytes obtained from the waveform server that was retained as the “original” (Sec. 2.11.8.9). The procedure that accomplishes this is `replace_paramdata` (Sec. 7.6.5). The new data are stored into the appropriate parameter data blocks by specifying the block names. For each block that is written, the input IDL variable should have the same structure that is desired for the parameter data block. That is, when the editor extracts a parameter data block using `extract_paramdata` (Sec. 7.6.2), the data are received in an IDL variable with the same data structure as the parameter block. Normally, this same variable structure would be used to hold the data modified by the user so that when the parameter block is written with the data from the IDL variable the parameter data block structure will not change (except, perhaps, the size of the array in the case where the parameter block holds an array of structures or arrays).

After the array of bytes is prepared with all of the new data, it is sent to the waveform server. However, the static data editor must send only those parameter data blocks that have actually been altered. This helps reduce the chance that a parameter data block that has been altered by another user since the editor’s local copy was obtained will not be returned to the waveform server, thus overwriting the changes made by the other user.

The routine to use to send the parameter data to the waveform server is `alterparamdata` (Sec. 7.6.1). This routine takes as input the array of bytes containing the parameter data, a list of names of parameter data blocks that should be sent to the waveform server, and the address of the waveform server.

Remember, under no circumstances should an unaltered parameter data block be sent back to the waveform server.

2.11.8.12 Adding the editor code to the user interface

Once the IDL code is written for the static data editor, it is necessary to make the code known to the “make” procedure for the PCS. To do this simply add the names of the IDL source files to the PCS installation-specific file called `compileparamidl.pro`. This file is used to compile all installation-specific IDL routines that must be included in the user interface.

The file `compileparamidl.pro` has several sections. At the top is the place to put the names of IDL source files that make IDL functions. In the next section is the place to put the names of IDL source files that make IDL procedures.

2.11.8.13 generic editor

A standard editor found in the infrastructure is the `generic_editor`. This standard editor can display individual values in parameter data that is an int array, a float array or a structure. There must also be a parameter data block which contains labels, one label per value.

By default, the generic editor creates one `text` widget per value with the corresponding label to the left of the box. This widget allows the value to be edited. Other types of widgets can be used also. These widgets require an integer value and one or more "sublabels" which interpret the value.

Keywords are specified inside the label that corresponds to the value. Keywords are followed by a colon and the value for the keyword. For example, the keyword `TYPE` can have the values `BUTTON`, `LIST`, `DROPLIST`, or `TEXT`. Some of these keywords require more information, like `DROPLIST` which requires sublabels. For example, the following label

```
"TYPE:DROPLIST|TITLE:option|first choice|second choice|third choice"
```

would create a droplist with the given title to the left. There would be three possible choices in the droplist. The value would then be set to 0, 1 or 2 corresponding to the three choices. An additional keyword `BASE:n` allows the value to be shifted by n in case the range of values needs to be different. Notice that keywords and sublabels are separated by a vertical bar.

Other examples:

```
"BASE:1|TYPE:BUTTON|use option"                    ;nonexclusive button
"TYPE:BUTTON|first choice|second choice|third choice" ;exclusive button
"TITLE:option|first choice|second choice|third choice";droplist
"TYPE:LIST|TITLE:Choose option|option 1|option 2"     ;list
```

Some notes about these examples:

- Buttons are nonexclusive (button is in or out) if only one sublabel is given. If more than one sublabel is given, then the button is exclusive (the user has to choose one of the choices).
- The `BASE:1` keyword means the value of the button is 1 or 2 rather than the default of 0 or 1.
- A droplist is the default if no `TYPE` keyword is given and there is more than one sublabel.
- A title can be pecified with the `TITLE` keyword. The title is placed above a `LIST` widget, to the left of a `DROPLIST` widget if it ends in a colon and to the right if it does not, or above an `EXCLUSIVE BUTTON` group.

See the sections on the functions `putblock_labeled_int_array` (Sec. 14.3.9), `putblock_labeled_float` (Sec. 14.3.10), and `putblock_labeled_structure` (Sec. 14.3.11) for more information about creating the parameter data and adding a data item into the user interface.

2.11.8.14 matrix editor

A standard editor found in the infrastructure is the `matrix_editor`. This standard editor displays a matrix. The editor allows values in the matrix to be changed. Several values in a row or column or a group of rows or columns can also be changed together. The same values in more than one matrix can also be changed. Matrix values can be manipulated by adding or subtracting a constant, or matrix values can be multiplied or divided by a scale factor. A new matrix can be added and an existing matrix can be deleted.

See the section on the function `putblock_labeled_matrices` (Sec. 14.3.12) for more information about creating the matrix parameter data and adding a data item into the user interface.

2.11.9 Parameter data examples

For examples of routines to initialize and compute parameter data see the file `isodnull_master.h`.

2.11.10 Scratch parameter blocks

An algorithm can provide for any number of "scratch memory parameter data blocks." Scratch memory is simply a block of memory of a size specified by an algorithm that is initialized to contain zeros at the beginning of the shot. Thus, a scratch parameter data block is a parameter data block initialized to zeros.

Scratch memory has many uses. Here are two examples.

1. Storage for data generated in real time. An algorithm may have need for space to store temporary computation results, for instance, that must be preserved between calls to the algorithm's real time code.

   Scratch memory should be used instead of allocating variables of type `static` in the algorithm's real time C code, especially if the storage space required for `static` variables is large. Because of the way the memory of the real time CPU is organized, the amount of memory available for `static` variables is limited so the use of scratch memory helps ensure proper operation of the PCS.

   Also, static variables are shared between all phases using the same algorithm. This can lead to confusion if two phases using the same algorithm are programmed to behave differently. Each shot phase has its own set of parameter data blocks, so each phase has separate scratch parameter data blocks also.

2. Storage for preloaded data. It may be convenient to have a preshot initialization routine load data for an algorithm rather than having the data loaded by the waveform server and then transferred to the real time processor before each shot. This is particularly useful when the data are fixed values, not requiring modification or examination by the operator, and when the amount of data is large.

A scratch parameter data block is created by the same routines in the waveform server as the routines used to create a parameter data block that is initialized with data (Sec. 2.11.4.2). There are two differences, though, in the arguments for the function call.

1. The "options" argument (Sec. 2.11.4.4) to the parameter data block creation function needs to have the `SCRATCH` keyword to indicate that a scratch parameter data block is being created.

2. Because no data need to be provided to fill the block, the argument to the function that creates the parameter data block (Sec. 2.11.4.2) that usually points to the data is specified as a null pointer.

Here is an example function call to create a scratch parameter data block. In this example, the scratch block is located on virtual CPU A. The block can be found on the real time CPU by using the block order index `BA_MYALGORITHM_SCRATCH_AREA` (Sec. 2.11.7) and the block will be used to contain a structure of type `my_scratch_block_structure` (`D_my_scratch_block_structure` is the descriptor (Sec. 2.11.4.5 for this structure).

```
putblock(phase,
         "my scratch area",
         "RTCPUS=A|SCRATCH",
         BA_MYALGORITHM_SCRATCH_AREA,
         D_my_scratch_block_structure,
         NULL);
```

If the scratch parameter data block has a fixed size, then it would normally be created by the `alg_parameters` routine. If the size is variable, depending on some data provided by the operator, the scratch parameter data block would be created by the `alg_vectors` routine.

2.12 Standard static data types

Static data items (Sec. 2.10.1.2) are data items which access parameter data. They are defined in a similar way as waveforms (Sec. 2.10.1.1).

Sets of support routines are available to provide the algorithm author with easy access to several generic types of static data. These generic static data types are created using the block parameter data facilities.

For each generic data type the following support is provided.

- A routine for use in the waveform server to create the parameter data blocks required with the initial set of data (see the functions listed below).

- A routine for use in the user interface that provides the capability to edit the data (see Sec. 2.11.8.13).

- The automatic restoring of the data block even if the author changes the number of values or its size.

A generic data type makes use of several "associated" parameter data blocks (Sec. 2.11.4.3) to hold the required information for the user interface routine in addition to the actual data.

See the following subsections under `parameter data`: (Sec. 2.11)

- `putblock_labeled_int_array` (Sec. 14.3.9) creates blocks associated with a `labeled int array`. Use macro `SD_LABELED_INT_ARRAY` for the `description` string and `generic_editor` for the `xunits` string in the waveform structure.
- `putblock_labeled_float_array` (Sec. 14.3.10) creates blocks associated with a `labeled float array`. Use macro `SD_LABELED_FLOAT_ARRAY` for the `description` string and `generic_editor` for the `xunits` string in the waveform structure.
- `putblock_labeled_structure` (Sec. 14.3.11) creates blocks associated with a `labeled structure`. Use macro `SD_LABELED_STRUCTURE` for the `description` string and `generic_editor` for the `xunits` string in the waveform structure.
- `putblock_labeled_matrices` (Sec. 14.3.12) creates blocks associated with `labeled matrices`. Use macro `SD_LABELED_MATRICES` for the `description` string and `matrix_editor` for the `xunits` string in the waveform structure.

2.12.1 Automatic restoration of standard static data types

Use of "putblock_labeled_" functions to create parameter data blocks associated with a static data item allow for the automatic restoration of that static data item. This is possible because, in addition to a parameter data block being created with the data, another parameter data block is created with labels that match up to the values in the data. If new values are added to the data block, there will be new labels.

During a restore, a standard static data item is processed separately and differently than parameter data blocks that are not part of a standard static data item. In the latter case, parameter data blocks are usually overwritten with the data in the archive. But, this would not be the desired effect if a block has been changed in any way. Special handling is required to make sure the existing block retains its current structure.

For standard static data items, the special handling is automatic. Instead of overwriting the existing block in the waveform server with the data found in the archive, a new block is created with the same name and "_a" appended to the end of the name. Similarly, a new block is created with the labels found in the archive. Then the older labels from the archive and the current labels are matched up and data values from the archive are moved one at a time to the existing data block in the waveform server.

If there are values in the existing block that do not exist in the archive, then those values are set to their "substitute" values or to zero if no substitute values were provided during the creation of the labeled blocks. This ensures that the new values are set to something that disables any new features added since the old discharge was run.

If the data block does not exist in the archive at all, then the existing data block in the waveform server is initialized to the substitute values, which usually are the same as the initial values used when the data block is created.

2.13 Changing default attributes of waveforms - special cases

A data item descriptor (Sec. 4.7) has a field called "yunits" (Sec. 4.7) which describes the Y axis label and optionally, any labels associated with Y grid levels that should be used on a waveform plot in the user interface. If the waveform is not of the "gridded" type, then the entire string is used for the Y axis label. If the waveform is a "gridded" type, then the first 20 characters are used for the Y axis label and the remainder of the string is used to define the labels on the grid levels.

In the simplest case, the remainder of the string is one or more 10 character labels, one for each grid level. For example, a string like

```
"Off       " "On        "
```

describes two grid level labels.

One special case involves the use of the keyword "/LabelsBlock" which allows the Y axis grid levels to be determined from a parameter data block. For example, a "Which Matrix" waveform could specify that the names of matrices be used as the grid level labels and the count of matrices be used as the number of grid levels. Two more labels could be added, "None" and "All" to allow all possibilities. So the "yunits" string could be the following:

```
"None      " "/LabelsBlock/A matrices : count/A matrices :names/" "All       "
```

If matrix names are changed or a matrix is added or deleted, the "Which Matrix" waveform display would indicate the changes.

Another special case is the labeling of a phase sequence. Since a category can have any number of phases, it is nice if the display of the phase sequence waveform would show all possible phases. This is accomplished by setting the "yunits" string to the following:

```
"/LabelsBlock/phase sequence/Discharge Shape/"
```

Yet another special case is that of an "indexed" waveform where a waveform can be used for multiple purposes depending on the value of an index. For example, a target waveform might be labeled "Voltage" in one case and "Current" in another case. The Y label to use would depend on the value of a parameter data block. The parameter data block must be defined using the structure "indexed_waveform" which looks like the following.

```
struct indexed_waveform {
                  char ylabel[64];
                  char description[64];
                  float yscales[2];
                  float ymin;
                  float ymax;
                 };
```

Each structure contains the following elements.

- `ylabel`: The Y label to use for the waveform.
- `description`: The description that follows the waveform name on the plot.
- `yscales`: A two element array giving the ymin and ymax for the plot.
- `ymin`: The absolute value of ymin allowed.
- `ymax`: The absolute value of ymax allowed.

Each element replaces the corresponding element in the waveform structure (Sec. 4.7). The parameter data block would be an array of structures. Each element gives the information to use for the plot. Another parameter data block then specifies the index into this array to use. So the yunits string has the following syntax.

```
"/LabelsBlock/index block[|category name or id[|phase name]]/indexed_waveform block"
```

```
For example,
```

```
"/LabelsBlock/Icoil usage|System[|phase name]/n1rwm replacements"
```

Here, the parameter data block that specifies the index is found in the System category (the vertical bar separates the name of the block from the category name or identifier). An optional phase can also be given. The slashes separate the keyword "LabelsBlock" and the names of the two parameter data blocks. So the value of the single integer in the "Icoil usage" block is used as the index into the array of structures block "n1rwm replacements". If the range of the index is 0-2, then the structure array has to be of size three.

Change this example slightly to allow the index to be an array.

```
"/LabelsBlock/Icoil usage[4]|System[|phase name]/n1rwm replacements"
```

Here, the first parameter data block is an array and the value of index 4 of the array would be used for the index into the array of structures. Other waveforms could use indices zero through three. By default, the first element is used if no index in brackets is given.

2.14 Communicating with the user

At various points in the execution of the code written for a control algorithm, it may be necessary to provide some type of information to the control system user. Typically, communication with the user would be necessary to provide notification of some type of problem. The following are the ways that the algorithm author can communicate with the user.

- When the user sends data to the waveform server from the user interface (typically by pushing an "apply" button) a reason to inform the user of something might occur immediately. An example is the popup message that informs the user that the PCS is locked out so that the data change will be delayed until after the shot in progress completes. This type of message is generated by the PCS infrastructure code. There are no options for the algorithm author to generate this type of immediate message.
- When the `alg_vectors` routine is run as a response to the receipt of new raw data for the algorithm, the algorithm code might detect some problem with the new raw data or a conflict between several individual pieces of raw data. For instance, if a waveform selects for use a matrix that hasn't been specified (i.e. matrix number 4 is selected from a group of 3 matrices). This type of "raw data problem" must be resolved by the operator by changing some data. The algorithm author places an entry on the "data problem list." See Sec. 2.14.1. This facility is used only in waveform server code.
- The PCS message log facility can collect messages from any of the PCS processes and display them in a message logging utility. This is the way to send messages to the operator from processes other than the waveform server. See Sec. 2.14.2. These messages are also collected in log files.
- Anything printed to `stderr` is added to the log file for the process where the output is generated. See Sec. 2.14.3.

2.14.1 Raw data problem list

When the `alg_vectors` routine is run as a response to the receipt of new raw data for the algorithm, the algorithm code might detect some problem with the new raw data or a conflict between several individual pieces of raw data. For instance, if a waveform selects for use a matrix that hasn't been specified (i.e. matrix number 4 is selected from a group of 3 matrices). In fact, if it is possible for the user to cause a problem by incorrectly specifying raw data, the algorithm author should always test for the problem and alert the user.

This type of "raw data problem" must be resolved by the operator by changing some data. The `alg_vectors` code must simply avoid errors that would be caused by the bad data (such as a divide by 0), and notify the user that a problem exists that must be corrected.

The way the user is notified is by putting an entry on the "data problem list." This is done by calling the routine `raw_data_problem` (Sec. 14.5.1). This allows the algorithm author to either add, append, or remove a problem entry from the list. These functions are described in this section. The PCS operator can view the data problem list to determine what needs fixing. Each entry on the list describes a specific problem.

Each data problem list entry has the following attributes.

- A short string giving the "type" of problem. This string is used to identify the entry in the list to the user and to software.
- A location for the problem. The shot phase where the problem is located is recorded.
- A severity level. This is either "warning" which indicates an informational message, or "fatal" which indicates a problem that would cause the discharge to fail. The PCS will not start a new discharge if there are any fatal problems on the data problem list.
- A string giving the generic "description" of the problem. This string can contain newline characters and should describe the problem sufficiently well for the operator to locate it.
- Two items that together specify the "instance" of the problem. The instance string gives more specific information about the problem than is provided by the description string. The first item is a format string for the `sprintf` function and the next is a variable length list of arguments matching the requirements for arguments of the format string.

The raw data problem list can be viewed by the operator using the user interface. Here is an example of the entry created by a call to `raw_data_problem`.

```
--------ignored vertex summary                    --------WARNING
CATEGORY   = Discharge Shape
PHASE      = ShotStart
ALGORITHM  = Isoflux Single Null Divertor
DESCRIPTION:
Summary of isoflux algorithm ignored vertices:
2.000000 sec.
2.300000 sec.
2.400000 sec.
```

In this example the problem is located in the "Discharge Shape" category in the phase called "ShotStart" which uses the "Isoflux Single Null Divertor" algorithm. The "type" identifier of the problem is "ignored vertex summary" which informs the user that some vertices on some waveforms are being ignored because of some type of problem with the vertex values. In this case this is just a summary message and additional data problem list entries are also present (not shown) giving more information about why the vertices are being ignored. In this case there are 3 instances of the problem. The description of the problem is given as

"Summary of isoflux algorithm ignored vertices:" and the list at the end of the entry gives the instances of the problem. Each instance was added to this entry by "appending" to the problem list entry.

There are three types of modifications that the `raw_data_problem` routine can make to the data problem list.

1. `REMOVE_RAW_DATA_PROBLEM`: the problem identified by "type" and located in the specified shot phase is removed from the list if it is there.
2. `ADD_RAW_DATA_PROBLEM`: the problem identified by "type" is added to the list if it is not already there. If the problem is already on the list, nothing is done. The text string used to describe the problem is the description string followed by the instance string.
3. `APPEND_RAW_DATA_PROBLEM`: the problem identified by "type" is added to the list if it is not already there. If the problem is already on the list, the description string is ignored and the instance string is appended to the string that is already on the problem list. This can be used to collect information about multiple instances of the same problem.

There are two primary ways the routine `raw_data_problem` is intended to be used.

1. In the case where there is a simple test for the existence of a problem, the problem is either added to the list or cleared from the list. For example,

   ```
   if(problem_exists)
      raw_data_problem(ADD_RAW_DATA_PROBLEM,...)
   else
      raw_data_problem(REMOVE_RAW_DATA_PROBLEM,...)
   ```

   In this case, the "description" string is used to provide an explanation of the problem to the user and the "instance" string can either be empty or can provide some extra information, typically involving some variables that must be encoded into the string. Or, the description string can be empty and all explanation can be in the instance string.

   If the problem just appeared, it is added to the list. If the problem existed previously and has now cleared up, the problem entry will be removed from the list. If the problem was already on the list, and the problem still exists, nothing is done and the problem remains on the list. If the problem didn't exist previously so that it isn't presently on the list, removing the problem from the list does nothing.
2. In the case where each item in a list of values is checked for the same problem, the problem might exist multiple times. In this case the "instance" string is intended to be used to create a list of places where the problem exists.

Here it is necessary to clear out the problem list first and then recreate the problem list entry completely.

For example,

```
raw_data_problem(REMOVE_RAW_DATA_PROBLEM,...)
for(i=0,i<Maximum,i++){
  if(problem[i] exists)
     raw_data_problem(APPEND_RAW_DATA_PROBLEM,..., "Problems exist here:\n",
                       "index %d\n", i)
}
```

2.14.2 Log messages

The PCS message logging facility (the message server) is the mechanism provided to communicate with the PCS operator from any process that is part of the PCS system. This facility is provided for ”system” level messages rather than general communication with the user.

Messages sent to the message log facility can appear in the windows of the ”view log” (Sec. 7.1) utility and are also entered into log files for archival.

To send a message to the message log use the routine `msg_log` (Sec. 14.5.2) from the waveform server or host real time code. For the real time code use the real time version: `rtmsg_log` (Sec. 15.1.1) which buffers the message before, during, and after the shot until it is safe to send the messages to the message log.

A call to `msg_log` provides the following.

- A value giving the type of the message. Each message type is identifiable in the view log utility with a specific character. The message type values and corresponding flag characters are the following.
  - `NOSTAT` (no char) no state.
  - `PCSMSG` (M) standard message.
  - `PCSSTATE` (S) PCS state message.
  - `PCSTST` (T) Test; messages for use in “test” version.
  - `PCSERR` (E) PCS Error message. This will turn the status light of the view log to yellow if sent during the shot cycle.
  - `PCSPRGRS` (P) A standard message which is used to determine shot cycle progress.
  - `PCSPID` (I) PCS identification message.
- A format string for the `sprintf` function that specifies the string to be printed for the message. The message can have multiple lines.

- A variable length list of arguments for the `sprintf` function as required by the format string.

A call to `rtmsg_log` has the same arguments as `msg_log` in addition to the `rtheap` structure. Since printing cannot be done during the discharge, any real time log messages are buffered in a buffer region whose size is given by the installation specific macro `RTMSG_LOG_SIZE` which is 10000 by default. Messages are sent after all setup is complete at first lockout and again after the shot is finished. Messages of type `PCSERR` will be counted by the view log, the count displayed, and will cause the status light to turn yellow (Sec. 7.1). The current time during the shot will automatically be inserted at the beginning of the message if the messsage is buffered during the shot.

2.14.3 Debug printouts

Anything printed to `stderr` using the `fprintf` function is written to the log file and can be examined for debugging. Note that there is a separate log file for each PCS process.

Note that it isn't a good idea to print from the real time process if it is expected that the cycle times should be appropriate for actual control algorithms. Printing typically requires a significant amount of time.

Instead of using `fprintf` in real time code, use the function `rtprintf` which is the same as `printf` except that the message gets buffered. Note that the use of `rtprintf` requires some setup before the shot by calling the function `rtprintf_init` and the calling of function `rtprintf_end` at the end of the shot. The first function sets up the memory for the message buffer and the latter function prints what is put in the buffer. For more information see Sec. 15.1.2.

2.15 Using phase sequences

The time evolution of a discharge for the quantities controlled as part of a given category is specified by the order of use of the shot phases defined for the category and by the vertices specified for the waveforms in each phase. This capability to specify a sequence of phases rather than simply using a single algorithm with a single set of waveforms to specify time evolution provides extra flexibility to organize the control system.

The order of use of the shot phases is specified by the operator using a phase sequence waveform. There is only one phase sequence waveform for each category which is referenced to absolute time, that is, relative to the $t = 0$ for the discharge. This is usually called the "primary" phase sequence (this phase sequence that is referenced to absolute time is always phase sequence number 1 for the category and is the first phase sequence listed in the `PHASESEQTABLEDEF` section of the category master file).

However, a control algorithm can be programmed to force an asynchronous change to another phase sequence, usually in response to some event detected in real time. There would usually be one of these "secondary" phase sequences defined to handle each of the possible real time events. The operator would program one or more phases with the appropriate control algorithm to handle each possible event and would specify these phases on the appropriate phase sequence waveform.

2.15.1 Forcing a phase sequence change

A control algorithm that is designed to recognize and handle a particular condition or event in real time will normally have a test for the event and a conditional switch to a specific phase sequence.

```
if(condition exists){
   force a switch to a specific secondary phase sequence
}
```

The particular secondary phase sequence is provided specifically to be paired with the control algorithm. It holds the operator's specification for discharge control that will handle the event. To force a change to another phase sequence, the algorithm code simply writes the number of the new phase sequence in the correct location.

Each phase sequence for a given category is assigned an identifying "number." This number is an integer that is specified in the `PHASESEQTABLEDEF` section of the category master file which is where the phase sequences for the category are defined. Phase sequence numbers start with 1 for the first sequence in the list and increment by 1 for each additional phase sequence. When the real time code needs to specify the number of a particular phase sequence, the macros discussed in Sec. 2.16.2 should always be used.

On the real time processor, there is an array which holds, for each category, the number of the phase sequence to be used during the next shot phase tick (see Sec. 2.8 for a description of timing within the PCS). This array is at `rtheap->next_phs_seq`.

At the end of a shot phase tick, the number of the phase sequence to be used during the next phase tick is compared to the number of the phase sequence that was in use for the just-completed shot phase tick (which is stored in the array at `rtheap->current_phs_seq_number`). If the numbers differ, a switch is made to the new phase sequence. So, forcing a switch in phase sequence is simply a matter of writing the number of the desired phase sequence into the element of the `rtheap->next_phs_seq` array corresponding to the category for which a phase sequence change is desired.

In a multi-processor system, if a category uses more than one processor to execute its algorithms, the phase sequence number must be written to the `rtheap->next_phs_seq` array on each of the processors used by the category so that the phase sequence change is made on all of the category's processors. For this reason, the `rtheap->next_phs_seq` array on each processor is part of the communication vector (see Sec. 2.9 for more on the communication vector). (The first block of space in each communication vector is allocated to the `rtheap->next_phs_seq` array.) Because this array is part of the communication vector, the `rtheap->next_phs_seq` array can be written by algorithm code but must never be read by algorithm code (see Sec. 2.9). The array at `rtheap->next_phs_seq_loop` is the corresponding portion of the copy of the communication vector that can be read by algorithm code.

A request for a change in phase sequence is recognized only at the end of a phase clock tick. If the control cycle time is short enough that there is more than one control cycle within a phase clock tick, then the phase sequence change will not necessarily take place at the end of the control cycle during which the phase sequence change was requested. Instead, the phase sequence change will take place after the final control cycle in the current phase clock tick. However, because of the way the communication vector content is synchronized with the control cycle (see Sec. 2.9), if a change in phase sequence is requested during the final control cycle of a phase clock tick, the requested change in phase sequence is not recognized until the end of the following phase clock tick. So, the algorithm requesting a change in phase sequence should be written to handle the case that the control cycle within which a phase sequence change is requested will not necessarily be the last control cycle before the phase sequence change is actually made. If the control category uses only one processor and the control algorithm code forcing the phase sequence change executes on the same processor, the possible one phase clock tick delay can be avoided by writing the phase sequence pointer directly to the `rtheap->next_phs_seq_loop` array rather than to the `rtheap->next_phs_seq` array.

The element of the `rtheap->next_phs_seq` array that corresponds to a given category is specified by using the macros discussed in Sec. 2.16.1. Note that these macros give values based at 1 so that 1 must be subtracted in order to obtain the proper index into a C language array. Also note that the category index could be different on each real time CPU.

So, forcing a switch in the phase sequence for a particular category requires, in the simplest case, only a single line of code. For example, in a routine that is executing on CPU A of the `example` category, a change to the phase sequence of that category called `special case` is effected with this code.

```
rtheap->next_phs_seq[CATA_EXAMPLE - 1] = SEQ_EXAMPLE_SPECIAL_CASE;
```

If it is desirable to give the operator the ability to choose the new phase sequence from those available, the sequence number could be obtained from an integer step target vector element that is derived from a waveform.

If a control algorithm forces a change in phase sequence in order to recover from a problem with the discharge, it is possible that the problem will be resolved so that the algorithm will want to return control to the phase sequence that was in use just before the most recent phase sequence change. To provide for this, if a the phase sequence number specified is 0, the phase sequence will be changed to the sequence that was in use before the most recent phase sequence change. This is possible because a record of each phase sequence change is made in the “phase sequence change stack.”

2.16 Useful index macros

Macros specifying various indices that are determined when the waveform server is compiled are defined in the file `rtwavedefs.h`. This file is created as part of the build process for the real time code and is located in the directory in which the products of the PCS build are placed. These macros are useful for the real time code to index into arrays that list values for the various categories, real time processors and phase sequences. Note that these macros give indices based at 1 so that in order to get the 0 based index required for indexing into a C language array, 1 must be subtracted.

2.16.1 Category indices

In general, on each of the real time processors code will execute for only a subset of the categories defined for the PCS. Each real time processor holds several arrays with entries for each of the categories executing on that processor. It is often necessary to know the proper index into these arrays for a given category.

There are three sets of macros that can be used to find category indices.

1. The simplest to use macro that specifies a category index has the form `identifier_CATEGORY_INDEX`. Here `identifier` is the identifier of the category converted to uppercase. This macro specifies the index of the category on the real time CPU on which the code is executing. When the code is compiled, a set of preprocessor macros is used to determine for which virtual CPU of the category the code is being compiled and this category index macro is then set to the proper value. For example, for the category with the identifier `example`, the macro giving the index of that category is `EXAMPLE_CATEGORY_INDEX`.

2. There is a macro specifying a category index that has the form `CATvcpu_identifier`. Here, `vcpu` is the letter index specifying the virtual CPU for the category (e.g. A, B, :.), and `identifier` is the identifier for the particular category converted to uppercase. For example, for the category with the identifier `example`, the macro giving the index of that category on virtual CPU `B` of that category is `CATB_EXAMPLE`.

3. There is a macro specifying a category index that has the form `CATvcpu_identifier1_identifier2`. Here, `vcpu` is the letter index specifying the virtual CPU for the category (e.g. A, B, :.) specified by `identifier1` (where the identifier is converted to uppercase in the macro name). This portion of the macro name specifies a particular virtual CPU. `identifier2` specifies the identifier of the category for which the index is desired. For example, for the category with the identifier `example2`, the macro giving the index of that category on virtual CPU `B` of the category with the identifier `example1` is `CATB_EXAMPLE1_EXAMPLE2`.

These macros are defined in the file `rtwavedefs.h`.

There is another method that can be used to obtain a category index. The macro `case_category_offset` (Sec. 14.1.4) can be used in the `alg_vectors` routine to have an integer step target vector element loaded with the correct value of the category index.

2.16.2 Phase sequence indices

The macro specifying the index of a given phase sequence for a particular category has the form `SEQ_identifier_SequenceName`. Here, `identifier` is the identifier for the particular category, converted to uppercase, and `SequenceName` is the name of the phase sequence with any spaces in the name replaced with an underscore character and all characters converted to uppercase.

The index and name for the phase sequence are taken directly from the phase sequence table defined in the category master file. For example, for the category with the identifier `example`, the macro giving the index of the phase sequence named `example phase sequence` is `SEQ_EXAMPLE_EXAMPLE_PHASE_SEQUENCE`.

Note that on every real time processor where a particular category executes code, the list of phase sequences for that category is the same. So, this macro has no dependence on the virtual CPU number.

These macros are defined in the file `rtwavedefs.h`.

2.16.3 Processor indices

The macro specifying the physical CPU number that corresponds to a particular virtual CPU for a particular category has the form `CPUvcpu_identifier`. Here, `vcpu` is the letter index specifying the virtual CPU for the category (e.g. A, B, :.), and `identifier` is the identifier for the particular category converted to uppercase. For example, for the category with the identifier `example`, the macro giving the physical CPU number of virtual CPU `B` of that category is `CPUB_EXAMPLE`.

These macros are defined in the file `rtwavedefs.h`.

2.16.4 Algorithm indices

The macro specifying the identifying index that is assigned to a particular algorithm has the form `ALG_category_algorithm`. Here, `category` is the identifier for the particular category converted to uppercase and `algorithm` is the identifier for the particular algorithm converted to uppercase. For example, for the category with the identifier `example`, and algorithm with the identifier `myalg` the macro giving the index of that algorithm is `ALG_EXAMPLE_MYALG`.

These macros are defined in the file `rtwavedefs.h`.

2.17 Regulating the PCS setup

This section discusses methods that the algorithm author can use to ensure that the PCS operator chooses setup options that will correctly run a discharge. A variety of tools is available, ranging from methods to issue error messages to methods to establish a default setup and make certain that the default settings remain unless the operator explicitly overrides them.

Methods to communicate with the PCS user are discussed in Sec. 2.14. Of particular use to the algorithm author is the raw data problem list (Sec. 2.14.1). Code in the `alg_vectors` function (Sec. 2.10.5) can perform checks on the raw data provided by the operator and, if problems are found, place an entry on the raw data problem list. The entry can simply indicate a warning about a potential problem, or the entry can indicate a "fatal" problem which would cause a discharge to fail. If there are any fatal entries on the raw data problem list, the PCS will not start the discharge because it is already known that the discharge will fail to work properly.

The default settings for the data items in an algorithm could have two purposes.

- The default settings could establish a reasonable mode of operation for the algorithm. But, this mode of operation would normally be expected to be replaced by the PCS operator. For instance, by default each waveform has a single vertex at a Y axis value of 0.0. This setting is probably not one that would normally be chosen by the operator, so the algorithm author might provide default waveform vertices that would implement one particular control method. (Setting default vertices for a waveform is discussed in Sec. 2.10.9.)

- The default settings could establish a mode of operation that is normally required. In this case, the operator would be expected to change these settings only rarely. The algorithm author would want to ensure that the desired settings are not accidentally altered and that it is easy to return to the default settings.

The second item is the focus of this section. How to establish a "required" set of default PCS settings and the various methods to regulate changes in these settings are discussed. First, there is an introduction to how the default settings for data items are established (Sec. 2.17.1). The PCS operator is free to change the settings for any data item if the default values are not the values required for a given discharge setup. Access control (Sec. **??**), if implemented, can prevent users who have not been granted write access from making changes. But, once granted access, there is no way to prevent a PCS operator from making changes. The plan is to eventually allow a data item to be marked "read only" so that it can never be changed. The beginnings of this access control facility are discussed in Sec. 2.17.2. The remainder of this section discusses how to control changes in specific portions of the PCS setup during a restore from an old setup or a future shot setup. There are two basic methods.

1. A portion of the PCS can be marked "no restore" so that it is not altered during a normal setup restore procedure (Sec. 2.17.3). Data that hold values that, for whatever reason, are considered specific to the current shot setup, that might need to be restored from an old shot, but which would normally want to be kept unchanged during a restore procedure, would be marked "no restore."

2. Data for a static data item can be stored in an "external file" in addition to being stored with the remainder of the PCS setup (Sec. 2.17.4). During a setup restore, the static data item is restored normally, but then the data in the external file are used to overwrite the data from the old or future shot setup. In this way, for the PCS setup used for normal tokamak operations, data that reflect the current settings of hardware on the tokamak can be always kept at their current settings since the settings used previously may not be appropriate. However, the settings actually used during a shot are archived with that shot's setup. Thus, all settings used for a given shot are available for future PCS testing using setup data from that old shot.

2.17.1 Establishing default setup data

The way in which an algorithm establishes the default values for a data item is determined entirely by the algorithm author. When a shot phase is initialized to use a particular control algorithm, the `alg_vertices` function (Sec. 2.10.9) is called so that the vertices for any waveform used by the algorithm can be set. Also, the `alg_parameters` function (Sec. 2.11.4.1) is called so that the data for any static data items can be set. The data used by these functions can be hard coded in the algorithm master file or the data could be read from a file.

After the raw data for a data item have been changed by the PCS operator it is possible to return to the default settings. Each static data item editor widget should have a "load defaults" button. In addition, the waveform editor window has a "defaults" menu bar item that can be used to restore the default settings for a single waveform, a complete phase, a complete category or the complete PCS setup.

In order to return to default settings for a single data item, a special command from the user interface to the waveform server is used to load the default values for that data item. In response to this command, the waveform server creates and initializes a temporary shot phase using the same control algorithm as the phase specified by the user interface. Then, the data from this temporary phase are returned to the user interface and the temporary phase is deleted. The PCS operator then has the choice to "apply" the data that were loaded by the user interface, or "cancel" to return to the original non-default settings.

The return to default settings for an entire phase, category, or the entire PCS setup is managed completely within the waveform server upon receipt of a command from the user interface. The operator doesn't have the choice to "cancel" the changes once the command has been given.

2.17.2 The access control mask

An access control mask contains bit fields that control how a portion of the PCS is allowed to be modified. It is planned that eventually the access control mask will be used to implement the capability to control who has the permission to modify a particular piece of data. For the moment, the use of the access control mask is limited to just the "no restore" bit described in Sec. 2.17.3.

For each data item used by an algorithm, for each phase sequence waveform, for each default phase specified for a category, and for each category there is a default access control mask provided. It is planned that eventually the capability to modify the access control settings for a portion of the PCS setup will be provided. For the moment, the access control mask is always set to its default value.

See Sec. 4.7 for the location to provide the default access control mask for a data item.

2.17.3 Inhibiting restore of data

It is possible to specify that any data item, shot phase, phase sequence waveform or complete category should remain unchanged during the process of restoring data from an archived shot setup or a future shot setup. This is controlled by the "no restore" bit in the access control mask (Sec. 2.17.2).

Presently, it is only possible to specify the default setting for this bit. The default access control mask for a data item is contained in the data item descriptor as described in Sec. 4.7. Use the macro `AC_NORESTORE` to set the bit in the mask.

The procedure followed when a complete or partial shot setup is restored is described in Sec. 2.17.3.1. This procedure shows in more detail the way the `AC_NORESTORE` access control bit is used.

Note that the `AC_NORESTORE` bit is only referenced when a particular portion of the PCS setup is restored as part of a larger portion of the PCS setup. For instance, the `AC_NORESTORE` bit is checked for each data item when a complete phase is restored. However, when a single data item is restored, the `AC_NORESTORE` bit is ignored. Thus there is a way to override the setting of the `AC_NORESTORE` bit if the PCS operator specifically chooses to restore a particular portion of the PCS that would normally not be altered during the restore procedure.

There are several situations in which it is useful to ensure that a small portion of the PCS setup is not disturbed when the PCS is restored. Here are some examples.

- If the algorithm author would like to specify a set of default settings and ensure that these defaults are normally used.
- If a portion of the PCS setup reflects the current setup of the tokamak (e.g. a switch setting). In this case it is desirable to leave this portion of the setup unchanged when an old shot setup is restored since the current switch setting is relevant, not the setting that was used during the old discharge.

2.17.3.1 The setup restore procedure

The procedure followed to restore a complete or partial PCS setup is outlined here. The primary purpose for providing this outline is to highlight the usage, during the process of a PCS restore, of the `AC_NORESTORE` bit in the access control mask (Sec. 2.17.2) and the situations in which data are initialized to their default values. The data initialization is discussed further in Sec. 2.17.3.2.

When a PCS setup is restored, the input data are taken from the setup archived for a previous discharge or from a future shot setup. These are called the "source" setup in this outline. The PCS setup stored in the waveform server prior to the restore process is called the "target" in this outline. If only a partial PCS setup is being restored, then by default the target (e.g. phase or data item) is the location with the same name as the source. The PCS user interface, however, also provides the capability to specify a target that has a different name from the source. Here, target and source refer to whatever has been designated by the PCS operator on the user interface.

In this outline of the restore process, the status of the `AC_NORESTORE` bit always refers to the bit in the access control mask of the target.

The description of a "command" here refers to the actions taken when the corresponding button is clicked on the user interface.

1. Command `load all categories`: For all categories in the target setup, do the following. This command completely replaces the PCS setup existing in the waveform server with the source setup.
    (a) If the category does not exist in the source setup and the `AC_NORESTORE` bit is not set, initialize the category to its default setup.
    (b) If the category does exist in the source setup and the `AC_NORESTORE` bit is not set, then execute the `load category` command.
    (c) If the `AC_NORESTORE` bit is set for the category, then that category is not altered during the restore process.
2. Command `load category`: Do the following.
    (a) For all phases defined for the target category, if the `AC_NORESTORE` bit is clear for the phase, do the following.
        i. If the phase is not "permanent" and the phase does not exist in the source category, then delete the phase.
        ii. If the phase is permanent and the phase does not exist in the source category, then initialize the phase, setting it to use the default algorithm.
        iii. If the phase exists in the source category, execute the `load phase` command for that phase.

   (b) For all phases defined in the source category that do not yet exist in the target category, execute the `load phase` command.

   (c) Execute the `load all phase sequences` command.

3. Command `load all phase sequences`: For all phase sequences defined in the source category, if the same phase sequence exists in the target category and the `AC_NORESTORE` bit is not set, execute the `load phase sequence` command.

4. Command: `load phase sequence`: Restore the phase sequence waveform without regard to the `AC_NORESTORE` bit.

5. Command `load all phases`: For each phase defined in the source category, if the `AC_NORESTORE` bit is not set, execute the `load phase` command.

6. Command `load phase`:

   (a) If the phase to be loaded is not defined in the target category, create the new phase and initialize it for the specified algorithm. A complete phase initialization is performed without regard to the `AC_NORESTORE` bit. Then, restore all data items in the newly initialized phase that do not have the `AC_NORESTORE` bit set.

   (b) If the phase to be loaded already exists in the target category, but the algorithm is different from the algorithm in the source phase, initialize the existing phase for the specified new algorithm. A complete phase initialization is performed without regard to the `AC_NORESTORE` bit. Then, restore all data items in the newly initialized phase that do not have the `AC_NORESTORE` bit set.

   (c) If the phase to be loaded already exists in the target category and the algorithm is the same as the algorithm in the source phase, do the following.

      i. Initialize all data items in the existing phase that do not have the `AC_NORESTORE` bit set.

      ii. Restore all data items in the existing phase that do not have the `AC_NORESTORE` bit set.

7. Command `load subset`: For each data item in the source subset, if the same data item exists in the target setup and if the `AC_NORESTORE` bit is not set for the data item in the target setup, restore the data item.

8. Command `load data item`: Restore the data item without regard to the `AC_NORESTORE` bit.

2.17.3.2 Initializing data during a setup restore

The outline of the procedure for restoring a PCS setup in Sec. 2.17.3.1 identifies several cases in which a portion of the existing PCS setup is "initialized." This means that the setup data are set to their default values.

This initialization is done in order to handle the situation where the PCS implementation has been changed since the source PCS setup was archived. Examples of this situation are the following.

- A data item was added to an algorithm.
- A permanent phase was added to a category.
- A phase sequence was added to a category.
- A category was added to the PCS implementation.

These cases all involve the addition of PCS setup data to the implementation.

The issue here is: what should be done with the setup data in a portion of the PCS setup that exists in the target setup but not in the source setup? The answer implemented in the PCS infrastructure is: if the new portion of the PCS is contained within a larger portion that is being restored (e.g. a new data item within a phase that is being restored), unless prevented by the `AC_NORESTORE` bit in the access control mask, the new portion of the PCS is set to its default values.

This choice was made because the default values in the PCS setup are normally either harmless or they are set to the way the PCS is normally used, whereas the current PCS settings might perform some control that wasn't intended in the setup being restored. Consider, for example, the case where a waveform was added to the PCS that enables or disables a new type of control. The default value for the waveform disables the new control feature. If a PCS operator restores the setup for an old discharge when this new control feature didn't exist in order to reproduce that old discharge, the preference would probably be that the new control feature is turned off since it didn't exist when the old discharge was run. But, if the existing PCS setup had the new feature enabled when the restore was performed and the new waveform was not initialized, the default would be to leave the new feature enabled, with unknown consequences when the discharge was run.

The procedure for initializing a portion of the PCS setup is the following.

- Category: all phases that are not permanent are deleted. All permanent phases are initialized to their default algorithm without regard to the `AC_NORESTORE` bit. All phase sequence waveforms are set to their default values without regard to the `AC_NORESTORE` bit.
- Phase: when a phase is initialized for a particular algorithm, all data items are set to their default values. All waveforms are set to have a single vertex at Y axis value

of 0.0. Then, the `alg_parameters` and `alg_vertices` functions are called to initialize the data items.

Section 2.17.3.1 lists the case that a data item is initialized while obeying the `AC_NORESTORE` bit, meaning that a test is necessary to determine if the data item should actually be initialized. This is implemented as follows.

- The `alg_vertices` function is used to initialize the vertices for waveforms. For each waveform for which vertices are to be set, this function calls `set_vertices` (Sec. 14.6.5). The function `set_vertices` performs a test to determine whether the vertices for the specified waveform should actually be initialized, depending on the setting of the `AC_NORESTORE` bit for the waveform and the current stage of the initialization or restore procedure.

- The `alg_parameters` function that initializes the parameter data blocks for the static data items has a structure determined by the algorithm author. So, it is the algorithm author's responsibility to check, within this function, whether the initialization of each static data item should actually be performed. This is done with the `check_access_control` function (Sec. 14.6.39). Here is an example.

  ```
  if(check_access_control("my static data item",phase,AC_INITIALIZE){
  /* Initialize the static data item. */
  }
  ```

2.17.4 Parameter data block external files

In general, the PCS setup for a given shot can be restored and used again to duplicate that shot. However, some of the PCS setup data might be intended to reflect the current settings of hardware on the tokamak. Data stored with the PCS setup for an old shot indicating the state of the hardware settings when that shot was executed wouldn't usually be relevant when a new shot is executed. The new shot would want to use the settings appropriate for the tokamak hardware on the day that the new shot is executed. Examples of this type of data are calibration factors for diagnostic data that are used by the PCS and the configuration of the wiring of power supplies to poloidal field coils.

The place to store data indicating the current tokamak hardware setup is in one or more "parameter data block external files." These are files separate from the file in which the PCS setup is stored. Each file contains the data for a particular parameter data block. This separate file always contains the current content of the parameter data block and is always used by the PCS to obtain data for that block when the parameter data block content is initialized or restored.

The data in any parameter data block that is part of a static data item can be automatically stored in an external file. To have the data for a parameter data block stored in an external file when the parameter data block is created simply set the `EXTERNALDATA` keyword in the parameter data block "options" value (see Sec. 2.11.4.4). Only parameter data blocks with this option set are stored in external files. Thus, if a static data item is composed of several parameter data blocks, it wouldn't be necessary that all of these parameter data blocks be stored in external files.

When a static data item that has a parameter data block stored in an external file is restored from an old shot setup or a future shot setup, the entire static data item is first restored using the standard procedure (Sec. 2.17.3.1). Then, the external file is read and the data are used to overwrite the data in the corresponding parameter data block. Similarly, after the `alg_parameters` function is called during initialization of a phase to set the default parameter data block content, the contents of any parameter data blocks that had `EXTERNALDATA` keyword are overwritten with the current content of the corresponding external data file.

Whenever the PCS operator uses the user interface to alter the setup data for a static data item with a parameter data block that is stored in an external file, the new data are immediately written to the external file. When an algorithm is added or modified to use an external file, the external file will be created if it doesn't exist already.

The parameter data block external files used during normal tokamak operations are stored in the PCS archive directory (Sec. 2.18). The name of a parameter data block external file is the name of the parameter data block (with each space replaced by an underscore) followed by the extension `.extdata`.

If the PCS is used in one of its testing modes, hardware, software or simulation test mode, the PCS is usually started with the `runtest` or `runsa` script (Sec. 9). This is usually the case when new software is being tested. In this case, the PCS operates in "test" mode rather than the normal mode used for tokamak operations.

When the PCS is operated in test mode, it is usually not desirable to read and write parameter data block external files at all. When testing the PCS, a common procedure is to use data from an old shot in a simulation. In this case, it is usually desirable to have exactly the same setup that was used in the old shot.

However, to test the usage of external data files, the author can simply set the environment variable WRITE_EXTERNALDATA to the value of 1 and external data files will be written and read when the PCS is brought up in a test mode. But, the files are not obtained from the PCS archive directory. Rather, the files are read and written from the default directory for the waveform server. The waveform server normally has its own default directory as described in Sec. 9. The author will need to unset this environment variable and remove these files to properly clean up, else, these files will continue to be used in the future.

Occasionally, the format of the data in a parameter data block will be changed as algorithm code evolves. Normally, in this case a developer will take care to make certain that when a parameter data block with a changed format is restored from an old shot setup, the algorithm setup restore code detects the change and adapts the format. Then when the new algorithm code is used in operations mode, any existing external data file will be automatically overwritten with the new format.

2.18 PCS directories

The PCS uses various directories for special purposes. This section documents each of these directories.

- The log file directory. When the PCS executes, each of its processes creates a log file into which all output from the `msg_log` function (Sec. 2.14.2) is copied (in addition to sending the output to the message server) and to which output to `stderr` is written. The log files have names indicating the associated process and the current date (e.g. wave_20010919.log). If the PCS is executing when the date changes, a new file with the new date is automatically created.

  The conventional location for the log files for normal PCS operations is `/link/ops/log_v10`. When stand alone mode is used, the log file is normally in a subdirectory under `./runsa_dir` (see Sec. 9.1.3).

  The log file directory is specified by the variable `LOG_DIR` in the configuration files used by the PCS startup scripts (see Sec. 9). The path name for the log file directory for normal operations is defined to the PCS C code by the macro `LOGFILEDIR` in the installation-specific file `installdefs.h`.

- The core file directory. Independent of which PCS startup method is used (Sec.. 9), each of the PCS processes is executed with its default directory set to a unique directory. Thus, if a process creates a core file or any other files that are written to its default directory, it will be easy to determine which process created the files.

  In PCS normal operations, the directory specified to the `startpcs` script by the configuration variable `STARTDIR_TOP` is the top level directory for this purpose. Under this directory a new directory is created and named with the time that the `startpcs` script was executed. Under this new directory is where the directories for the individual PCS processes are placed. The PCS version information file is also placed in this directory to identify the version of the PCS executed each time `startpcs` is executed. Thus, a separate storage location is provided each time `startpcs` is executed so all core files can be collected and correlated with reports of problems from the PCS operators. The conventional location for the top level core file directory is `/link/ops/start_v10`.

  When the PCS is started with the `runtest` script (Sec. 9.3), the core files are placed in the individual PCS process directories under the directory `./runtest_dir`. When the PCS is started with the `runsa` script (Sec. 9.1.3), the core files are placed in the individual PCS process directories under the directory `./runsa_dir`.

- The archive directory. This is the directory where PCS setup files are stored. This includes all future shot setup files and the file holding the setup for the next tokamak shot (usually called `NextShot.wa10`). In addition, parameter data block "external files" (Sec. 2.17.4) and the file `future_shot.descr`, the file that contains the description string for each future shot file, are stored in the archive directory

  The conventional location for the archive directory is `/link/ops/archive_v10`.

  The path name of the "next shot" PCS setup file is provided in the PCS configuration scripts (used by the PCS startup scripts, Sec. 9) in the variable `WS_SETUP_FILE_NAME`. The path name for the archive directory is defined to the PCS C code by the macro `ARCHIVEDIR` in the installation-specific file `installdefs.h`. The path name for the archive directory is defined to the PCS IDL code by the variable `archivedir` in the installation-specific file `waveinstall.pro`.

- The directory for storage of the "s" files into which a dump of the real time processor PCS memory area is placed after each shot (Sec. 13.4). The conventional location for this directory is `/link/ops/store_v10`.

  The path name for the s file directory is defined to the PCS C code by the macro `SFILEDIR` in the installation-specific file `installdefs.h`. The path name for the s file is also coded into the installation-specific file `getrtdata.pro`. The s file path name can optionally be provided by the environment variable `SFILEDIR` when using `getrtdata.pro` to examine the s file content (Sec. 13.4).

- The directory where the PCS executables are located. The path to this directory is provided by the variable `EXE_DIR` in the configuration files used by the PCS startup scripts (see Sec. 9). The conventional location for the executable files for normal PCS operations is `/link/ops/exe_v10` which is normally a link to the production version of the PCS (see Sec. 8.6). When one of the PCS startup scripts for testing code is used, the executables are usually located in the same directory as the startup script (see Sec. 9).

  The path name for the directory containing the executables for normal operations is defined to the PCS C code by the macro `EXEDIR` in the installation-specific file `installdefs.h`.

- The directory where the PCS looks for data. The path to this directory is not provided for in the configuration files used by the PCS startup scripts (see Sec. 9). Instead, this directory is the same for all instances of the PCS. The conventional location for data files is `/link/ops/data_v10` which could be a link to the installation-specific path.

  This directory would contain any files that are needed by installation-specific algorithms. This includes files specific to the rtefit algorithm, isoflux algorithms, and other algorithms. The operations version of the PCS also writes and reads files like the following:

– user_info.txt which contains information about users logged into the PCS.
– history.txt which contains a history of changes made to the PCS.
– access_control.data which contains default settings for access control.
– command files with the extension of .com which have one or more commands that make it easy to do something like turn gas on and off or restore a shot including those items that do not normally get restored.
– calib.dat which could contain default calibration settings.

The path name for the directory containing data files is defined to the PCS C code by the macro `DATA_DIR` in the installation-specific file `installdefs.h`.

Chapter 3

Algorithm Specification and Documentation

This section describes the standard way to write a specification for a control system algorithm. This specification will also double as the algorithm documentation in the installed code. The specification can be used by you in writing your own C-code implementation of the algorithm or it could be given to someone else to implement the code for you. In any case, it is useful to provide as complete a specification as possible to facilitate implementation.

The algorithm specification/documentation should consist of the following:

1. A brief algorithm summary giving a general description of what the algorithm controls and how it does it.
2. List the input data used by the algorithm, including any special conditioning of the data that is assumed.
3. Specify the user interface for the algorithm. List all waveforms by name and their characteristics (continuous, step, Y value range limits, default plot scales etc.).
4. Specify the algorithm name and identifier.
5. List the target vector values that will be used by the real time code. Specify how each target vector is generated from the input data (waveforms etc.).
6. Describe the structure and values of any parameter data used by the real time code. For example, you may wish to use a matrix multiplication to obtain several values simultaneously from several input diagnostics. The description of parameter data should include the matrix dimensions as well as the value of each entry or a description of how the matrix elements will be calculated.
7. Specify the user interface for the parameter data including how it should look and behave.

8. Give a detailed description of each calculation performed by the algorithm. This can be done using pseudo-code if you desire. However, if explicit C code is not provided, the pseudo-code must be specific enough so that the required C code can be written. For example,

$$error(gaptop) = shape(gaptop) - target(gaptop)$$
$$shape(gaptop) = shapefunction(data_1, data_2, \ldots)$$
$$target(gaptop) = targetfunction(waveform_1, waveform_2, \ldots)$$

   where an explicit description of the functions $shapefunction$ and $targetfunction$ should be given. The diagnostics $data_1, data_2, \ldots$ should be identified by pointname. The waveforms and their names $waveform_1, waveform_2, \ldots$ are determined by you, although you may wish to look at existing algorithms for examples.

9. Define restrictions which are placed on execution, if any. For example, does the algorithm require another algorithm running simultaneously to work correctly?

10. List the outputs of the algorithm. Specify what values will be placed in any of the shape, error, P or command vectors.

11. List the control actuators that will be driven. For example, power supplies, gas valves etc.

12. List the pointnames that are used to archive the target vectors and output data. Specify which values should not be archived. Note that even if a value is not archived it must be assigned an identifying name. Provide a short description of each pointname that can be used by the control system operator to remember what the pointnames are. Provide inherent number and physics zero for each value to be archived in the shot database.

13. Specify in detail how to test the algorithm. Include, for instance, shot numbers that would provide sample input data.

Chapter 4

The control algorithm code

4.1 Introduction

A control algorithm is implemented by writing an "algorithm master file." This file contains essentially all of the C language code required to implement the algorithm. The master file is, then, a single reference point for the algorithm making it relatively easy for a programmer to view all of the software components of which the algorithm is composed. It is reasonable to use "include" files to bring code into the master file so there may be other files that are used with the master file to implement the algorithm, but the use of these include files will be clearly visible in the master file. There may also be a few assembly language routines that are used by an algorithm. These routines would be located in separate files.

The name convention for the algorithm master file is `alg_master.h` where `alg` is the identifier for the algorithm. The algorithm master file is used as a C language include file as indicated by the `.h` suffix. The code contained in the master file is added to the various processes in the control system by including the master file in the source code for those processes.

The master file is composed of a number of separate code sections. Only specific sections of the master file are included in a given process. The sections of a file are identified by a C preprocessor `ifdef` statement as in the following example.

```
#ifdef CODE_SECTION_1
  /* a bunch of code here */
#endif
```

Then, in a particular process, master file code is included as in this example.

```
#define CODE_SECTION_1 1
  /* include one or more master files here */
#undef CODE_SECTION_1
```

So, in this example, only the code in the master file that is compiled when the macro CODE_SECTION_1 is defined will be compiled into the example process. There is a more detailed description of the way the algorithm master files are assembled into the complete control system in Sec. 4.16.

To add an algorithm to the PCS code, the algorithm master file is included in the file categoryalgorithms.h, where category is the identifier for the particular category to which the algorithm belongs. The "category algorithms" file is created when a control category is defined for the control system. For instance, there might be a file called demoalgorithms.h which defines the algorithms for the category demo:

```
#define A_CODE 1
#include "mydemoalg_master.h"
#undef A_CODE

#define A_CODE 2
#include "myotheralg_master.h"
#undef A_CODE
```

In this example, there are two algorithms in the category demo. The algorithms have the identifiers mydemoalg and myotheralg.

Note in the example the use of the macro A_CODE. This defines the "algorithm code number" which some of the infrastructure software uses to identify the algorithm. The macro A_CODE will appear in the algorithm master file in several places. It is defined in the category algorithms file to a value that is unique for each algorithm. There is also a corresponding "category code number." The macro for this is C_CODE. This macro will also be used in the algorithm master file in a few places.

The remainder of this section of this document will be used to discuss each section of an algorithm master file in detail. The easiest way to write a new algorithm master file is to take an existing master file and modify it using the description of the code sections given here. It is useful to have a listing of an algorithm master file available while reading the following few sections in order to compare the code and the description here.

4.2 Configuration definitions

4.3 Symbol definitions

The CONTROLDEF section provides definitions of all macros and data structures that are unique to the algorithm. To help in making the algorithm code easier to understand, the algorithm author should follow the convention of putting all definitions in this section.

Examples of definitions that should appear in this section are:

1. Definitions of macros that specify the element number of a vector in the real time data structures or the block order index of a parameter data block. Sec. 2.10.3 discusses these macros in detail.

2. Definitions of C language structures that are used by the algorithm code. An example would be a structure that defines the organization of a parameter data block for the algorithm or the organization of a scratch parameter data block.

3. Macros to generate data object descriptors for parameter data blocks.

4. Other miscellaneous macro definitions that define sizes of arrays, etc.

The code in this section is located between the preprocessor statements:

```
#ifdef CONTROLDEF
/* your code */
#endif
```

The code between these statements is included in the file `controldefs.h`. This file is included in all infrastructure source files in which code from the algorithm master file will be used. So, all definitions in this section will be available to all of the remainder of the code in the master file, no matter where the code is used within the control system processes.

It is important to note that code should never refer to a vector element number by other than a proper macro definition. Hardwired constants should never be used because the allocation of space in the various vectors to the various categories can change. These changes are automatically taken into account by using the proper macros.

Also, this definitions section should never contain code that generates storage or processor instructions. The `CONTROLDEF` section should contain only preprocessor or compiler definitions.

4.4 Fast data save code definitions

4.5 Waveform server global variables

The `WAVEGLOBALS` section of the algorithm master file contains definitions and initializations of variables that will become global variables in the waveform server process. This section of code is located as follows.

```
#ifdef WAVEGLOBALS
/* waveform server global variables defined here. */
#endif
```

Several of the variables in this section serve a standard purpose in the implementation of a control algorithm. In addition, if there is a reason that an algorithm must define global variables for its own purposes, this is the place that these variables should be defined and initialized.

The waveform server global variables that are a standard part of an algorithm are arrays of constants that provide definitions of data relevant to the algorithm. The arrays are referenced through the `categories_algorithms` structure (described in Sec. 4.6.1). These arrays are as follows.

1. Vector element information: Recall that each control category has assigned to it one or more blocks of elements in each real time vector (Secs. 2.5, 2.5.6). A given algorithm can make use of any of the vector elements that are assigned to its category. The vector element information lists which of the vector elements are used by the algorithm and gives some information about each element. Much of this information is provided for the purpose of archiving the vector element data or for diagnostics. Background information on this is in Sec. 2.7.

   The vector element information is in a single array of structures of type `all_vector_info` (there is an older method that used several arrays that is still usable, see Sec. D.6). Each structure provides information about one element in one of the following vectors: target, pointer target, error, shape, command, and function. The descriptions of error vector elements are also used for the P vector and the descriptions of the command vector elements are used for both the intcommand and fpcommand vectors. The vector element information includes the specification of the virtual CPU. The virtual CPU specification can be a compile-time macro so that an algorithm can have sections of code which can be designed to run on different real time cpus depending on the desired PCS configuration. The vector element information array of structures ends with an empty structure to indicate the end of the array.

   Each structure contains the following information (see below for some slight modifications that apply to function vector elements).

   (a) A macro indicating the vector type:
      - `T_VECTOR` used for a target vector element.
      - `CT_VECTOR` used for a continuous target vector element.
      - `IST_VECTOR` used for an integer step target vector element.
      - `FST_VECTOR` used for a floating step target vector element.
      - `PTR_VECTOR` used for a pointer target vector element.
      - `E_VECTOR` used for an error vector element.
      - `S_VECTOR` used for a shape vector element.
      - `CMD_VECTOR` used for a command vector element.
      - `FCN_VECTOR` used for a function vector element.

(b) A macro indicating the virtual cpu, e.g., CPUA (Sec. 2.9.1) where the vector element is located.

(c) A string of 1 to 10 characters giving a name to assign to the vector element. If the vector element data are archived, then the name becomes the pointname for that data. If the data are not archived, then the name should still be provided because it is used for diagnostics.

By convention, the name has the following format.

```
\ntt{AlgVectorElement}
```

`Alg` is a two character abbreviation for the algorithm. For example, `DN` for the `dnull` algorithm.

`Vector` is a single character indicating the real time vector: `T` for floating point values in the target vector (i.e. the continuous or float step portions of the target vector), `I` for integer portions of the target vector, `R` for the pointer target vector, `E` for the error vector, `S` for the shape vector and `C` for the command vector. For the P vector, the letter `E` in the error vector name is replaced by `P` (automatically by the code when the data is archived) and the remainder of the name remains the same. For the command vector, the name is used as given for the intcommand vector (indicated by the `C`) and the `C` is replaced by `F` for the fpcommand vector, which is composed of floating point values.

`Element` is up to 7 arbitrary characters giving the remainder of the name. Ideally, the name is an abbreviation for something relevant to the algorithm.

Note that the name must be unique among all data archived in the tokamak database if it is to be used for archiving the data.

(d) The macro giving the number of the vector element. See Sec. 2.10.3 for a description of these macros. The macro should have been defined in the section of the algorithm master file containing the symbol definitions (see Sec. 4.3).

(e) The archive flag. Set this value to 1 to have the data archived in a pointname automatically after each discharge (Sec. 2.7), 0 otherwise.

(f) The inherent number. If the data is archived, this value is stored in the pointname header (Sec. 2.7).

(g) The physics zero. If the data is archived, this value is stored in the pointname header (Sec. 2.7).

This same structure is also used for the function vector elements except that the "element name" string is replaced by a string giving the name of the function (see Sec. 2.5.2 for background on the function vector). A function that is written in the C language may need to have its name prefixed with an underscore on some operating systems. Also, the last three elements of the structure (archive flag, inherent number, physics zero) are ignored for the function vector and can be omitted from the structure data.

By convention the array of vector element information is named `alg_info` where `alg` is the algorithm identifier. An example of defining this array of structures follows.

```
struct all_vector_info alg_info[] =
{
        {
                T_VECTOR,         /* vector type */
                CPUA,             /* virtual cpu */
                "DNTZPP",         /* element name */
                TA_ANLGSTD_ZPP,   /* element number */
                1,                /* archive mask */
                1.0,              /* inherent number */
                0.0               /* physics zero */
        },
        {
                FCN_VECTOR,       /* vector type */
                CPUA,             /* virtual cpu */
                "dnull",          /* function name */
                FCNA_SHAPE(1),    /* element number */
        },

LAST_ELEMENT  /* specifies an empty structure */
};
```

2. Waveform subset names. This is an array of strings giving the names to be assigned to the waveform subsets used by the algorithm (see Sec. 2.10.1 for more information about waveform subsets). The `categories_algorithms` structure (Sec. 4.6) specifies the number of subsets defined by the algorithm. This array must have exactly one string for each waveform subset. The first string gives the name for subset 1, the second for subset 2 etc.

   By convention this array is named `alg_subsetnames` where `alg` is the algorithm identifier.

4.6 The algorithm descriptors

In the `CONTROL`, `HOST_CONTROL`, and `RT_CONTROL` sections of the algorithm master file several structures are initialized that give details of the definition of the algorithm. In the `CONTROL` section there is one mandatory structure that, when the code is compiled, becomes one element in an array of structures called `categories_algorithms` in the waveform server. This array is used to hold the list of categories and algorithms known to the control system. There are two optional structures, one that becomes part of the host_cpu process at compile time and one that becomes part of the real time code.

These structures are located in the master file as follows.

```
#ifdef CONTROL
{  /* brace beginning the waveform server structure. */
   /* initializers for the structure here */
}, /* Brace ending the structure.
      Note the comma, it must be present */
#endif
#ifdef HOST_CONTROL
{  /* brace beginning the host_cpu process structure. */
   /* initializers for the structure here */
}, /* Brace ending the structure.
      Note the comma, it must be present */
#endif
#ifdef RT_CONTROL
{  /* brace beginning the real time code structure. */
   /* initializers for the structure here */
}, /* Brace ending the structure.
      Note the comma, it must be present */
#endif
```

4.6.1 The waveform server structure

The algorithm descriptor in the waveform server process is a structure of type `cat_alg` defined as follows (in the infrastructure file `serverdefs.h`).

```
struct cat_alg {
                int category;
                int algorithm;
                int version;
                int subset_count;
                char **subset_names;
                char *name;
                char *identifier;
                int *function_map[MAX_VCPUS];
                char **functionnames[MAX_VCPUS];
                struct algorithm_usage alguse;
                struct category_usage catuse;
                };
```

The structure of type `category_usage` contains information that is only relevant to defining a category. In an algorithm master file, this structure is left to be filled with default values by the compiler. The structure of type `algorithm_usage` contains information that is specifically about an algorithm. This structure is defined as follows.

```
struct algorithm_usage {
                char *parampro;
                void (*vectorfunction)(int, struct shotphase *);
                void (*paramarchivefcn)(struct shotphase *,
                                        char **, int *,
                                        short **, int *,
                                        int **, int *,
                                        float **, int *,
                                        double **, int *,
                                        struct forarchive_struct *);
                void (*paramrestorefcn)(struct shotphase *,
                                        char *, int,
                                        short *, int,
                                        int *, int,
                                        float *, int,
                                        double *, int,
                                        struct fromarchive_struct *);
                void (*parameterfunction)(struct shotphase *);
                void (*paramforhostfcn)(struct shotphase *, int,
                                        char **, int *,int,
                                        struct forhost_struct *);
                void (*paramforuserfcn)(struct shotphase *,
                                        char **, int *,
                                        struct foruser_struct *);
                void (*paramfromuserfcn)(struct shotphase *,
                                         char *, int,
                                         struct fromuser_struct *);
                int (*paramsizefcn)(struct shotphase *, int);
                void (*vertexfunction)(struct shotphase *);
                int (*scratchsizefcn)(struct shotphase *, int);
                struct vector_info *target_info[MAX_VCPUS];
                struct vector_info *pointer_target_info[MAX_VCPUS];
                struct vector_info *error_info[MAX_VCPUS];
                struct vector_info *shape_info[MAX_VCPUS];
                struct vector_info *command_info[MAX_VCPUS];
                int param_targets[MAX_TARGETS_AFFECTED];
                int global_targets[MAX_TARGETS_AFFECTED];
                int unattached_targets[MAX_TARGETS_AFFECTED];
                struct all_vector_info *all_info;
                      };
```

Each element of the structures is described in the following list. Refer to an algorithm master file as an example in conjunction with the list here. The elements of the structure are described in order here. The comments in the sample master file can be used to match the description here with the source code in the example file.

- `category`: Category code number. This is always the macro `C_CODE` which is defined outside the algorithm master file to the appropriate value.
- `algorithm`: Algorithm code number. This is always the macro `A_CODE` which is defined outside the algorithm master file to the appropriate value.
- `version`: Version number of the algorithm. This is an integer that is used to track changes in the algorithm. When changes are made to the algorithm this number should be incremented. This version number is archived in the tokamak database for each shot in order to record exactly what version of the algorithm was used on each shot. This version number keeps track of changes that are compatible with previous versions of the code. Changes that would make the algorithm incompatible with the way the algorithm functioned on a previous shot should never be made. A new algorithm should be created instead.
- `subset_count`: The count of subsets into which the data items used by the algorithm are divided. The use of subsets is optional so if subsets are not used put 0 here and put 0 for the subset number for all data items.
- `subset_names`: The name of the array holding the list of subset names. This is converted by the compiler into a pointer to the array. The use of subsets is optional so if subsets are not used put 0 here and do not define a subset name array.
- `name`: A string that gives the algorithm description that appears on the user interface list of algorithms for the category. This string should be 20 to 30 characters in length.
- `identifier`: A string that gives the identifier for the algorithm. This is a short string, in lowercase by convention, that is unique for each algorithm. The PCS code uses this string to identify an algorithm. Also this string is used by convention in naming many of the files, functions and arrays associated with the algorithm.
- `function_map`: This value should be set to `{0}` and the `all_info` field in the `algorithm_usage` structure should be used instead (see Sec. 4.5 for details). Previously, this was an array

of pointers to function vector map arrays, one for each real time CPU used by this algorithm.

- `functionnames`: This value should be set to {`0`} and the `all_info` field in the `algorithm_usage` structure should be used instead (see Sec. 4.5 for details). Previously, this was an array of pointers to the function name arrays, one for each real time CPU used by the algorithm.

- `algorithm_usage`: A structure that provides information that is used only to define a control algorithm. This structure has the following elements.

  – `parampro`: An obsolete pointer to a character string. This location should always contain an empty string (`""`). (Previously, this string provided the name of an IDL procedure to be called by the user interface code to provide an X windows interface that would allow the operator to modify the parameter data for this algorithm. This has been replaced by the capability to specify an editor widget routine for each static data item.)

  – `vectorfunction`: The name of the `alg_vectors` routine. See Sec. 2.10.5 for more information on this routine.

  – `paramarchivefcn`: Put the name of the routine that handles the archive of parameter data for this algorithm here in order to generate a pointer to this routine. Normally this is `pcs_forarchive`. See Sec. 2.11.5.1 for more information on this routine. If this function is not needed for this algorithm (because the algorithm has no parameter data blocks), put `NULL` here.

  – `paramrestorefcn`: Put the name of the routine that handles the restore of parameter data for this algorithm here in order to generate a pointer to this routine. Normally this is `pcs_fromarchive`. See Sec. 2.11.5.2 for more information on this routine. If this function is not needed for this algorithm (because the algorithm has no parameter data blocks), put `NULL` here.

  – `parameterfunction`: Put the name of the routine that handles initialization of parameter data for this algorithm here in order to generate a pointer to this routine. See Sec. 2.11.4.1 for more information on this routine. If this function is not needed for this algorithm (because the algorithm has no parameter data blocks), put `NULL` here.

  – `paramforhostfcn`: Put the name of the routine that handles the provision of parameter data to the real time CPU for this algorithm here in order to generate a pointer to this routine. Normally this is `pcs_forhost`. See Sec. 2.11.5.4 for more information on this routine. If this function is not needed for this algorithm (because the algorithm has no parameter data blocks), put `NULL` here.

- `paramforuserfcn`: Put the name of the routine that handles the provision of parameter data to the user interface for this algorithm here in order to generate a pointer to this routine. Normally this is `pcs_foruser`. See Sec. 2.11.5.5 for more information on this routine. If this function is not needed for this algorithm (because the algorithm has no parameter data blocks), put `NULL` here.
- `paramfromuserfcn`: Put the name of the routine that handles receiving parameter data from the user interface for this algorithm here in order to generate a pointer to this routine. Normally this is `pcs_fromuser`. See Sec. 2.11.5.6 for more information on this routine. If this function is not needed for this algorithm (because the algorithm has no parameter data blocks), put `NULL` here.
- `paramsizefcn`: Put the name of the routine that handles the computation of the size of parameter data for this algorithm here in order to generate a pointer to this routine. Normally this is `pcs_paramsize`. See Sec. 14.3.34 for more information on this routine. If this function is not needed for this algorithm (because the algorithm has no parameter data blocks), put `NULL` here.
- `vertexfunction`: Put the name of the routine that handles the initialization of the waveform vertices here in order to generate a pointer to this routine. See Sec. 2.10.9 for more information on this routine. If this function is not needed for this algorithm (because the default values for the vertices are satisfactory for all waveforms or the algorithm has no waveforms), put `NULL` here.
- `scratchsizefcn`: This value is obsolete. Put `NULL` here. (Previously this was the name of the routine that returned the size of the real time scratch area for this algorithm. Section D.4 has the background. Now, a scratch parameter data block is used instead of the general scratch area. Section 2.11.10 has background information on scratch parameter data blocks.)
- `target_info`: This value should be set to `{0}` and the `all_info` field (see below) should be used instead (see Sec. 4.5 for details). Previously, this had to be set to the target vector element information array pointers, each of which was an array of structures, one for each real time CPU used by the algorithm (Sec. D.6).
- `pointer_target_info`: This value should be set to `{0}` and the `all_info` field (see below) should be used instead (see Sec. 4.5 for details). Previously, this had to be set to the pointer target vector element information array pointers, each of which was an array of structures, one for each real time CPU used by the algorithm (Sec. D.6).
- `error_info`: This value should be set to `{0}` and the `all_info` field (see below) should be used instead (see Sec. 4.5 for details). Previously, this had to be set to the error vector element information array pointers, each of which was an array of structures, one for each real time CPU used by the algorithm (Sec. D.6).

  - `shape_info`: This value should be set to {`0`} and the `all_info` field (see below) should be used instead (see Sec. 4.5 for details). Previously, this had to be set to the shape vector element information array pointers, each of which was an array of structures, one for each real time CPU used by the algorithm (Sec. D.6).
  - `command_info`: This value should be set to {`0`} and the `all_info` field (see below) should be used instead (see Sec. 4.5 for details). Previously, this had to be set to the command vector element information array pointers, each of which was an array of structures, one for each real time CPU used by the algorithm.
  - `param_targets`: This entry is obsolete. Put {`0`} here. (Previously, this was an array of data entry numbers for pieces of processed data that needed to be recomputed when the parameter data for this algorithm changed. Now, data entry numbers are provided in the data item descriptor for each static data item.)
  - `global_targets`: This entry is obsolete. Put {`0`} here. Previously, this was an array of data entry numbers for pieces of processed data that had to be recomputed when the global parameter data changed. Global parameter data has been replaced with external data items.
  - `unattached_targets`: An array of data entry numbers for pieces of processed data that are not associated with a piece of raw data. This processed data computation is done only when a new shot phase is created or the user changes the algorithm used by a phase. If there are no data entry numbers that belong here then put {`0`} here.
  - `all_info`: A pointer to an array of structures. Each structure in the array gives the vector element information (Sec. 4.5) for a vector element used by this algorithm. If no vector elements are used by this algorithm, then no array needs to be defined and {`0`} is placed here.

- `category_usage`: A structure that provides information that is used only to define a control category. For an algorithm master file this is an empty structure initialized by the compiler with zeros. Put {`0`} here.

4.6.2 The host process structure

This structure provides some information about the algorithm for the host process for the real time process. It is possible that none of this information is necessary for the algorithm, so this structure is optional.

An example of how to specify this structure is shown in Fig. 4.1. Note that the code specifying this structure is included in a file at compile time to initialize an array of structures. So, a comma must follow the closing brace as shown.

The elements of this structure are as follows.

```
#ifdef HOST_CONTROL
      {
       C_CODE,
       A_CODE,
       alg_host_init_function,
       alg_host_cleanup_function,
       alg_identifier,
       alg_host_calculation_function,
       alg_host_archive_function,
      },
#endif
```

Figure 4.1: The host real time process algorithm structure.

- The category code number. This is always the macro `C_CODE`.
- The algorithm code number. This is always the macro `A_CODE`.
- The name of the function that performs pre-shot initialization within the host process for the real time process on behalf of this algorithm. See Sec. 2.6 for background information on this function. This function is optional. If no function is to be specified, then substitute `0` for a function name. This causes a null pointer to be provided instead of a pointer to the function. The call format for this function is:

  ```
  int alg_host_init_function(
  struct rtshotphase *phase,
  struct hostshotphase *host_phase,
  struct rt_heap_misc *rtheap)
  ```

  The function returns the integer `0` if there were no errors in the execution of the function, and it returns a nonzero integer if there were errors. If this function indicates that there were errors, the shot is aborted. The function arguments are:

  – `phase`: Pointer to the descriptor for the shot phase that is maintained in the real time process. This is a pointer in the address space of the real time CPU, so it must be converted to a host address space pointer before being used.

  – `host_phase`: Pointer in the host machine's address space to the host real time process's descriptor for the shot phase. This descriptor points to, for instance, the parameter data for the shot phase that is located in the host real time process.

  - `rtheap`: Pointer to the real time heap structure. This is a pointer in the address space of the real time CPU, so it must be converted to a host address space pointer before being used.

- The name of the function that performs post-shot cleanup within the host process for the real time process on behalf of this algorithm. See Sec. 2.6 for background information on this function. This function is optional. If no function is to be specified, then substitute `0` for a function name. This causes a null pointer to be provided instead of a pointer to the function. The call format for this function is:

  ```
  int alg_host_cleanup_function(
  struct rtshotphase *phase,
  struct hostshotphase *host_phase,
  struct rt_heap_misc *rtheap)
  ```

  The function returns the integer `0` if there were no errors in the execution of the function, and it returns a nonzero integer if there were errors. If this function indicates that there were errors, an error message is sent through the message server. The function arguments are the same as for `alg_host_init_function`.

- This must be the same algorithm identifier string provided for the variable `identifier` in the `cat_alg` structure described in Sec. 4.6.1.

- The name of the function that performs post-shot calculations within the host process for the real time process on behalf of this algorithm. See Sec. 2.6 for background information on this function. This function is optional. If no function is to be specified, then substitute `0` for a function name. This causes a null pointer to be provided instead of a pointer to the function. The call format for this function is:

  ```
  int alg_host_calculation(
  struct rtshotphase *phase,
  struct hostshotphase *host_phase,
  struct rt_heap_misc *rtheap)
  ```

  The function returns the integer `0` if there were no errors in the execution of the function, and it returns a nonzero integer if there were errors. If this function indicates that there were errors, an error message is sent through the message server. The function arguments are the same as for `alg_host_init_function`.

- The name of the function that performs post-shot archiving within the host process for the real time process on behalf of this algorithm. This function does archiving of data that is not part of the standard archiving. This could be the archiving of data stored

```
#ifdef RT_CONTROL
      {
       C_CODE,
       A_CODE,
       alg_init_function,
       alg_cleanup_function,
       alg_beginphase,
       alg_enterphase,
      },
#endif
```

Figure 4.2: The real time CPU algorithm description structure.

in a parameter data block, for instance. This function is called *before* the standard archiving for the host real time process. See Sec. 2.6 for background information on this function. This function is optional. If no function is to be specified, then substitute `0` for a function name. This causes a null pointer to be provided instead of a pointer to the function. The call format for this function is:

```
int alg_host_archive(
struct rtshotphase *phase,
struct hostshotphase *host_phase,
struct rt_heap_misc *rtheap)
```

The function returns the integer `0` if there were no errors in the execution of the function, and it returns a nonzero integer if there were errors. If this function indicates that there were errors, an error message is sent through the message server. The function arguments are the same as for `alg_host_init_function`.

4.6.3 The real time code structure

This structure provides some information about the algorithm for the code that runs on the real time CPU. It is possible that none of this information is necessary for the algorithm, so this structure is optional.

An example of how to specify this structure is shown in Fig. 4.2. Note that the code specifying this structure is included in a file at compile time to initialize an array of structures. So, a comma must follow the closing brace as shown.

The elements of this structure are as follows.

- The category code number. This is always the macro `C_CODE`.

- The algorithm code number. This is always the macro `A_CODE`.
- The name of the function that performs pre-shot initialization on the real time CPU on behalf of this algorithm. See Sec. 2.6 for background information on this function. This function is optional. If no function is to be specified, then substitute `0` or `NULL` for a function name. This causes a null pointer to be provided instead of a pointer to the function. The call format for this function is:

  ```
  void alg_init(struct rtshotphase *phase,
        struct rt_heap_misc *rtheap)
  ```

  To follow the standard naming convention, `alg` is the algorithm identifier. The function arguments are:

  – `phase`: Pointer to the descriptor for the shot phase that is maintained in the real time process.

  – `rtheap`: Pointer to the real time heap structure.

- The name of the function that performs post-shot cleanup on the real time CPU on behalf of this algorithm. See Sec. 2.6 for background information on this function. This function is optional. If no function is to be specified, then substitute `0` or `NULL` for a function name. This causes a null pointer to be provided instead of a pointer to the function. The call format for this function is:

  ```
  void alg_cleanup(struct rtshotphase *phase,
           struct rt_heap_misc *rtheap)
  ```

  To follow the standard naming convention, `alg` is the algorithm identifier. The function arguments are:

  – `phase`: Pointer to the descriptor for the shot phase that is maintained in the real time process.

  – `rtheap`: Pointer to the real time heap structure.

- The name of the function that is executed in real time when a shot phase that uses this algorithm is entered at the beginning of the phase. See Sec. 2.6 for background information on this function. This function is optional. If no function is to be specified, then substitute `0` or `NULL` for a function name. This causes a null pointer to be provided instead of a pointer to the function. The call format for this function is:

```
void alg_beginphase(struct rtshotphase *phase,
                    struct rt_heap_misc *rtheap)
```

To follow the standard naming convention, `alg` is the algorithm identifier. The function arguments are:

- `phase`: Pointer to the descriptor for the shot phase that is maintained in the real time process.
- `rtheap`: Pointer to the real time heap structure.

At the point in the real time control cycle that this function is called, all of the work has been done to switch to the new shot phase. So, all of the data that the real time control function can use is also available to this function (e.g. parameter data, scratch memory, continuous target values, integer step target values, float step target values). The digitizer data vector contains the data acquired for the control cycle that has just finished (the last cycle executed in the previous shot phase).

- The name of the function that is executed in real time when a shot phase that uses this algorithm is entered at any time other than the beginning of the phase. See Sec. 2.6 for background information on this function. This function is optional. If no function is to be specified, then substitute `0` or `NULL` for a function name. This causes a null pointer to be provided instead of a pointer to the function. The call format for this function is:

```
void alg_enterphase(struct rtshotphase *phase,
                    struct rt_heap_misc *rtheap)
```

To follow the standard naming convention, `alg` is the algorithm identifier. The function arguments are:

- `phase`: Pointer to the descriptor for the shot phase that is maintained in the real time process.
- `rtheap`: Pointer to the real time heap structure.

At the point in the real time control cycle that this function is called, all of the work has been done to switch to the new shot phase. So, all of the data that the real time control function can use is also available to this function (e.g. parameter data, scratch memory, continuous target values, integer step target values, float step target values). The digitizer data vector contains the data acquired for the control cycle that has just finished (the last cycle executed in the previous shot phase).

4.7 Data item descriptors

The WAVEFORMS code section contains the code that initializes an array of descriptors for the data items used by the algorithm. Each descriptor specifies the characteristics of either a waveform or a static data item. Recall that the PCS operator uses a waveform to provide raw input data to the waveform server specifying how some quantity should evolve in time. A static data item provides a table of raw data that is generally fixed in time. See Sec. 2.10.1 for more background information on waveforms and static data items.

A data item descriptor is a structure of type `waveform`. This section of code initializes a list of these structures. This code becomes part of an array of structures in the waveform server that defines all of the waveforms and static data items known to the control system.

The `waveform` structure is defined as follows in the infrastructure header file `serverdefs.h`. The individual components of the structure are described in detail below.

```
struct waveform {
                 int message_length;
                 float scale[4];
                 char *name;
                 char *xunits;
                 char *yunits;
                 char *restorepoint;
                 float restorescale;
                 float restoreoffset;
                 char *archivepoint;
                 char *description;
                 int category;
                 int algorithm;
                 int targets[MAX_TARGETS_AFFECTED];
                 int step;
                 int gridded;
                 float gridpoints[MAX_WAVEFORM_GRIDPOINTS];
                 float ymin;
                 float ymax;
                 int subset;
                 int access_control;
                };
```

The `waveform` structures are located in the algorithm master file as follows.

```
#ifdef WAVEFORMS
{  /* brace that starts a waveform structure */
/* structure initializations. */
}, /* ending brace, note the comma */

/* as many other structures as required. */

{
/* the last waveform structure for this algorithm */
}, /* note that there must be a comma here also */
#endif
```

Each structure contains the following elements.

- `message_length`: The first element of a waveform structure is always set to zero by the algorithm programmer. It is used by the infrastructure code.
- `scale`: For a waveform, an array of 4 floating point values that specify the default display scales for the waveform on the standard plot on the user interface. The array specifies values in the following order:
  1. The minimum value for the X axis on the plot.
  2. The maximum value for the X axis on the plot.
  3. The minimum value for the Y axis on the plot.
  4. The maximum value for the Y axis on the plot.

  There are two standard macros, `XMIN` and `XMAX` that are often used to define the first two values. These symbols are defined as part of the PCS installation and are an easy way to change the definitions for all waveforms at once if necessary.

  If the structure is used to describe a static data item, this entry is optionally used to pass data to the static data editor used for this data item.
- `name`: A string giving the name of the waveform or static data item. The string length should be limited to 20 characters so that the name fits in the space provided in various displays.
- `xunits`: For a waveform, this entry is a string giving the label for the X axis on the user interface plot of the waveform. This is `"time (seconds)"` by convention.

  For a static data item, this entry is a string giving the name of the IDL procedure that generates the user interface editor for the data.
- `yunits`: For a waveform, this entry is a string that, in its simplest format, gives the label for the Y axis on the user interface plot of the waveform. However, this string can have specific formats that carry some information on how the Y axis should be labeled.

1. If the waveform is not of the "gridded" type (see below for entries in the waveform structure that determine whether it is a gridded waveform), the entire string is used to label the Y axis. This string should specify the units of the waveform.
2. If the waveform is a "gridded" waveform (see below for entries in the waveform structure that determine whether it is a gridded waveform) the first 20 characters in the string are used to label the Y axis. If the string has fewer than 20 characters, the entire string is used.

   Characters in the string beyond the 20th are used to generate labels for the grid levels. This substring has the following general format.

   ```
   head_labels/keyword/count block name/labels block name/tail_labels
                                   OR
   head_labels/keyword/phase sequence/category name/tail_labels
   ```

   Note the slash characters that are used to separate portions of the string.

   Here, `head_labels` and `tail_labels` are strings containing one or more groups of exactly 10 characters (the label in each group must be padded with blanks if necessary to fill it out to 10 characters). `Keyword` is either `LabelsBlock` or `LabelsBlockRange`. If the first string is the keyword `phase sequence` then the second string is the name of the category (the name that appears in the `categories` droplist on the user interface) whose phase sequence names should be used as labels for grid levels. Otherwise, the first string indicates that the parameter data block with name given by `labels block name` contains an array of null terminated strings which are to be labels for grid levels and the block with name given by `count block name` contains a single integer giving the number of strings in the labels block. The group of strings indicated by `head_labels` is used to label the first set of available grid levels (starting with the smallest Y axis value). The group of strings obtained from the parameter data block is used to label the next set of available grid levels. Finally, the group of strings indicated by `tail_labels` labels the final set of available grid levels. Any of these groups of labels is optional. The null terminated strings in the parameter data block can each have any length.

   Here are several examples. Note that in C language code, successive character strings surrounded by quotes are concatenated. This feature is used in these examples to separate the different portions of the string.

   – A simple set of labels for a waveform that can have one of 2 Y axis values.

     `"Off       " "On        "`

   – All labels obtained from a parameter data block.

     `"/LabelsBlock/A matrices : count/A matrices :names/"`

   – Labels obtained from a combination of 10 character strings and labels from a parameter data block.

```
"None      " "/LabelsBlock/A matrices : count/A matrices :names/" "All
```

– Labels obtained from the phase sequence names of category "Discharge Shape":

```
"/LabelsBlock/phase sequence/Discharge Shape/"
```

The `LabelsBlockRange` keyword is intended only for waveforms where the value of `gridded` (discussed below) is -1. If `keyword` is `LabelsBlockRange` then the count of labels completely determines the allowed range of Y axis values. The `ymin` and `ymax` values provided in the data item descriptor (discussed below) are ignored. The minimum Y axis value is the smallest allowed grid value and the maximum Y axis value is the `n`th grid level, where here `n` is the total number of grid labels provided. Thus the allowed Y axis range will vary depending on the number of grid labels and the operator will not be allowed to place a vertex at a grid level that is not labeled. This mode is intended for use with waveforms that are used to select which of an array of data objects in a parameter data block should be in use at any given time during the shot. Each data object has a name that is used to label the waveform editor display and the operator is prevented from selecting a data object that doesn't exist.

If the structure is used to describe a static data item, this entry is optionally used to pass data to the static data editor used for this data item.

- `restorepoint`: For a waveform, this entry is available for an installation-specific purpose. The nominal usage for this entry is as part of a facility to restore waveform vertices from a legacy database. This is described in Sec. C.

  If the structure is used to describe a static data item, this entry is optionally used to pass data to the static data editor used for this data item.

  If this entry isn't used, it should be a string containing a single blank.

- `restorescale`: For a waveform, this entry is available for an installation-specific purpose. The nominal usage for this entry is as part of a facility to restore waveform vertices from a legacy database. This is described in Sec. C.

  If the structure is used to describe a static data item, this entry is optionally used to pass data to the static data editor used for this data item.

  If this entry isn't used, it should be floating point value equal to 1.0.

- `restoreoffset`: For a waveform, this entry is available for an installation-specific purpose. The nominal usage for this entry is as part of a facility to restore waveform vertices from a legacy database. This is described in Sec. C.

  If the structure is used to describe a static data item, this entry is optionally used to pass data to the static data editor used for this data item.

  If this entry isn't used, it should be floating point value equal to 0.0.

- `archivepoint`: For a waveform, this entry is available for an installation-specific purpose. The nominal usage for this entry is as part of a facility to restore waveform vertices from a legacy database. This is described in Sec. C.

  If the structure is used to describe a static data item, this entry is optionally used to pass data to the static data editor used for this data item.

  If this entry isn't used, it should be a string containing a single blank.

- `description`: For a waveform, this entry is a string of less than 40 characters giving a brief description of the use or purpose of the waveform. This description appears on the plot on the user interface and in other displays of waveform data.

  For a static data item, this must be a string with the format "StaticData: description string." In this string, "StaticData" is used to identify this structure as the definition of a static data item. The "description string" portion is a description of the type of static data. The description string is used when restoring a static data item from an archived PCS setup. Matching description strings are required for a static data item to be restored from an archived setup into the database in the waveform server. Thus, the description string should be unique for each unique static data format (i.e. associated parameter data block names, number of parameter data blocks, and format of the data in the blocks). However, if a particular type of static data item is used in multiple places in the PCS, the same description string should be used in each place. This will allow the operator to restore the static data item from one location into another.

- `category`: The category code for this algorithm. This is always the macro `C_CODE`.

- `algorithm`: The algorithm code. This is always the macro `A_CODE`.

- `targets`: A list of data entry numbers specifying the processed data that is computed using the the raw data for this data item as input data. Each time the raw data are changed, the processed data are recomputed. Note that the proper macros must be used to generate the data entry numbers (see Sec. 2.10.4).

- `step`: Set this value to 1 if this is a step waveform, 0 otherwise. See Sec. 2.10.1 for background information on this.

  If the structure is used to describe a static data item, this entry is optionally used to pass data to the static data editor used for this data item.

- `gridded`: Set this value to 0 if the Y axis values of vertices for this waveform are continuous, i.e. the Y axis values of vertices can take any value. Set this value to -1 if the Y axis is composed of a grid of allowed values. Set this value to a count greater than 0 if the Y axis value of the waveform vertices can only take one of a specific list of values. The value here is the number of values in this list. See Sec. 2.10.1 for more information.

If the structure is used to describe a static data item, this entry is optionally used to pass data to the static data editor used for this data item.

- `gridpoints`: For a waveform, this entry is ignored if the waveform is not gridded (see above). If the waveform is gridded, there are two options here.

  1. If the value of `gridded` (discussed above) is `-1`, then this is an array of two values that define the grid. The first value is the minimum value on the grid, the second value is the increment. The waveform vertices can have Y axis values equal to

     $$\text{Yaxis} = \text{minimumvalue} + n \times \text{increment}$$

     where n is an integer greater than or equal to 0.
  2. If the Y axis for this waveform can only have one of a list of values (`gridded` is greater than 0), this is an array that specifies these values. The number of values in this array must be exactly equal to the count specified by the value of `gridded`.

  If the structure is used to describe a static data item, this entry is optionally used to pass data to the static data editor used for this data item.

- `ymin`: The minimum value that the Y axis value of the waveform vertices can have. If there is essentially no limit, use `-1.0e30` here.

  If the structure is used to describe a static data item, this entry is optionally used to pass data to the static data editor used for this data item.

- `ymax`: The maximum value that the Y axis value of the waveform vertices can have. If there is essentially no limit, use `1.0e30` here.

  If the structure is used to describe a static data item, this entry is optionally used to pass data to the static data editor used for this data item.

- `subset`: The number of the data item subset of the algorithm to which this data item belongs. This is an integer with value 1 or greater. If the algorithm does not use subsets, put 0 here.

- `access_control`: The default value for the access control mask (Sec. 2.17.2). The possible values here are 0 or `AC_NORESTORE` (Sec. 2.17.3).

4.8 External data item descriptors

The `EXTERNALDATA` code section initializes an array of descriptors for the external data items used by the algorithm. External data items are raw data items located in another shot phase which contain data that is used to compute processed data for the algorithm.

The descriptor for an external data item specifies where that data item is located. The data item is identified by four strings: category identifier, algorithm identifier, phase name, and data item name. Note that the category and algorithm are specified by their identifier strings, not their name strings. Any of these strings can contain one or more asterisks as a multicharacter wildcard and one or more question marks as a single character wildcard. If one of these strings is irrelevant (like the phase name), the string can contain only an asterisk so that anything will match it.

An external data item descriptor is a structure of type `externaldata`. This section of code initializes a list of these structures. This code becomes part of an array of structures in the waveform server that defines all of the external data items known to the control system.

The `externaldata` structure is defined as follows in the infrastructure header file `serverdefs.h`. The individual components of the structure are described in detail below.

```
struct externaldata {
                int category;
                int algorithm;
                char *name;
 char *cat_id;
 char *alg_id;
                char *phase_name;
                int targets[MAX_TARGETS_AFFECTED];
               };
```

In the structure definition above, only the components of the structure that are initialize in the algorithm code are shown. There are some additional components at the end of the structure that are initialized by the waveform server code when the waveform server is started. Here, these additional components are ignored since they are ignorable by the algorithm author.

The `externaldata` structures are located in the algorithm master file as follows.

```
#ifdef EXTERNALDATA
{  /* brace that starts a external data structure */
/* structure initializations. */
}, /* ending brace, note the comma */

/* as many other structures as required. */

{
/* the last  external data structure for this algorithm */
}, /* note that there must be a comma here also */
#endif
```

Each structure contains the following elements.

- `category`: The category code for this algorithm. This is always the macro `C_CODE`.
- `algorithm`: The algorithm code. This is always the macro `A_CODE`.
- `name`: A string with a maximum length of 20 characters giving the name of the external data item. The string can contain one or more asterisks as a multicharacter wildcard and one or more question marks as a single character wildcard.
- `cat_id`: A string giving the identifier for the category in which the external data item is contained. The string can contain one or more asterisks as a multicharacter wildcard and one or more question marks as a single character wildcard.
- `alg_id`: A string giving the identifier for the algorithm in which the external data item is contained. The string can contain one or more asterisks as a multicharacter wildcard and one or more question marks as a single character wildcard.

  There are two possible special strings that can be used in place of an actual algorithm identifier.

  - If the algorithm identifier is `sequence`, then the external data item is assumed to be a phase sequence waveform in the category specified by the category identifier string. The phase name is ignored.
  - If the algorithm identifier is `phasetable`, then the external "data item" is the list of shot phases in the category specified by the category identifier string. The data item name and phase name are ignored. The phase table for a category is considered to have changed if the list of phases is changed (i.e. any phase is deleted or added), the algorithm used by any phase is changed or the entry behavior of any phase is changed.
- `phase_name`: A string giving the name of the phase containing the external data item. The string can contain one or more asterisks as a multicharacter wildcard and one or more question marks as a single character wildcard.
- `targets`: A list of data entry numbers specifying the processed data that is computed using the the data for this external raw data item as input data. Each time the raw data are changed, the processed data are recomputed. Note that the proper macros must be used to generate the data entry numbers (see Sec. 2.10.4).

4.9 Name changes

The `NAME_CHANGES` code section initializes an array of descriptors that provide information about data items whose names have been changed since an earlier version of the algorithm. By providing the appropriate name change descriptor, a data item name can be changed and the data in an archived PCS setup for the old data item will be restored automatically into the data item with the new name. If the name change descriptor isn't provided, the user can still restore the data for the archived data item into the data item with the new name. However, this requires the extra effort of specifying the source and target data item names in the restore graphical user interface Sec. 7.5.3.3.

A name change descriptor is a structure of type `name_change`. This section of code initializes a list of these structures. This code becomes part of an array of structures in the waveform server that defines all of the name changes known to the control system.

The `name_changes` structure is defined as follows in the infrastructure header file `serverdefs.h`. The individual components of the structure are described in detail below.

```
struct name_change {
                 int category;
                 int algorithm;
                 char *old_name;
                 char *current_name;
                   };
```

The `name_change` structures are located in the algorithm master file as follows.

```
#ifdef NAME_CHANGES
{  /* brace that starts a name change structure */
/* structure initializations. */
}, /* ending brace, note the comma */

/* as many other structures as required. */

{
/* the last name change structure for this algorithm */
}, /* note that there must be a comma here also */
#endif
```

Each structure contains the following elements.

- `category`: The category code for this algorithm. This is always the macro `C_CODE`.
- `algorithm`: The algorithm code. This is always the macro `A_CODE`.
- `old_name`: A string giving the old name of the data item.
- `new_name`: A string giving the new name of the data item.

4.10 Waveform server function prototypes

There are several functions that each algorithm provides that become part of the waveform server. Examples are the `alg_vectors` and the `alg_vertices` functions and the various functions to handle parameter data or real time scratch space. It is necessary to provide ANSI C function prototypes for these functions. These prototypes are located in the `SERVERPROTOS` section of the algorithm master file.

For most of the functions that the algorithm master file provides for the waveform server, the function arguments must follow a predefined format. So, providing the function prototypes is usually a matter of copying the prototypes from an example algorithm master file and changing the function names appropriately. An example of code to provide function prototypes is shown below.

```
#ifdef SERVERPROTOS
extern void alg_vectors(int, struct shotphase *);
extern void alg_vertices(struct shotphase *);
extern void alg_parameters(struct shotphase *);
extern void alg_forhost(struct shotphase *, int,
                        char **, int *);
extern void alg_foruser(struct shotphase *,
                        char **, int *);
extern void alg_fromuser(struct shotphase *,
                         char *, int);
extern void alg_restore(struct shotphase *,
                        char *,   int,
                        short *,  int,
                        int *,    int,
                        float *,  int,
                        double *, int);
extern void alg_archive(struct shotphase *,
                        char **,   int *,
                        short **,  int *,
                        int **,    int *,
                        float **,  int *,
                        double **, int *);
#endif
```

4.11 Functions for the waveform server

In the `MASTERVECTORS` section of the algorithm master file are located the various functions specific to the algorithm that become part of the waveform server. The functions are written as standard C language code. The code can use definitions from the symbols definitions section, the function prototypes are as given in the previous master file section and the utility routines described in Sec. 14 can be used.

This code is located between two standard C preprocessor directives as follows.

```
#if defined(MASTERVECTORS)
/* Code for the various functions. */
#endif
```

Examples of functions that belong in this section of the algorithm master file are the following.

1. The `alg_vectors` routine. See Sec. 2.10.5 for details on this routine.
2. The `alg_vertices` routine. See Sec. 2.10.9 for details on this routine.
3. The various routines to handle parameter data. These routines are only necessary if the algorithm uses parameter data. See Sec. 2.11 for background information on these routines.

4.12 Function prototypes for host_cpu process code

If a control algorithm has functions that are to be compiled and linked with any of the `host_cpu` processes, prototypes for these functions should be provided. These prototypes are placed in the `HOSTPROTOS` section of the algorithm master file. This section would look like the following.

```
#ifdef HOSTPROTOS
/* host_cpu process function prototypes placed here */
#endif
```

4.13 Functions for the real time host process

A control algorithm may have the need to have some code executed by the `host_cpu` process as part of the pre-shot setup or as part of the post-shot cleanup (see Sec. 2.6). The `HOSTCATCODE` section of the algorithm master file is the location where any C language functions that are to be linked with the `host_cpu` process can be included. The functions to be called either pre-shot or post-shot are declared to the PCS by placing their names in the `categories_algorithms` array for the host process (see Sec. 4.6).

In the `HOSTCATCODE` section there could be two main functions: the pre-shot and post-shot routines. See Sec. 2.6 for background information on this function and Sec. 4.6 for a description of the call format for the functions. Prototypes for these functions should be placed in the `HOSTPROTOS` section of the master file (Sec. 4.12).

If an algorithm uses multiple real time CPUs, there might be different code to be compiled for the `host_cpu` process for each of the real time CPUs. Macros are used to determine which code to compile. When the code for the `host_cpu` process for a given real time CPU is compiled, a macro with a name of the form `category_CPUx` is defined for each category that executes code on that real time CPU. Here, "`category`" is the identifier for the category in uppercase letters and "`x`" is the virtual CPU index (i.e. A, B,:.) (see Sec. 2.9.1). For instance, `MYCATEGORY_CPUA` would be defined when the `host_cpu` code is compiled for the real time CPU that plays the role of virtual CPU A for the category with identifier `MYCATEGORY`. Code in the `HOSTCATCODE` section should always be delineated by a test on a macro of this type. This will make clear to anyone examining the master file the code that is relevant to each virtual CPU used by the algorithm and it will prevent unneeded code from being compiled into the `host_cpu` process.

So, the code to be included in the `host_cpu` process is specified as shown in the following example.

```
#ifdef HOSTCATCODE
#ifdef MYCATEGORY_CPUA
/* code for the host_cpu process of virtual CPU A */
#endif
#ifdef MYCATEGORY_CPUB
/* code for the host_cpu process of virtual CPU B */
#endif
#endif
```

4.14 Function prototypes for the real time code

The algorithm master file can provide C language functions to become part of the code that executes on the real time processor. In the `REALTIMEPROTOS` section the algorithm author should provide ANSI C function prototypes for these functions. The prototype code is located as follows.

```
#ifdef REALTIMEPROTOS
extern void alg(struct rt_heap_misc *);
#endif
```

A control algorithm will usually have at least one custom function that is the main routine called to implement the algorithm in real time. There can be an unlimited number of other functions called by this main routine. The main function always has arguments as shown in the example above.

The structure of type `rt_heap_misc` is the main source for information about the data structures in the memory of the real time processor. All real time data structures can be reached through pointers in this structure. See Sec. 13.3 for a detailed description of the real time data structures.

If the algorithm uses pre-shot initialization, post-shot cleanup, phase-begin, or phase-enter functions, the prototypes for these functions would also be located here. See Sec. 2.6 for background information on these functions and Sec. 4.6 for a description of the call format for these functions.

4.15 The real time code

The code that becomes part of the executable for the real time process is located in the `REALTIME` section. If an algorithm uses multiple real time CPUs, there might be different code to be compiled for the executable for each of the real time CPUs. Macros are used to determine which code to compile. When the code for the executable for a given real time CPU is compiled, a macro with a name of the form `category_CPUx` is defined for each category that executes code on that real time CPU. Here, "`category`" is the identifier for the category in uppercase letters and "`x`" is the virtual CPU index (i.e. A, B,:.) (see Sec. 2.9.1). For instance, `MYCATEGORY_CPUA` would be defined when the code is compiled for the real time CPU that plays the role of virtual CPU A for the category with identifier `MYCATEGORY`. Code in the `REALTIME` section should always be delineated by a test on a macro of this type. This will make clear to anyone examining the master file the code that is relevant to each virtual CPU used by the algorithm and it will prevent unneeded code from being compiled into the `host_cpu` process.

So, the code to be included in the executable for a real time CPU is specified as shown in the following example.

```
#if defined(REALTIME)
#ifdef MYCATEGORY_CPUA
/* code for the executable for virtual CPU A */
#endif
#ifdef MYCATEGORY_CPUB
/* code for the executable for virtual CPU B */
#endif
#endif
```

Depending on how the function vector is organized for the category to which the algorithm belongs, there could be one or more routines that are called through the function vector. Each of these routines has the following call format.

```
void algfunction(struct rt_heap_misc *rtheap)
```

Here, `rtheap` is a pointer to the structure that defines the layout of the data in the memory of the real time process. All of the data that is available in real time can be reached through pointers in `rtheap`. See Sec. 13.3 for more information on this structure.

If the algorithm uses pre-shot initialization, post-shot cleanup, phase-begin, or phase-enter functions, the code for these functions would also be located here. See Sec. 2.6 for background information on these functions and Sec. 4.6 for a description of the call format for these functions.

4.16 How master files are combined

As described in sections 4.3 through 4.15, the algorithm master file is broken into several sections, each delineated by an `ifdef` test on a particular macro. For instance, the section containing C language header type definitions is the `CONTROLDEF` section (Sec. 4.3).

The code in all of the algorithm master files is combined when the PCS is built to make the executables for the various processes that make up the PCS. The code in the individual sections of the master files is included in the PCS in the locations where it is needed. This differs from the traditional C language code structure where the complete set of code for a particular task is located in a single `.c` file. The PCS code is organized into algorithm master files instead so that the code for a given algorithm can be easily organized together.

When an algorithm is written, it is possible to consider the algorithm as a single, modular unit that plugs into the PCS by filling in the required sections of the master file without worrying about exactly how the code is compiled into the PCS. The algorithm author can simply focus on the description of the algorithm that goes in each master file section. However, it is probably desirable at times to understand exactly how the C language code in the algorithm master files is distributed when the PCS is built. This is the subject of this section.

The code in each of the master file sections falls into one of the following categories.

- Header file type definitions (macros and variable type definitions).
- A list of elements that will become part of an array of structures (Sec. 4.16.3).
- Completely specified, preinitialized, arrays.
- Prototypes for functions.
- Complete C language functions.

The algorithm master file can, optionally, contribute code to each of the following processes within the PCS.

- The waveform server.

- The `host_cpu` process that is associated with each real time CPU.
- The executable image for each of the real time CPUs.

Usually a given algorithm wouldn't have code for all of the real time CPUs since the algorithm probably only executes on one or, at most, a few of the real time CPUs.

The algorithm master file code is assembled into each of the PCS processes listed above by including the code into one of a set of infrastructure files. These files fall into 2 groups.

- Include files that are used as components of the code for the PCS processes. These files are the following.
  - `categories.h` (Sec. 4.16.1).
  - `controldefs.h` (Sec. 4.16.2).
  - `control.h` (Sec. 4.16.4). This file is a component of only the waveform server (Sec. 4.16.5).
- The main file for each of the PCS processes. These are the following.
  - `wavemain.c`: the main routine for the waveform server (Sec. 4.16.5).
  - `hostmain.c`: the main routine for the `host_cpu` processes (Sec. 4.16.6).
  - `realmain.c`: the main routine for the executable for a real time processor (Sec. 4.16.7).

The remainder of this section discusses these infrastructure files.

4.16.1 categories.h

The basic method for including code from a master file was outlined in Sec. 4.1. Given a section of the master file delineated by the macro `CODE_SECTION_1`,

```
#ifdef CODE_SECTION_1
  /* a bunch of code here */
#endif
```

then, in a particular PCS infrastructure file, installation-specific master file code is included as in this example.

```
#define CODE_SECTION_1 1
  /* include one or more master files here */
#undef CODE_SECTION_1
```

So, in this example, only the code in the master file that is compiled when the macro `CODE_SECTION_1` is defined will be compiled into the example process.

As described in Sec. 4.1, an algorithm is installed into the PCS by including the algorithm master file into a file called `categoryalgorithms.h` where `category` is the identifier for the particular category to which the algorithm belongs. So, this allows the code for all of the algorithms for a given category to be combined.

The `categoryalgorithms.h` file is then included in another file, `categories.h`, which allows the code for all algorithms in all categories to be combined. For example, suppose a PCS installation had the categories `category1` and `category2`, then `categories.h` would contain the following.

```
#define C_CODE 1
#include "category1algorithms.h"
#undef C_CODE

#define C_CODE 2
#include "category2algorithms.h"
#undef C_CODE
```

Note in the example the use of the macro `C_CODE`. This macro is analogous to the `A_CODE` macro described in Sec. 4.1. It defines the "category code number" which some of the infrastructure software uses to identify the category. The macro `C_CODE` will appear in the algorithm master file in several places. It is defined in the `categories.h` file to a value that is unique for each category.

For each category, there is a category master file that has a structure like an algorithm master file. The details of the category master file are destined to be described elsewhere. For the purpose of this section, it is simply useful to note that the `categories.h` file also includes all of the category master files. These files describe to the PCS each of the categories that have been installed.

So, anywhere in the PCS code that it is necessary to include a particular code section from all of the category and algorithm master files, it is only necessary to place a short code section that looks like the following.

```
#define CODE_SECTION_1 1
#include "categories.h"
#undef CODE_SECTION_1
```

4.16.2 controldefs.h

The fundamental C language header file for the installation-specific definitions in the PCS code is called `controldefs.h`. This file is located in the infrastructure directory. It contains the following code segment.

```
#define CONTROLDEF
#include "categories.h"
#undef CONTROLDEF
```

Here, the file `categories.h` is included because it will cause all of the algorithm and category master files to be included as described in Sec. 4.16.1.

Recall that all macro and variable type definitions that are part of the code for an algorithm go into the `CONTROLDEF` section (Sec. 4.3). So, any place in the PCS that the complete set of installation-specific definitions is required, there will be the following line.

```
#include  "controldefs.h"
```

4.16.3 The partial array of structures

Several of the code sections in an algorithm master file contain code which is a portion of an array of initialized structures. These sections are `CONTROL`, (Secs. 4.6 and 4.6.1), `SAVE_CODE_INCLUDES` (Sec. 4.4), `HOST_CONTROL`, (Secs. 4.6 and 4.6.2), `RT_CONTROL` (Secs. 4.6 and 4.6.3), `WAVEFORMS` (Sec. 4.7), `NAME_CHANGES` (Sec. 4.9), and `EXTERNALDATA` (Sec. 4.8).

When the PCS is compiled, the array portions of a particular data type in all of the master files are combined to make a single large array of structures. This is done in the infrastructure files `control.h`, `hostmain.c` and `realmain.c`. The arrays assembled in `control.h` become global variables in the waveform server. The arrays assembled in `hostmain.c` become global variables in the `host_cpu` process for each real time CPU. The arrays assembled in `realmain.c` become global variables in the code that executes on the real time processors.

Here is a brief outline of the code for each of the master file code sections that contain partial arrays of initialized structures. The first item in the list below has some extra explanation that applies to all of the other items in the list.

- `CONTROL` (from `control.h`):

  ```
  #define CONTROL 1
  struct cat_alg categories_algorithms[] =
  {
  #include "categories.h"
        {0}        /* terminated with a blank structure */
  };
  #undef CONTROL
  ```

  This example shows how all of the category and algorithm descriptors contained in the master file code section `CONTROL` are combined to make a single array of structures of type `cat_alg`. The array is called, in this case, `categories_algorithms`. Note that each structure in an array that is created in this way contains the necessary information, usually the category and algorithm code numbers, to allow the PCS to identify the algorithm to which the structure belongs. In addition, the end of the array is always marked by a structure filled with the default values provided by the compiler for an uninitialized structure (usually zeros).

Here are other structure arrays which are also included:

- `NAME_CHANGES` (from `control.h`):

```
#define NAME_CHANGES 1
struct name_change name_changes[] =
{
#include "categories.h"
{0} /* terminated with a blank structure */
};
#undef NAME_CHANGES
```

- `WAVEFORMS` (from `control.h`):

```
#define WAVEFORMS 1
struct waveform waveforms[] =
{
#include "categories.h"
{0} /* terminated with a blank structure */
};
#undef WAVEFORMS
```

- `EXTERNALDATA` (from `control.h`):

```
#define EXTERNALDATA 1
struct externaldata externalitems[] =
{
#include "categories.h"
{0} /* terminated with a blank structure */
};
#undef EXTERNALDATA
```

- `SAVE_CODE_INCLUDES` (from `control.h`):

```
#define SAVE_CODE_INCLUDES 1
struct save_code_info save_code_infos[] =
{
```

```
#include "categories.h"
{0} /* terminated with a blank structure */
};
#undef SAVE_CODE_INCLUDES
```

- HOST_CONTROL (from `hostmain.c`):

```
#define HOST_CONTROL 1
struct host_cat_alg categories_algorithms[] =
   {
#include "categories.h"
      {0, 0, 0, 0},/* terminated with a blank structure */
   };
#undef HOST_CONTROL
```

- RT_CONTROL (from `realmain.c`):

```
#define RT_CONTROL 1
struct rt_cat_alg rt_categories_algorithms[] =
   {
#include "categories.h"
      {0, 0, 0, 0, 0, 0}, /* terminate with a empty structure */
   };
#undef RT_CONTROL
```

4.16.4 control.h

The waveform server process has a group of global variables that provide information about the algorithms, categories and real time CPUs that are configured into the PCS. These global variables are, for the most part, either arrays of structures that are assembled from the partial arrays (Sec. 4.16.3) provided in the master files or variables that are completely specified in the master files. All of these global variables for the waveform server are defined in the infrastructure file `control.h`. The file `serverglobals.h` is a companion file that has a reference to each global variable.

Section 4.16.3 shows how the arrays of structures assembled from partial arrays are included into `control.h`.

Below is another excerpt from the file `control.h` showing how the completely specified variables in master files are included as waveform server global variables. This excerpt also shows that `control.h` includes sections from the category master files. (The specifics of these sections still need to be documented somewhere.)

Note that `control.h` includes the master files by including the file `categories.h` as described in Sec. 4.16.1.

```
/*
A code excerpt from control.h.
*/
/*
Completely specified arrays.
*/
#define WAVEGLOBALS 1
#include "categories.h"
#undef WAVEGLOBALS
/*
=====================================================
The following entries include data from the CPU
and category master files.
=====================================================
*/
/*
The descriptors for the real time CPUs.
*/
#define CONTROLCPUS 1
struct cpu_specs rtcpus[] =
   {
#include "cpus.h"
      {0}   /* terminated with a blank structure */
   };
#undef CONTROLCPUS
/*
The descriptors for the default list of phases to be
defined for the categories.
*/
#define PHASETABLEDEF 1
struct default_shotphase phasetable[] =
                 {
#include "categories.h"
                  {0}                 /* terminated with a blank structure */
                 };
#undef PHASETABLEDEF
/*
The descriptors for the list of phase sequences to be
defined for the categories.
*/
#define PHASESEQTABLEDEF 1
```

```
struct initialize_wvphsseq phaseseqtable[] =
                {
#include "categories.h"
                 {0}                /* terminated with a blank structure */
                };
#undef PHASESEQTABLEDEF
```

4.16.5 wavemain.c

The infrastructure file `wavemain.c` contains the `main()` function for the waveform server. An excerpt from this file is shown below. The primary feature of note is that, at the bottom, all of the installation-specific code defining complete functions that is placed in the `MASTERVECTORS` section (Sec. 4.11) of the master files is included and compiled. Note also that the `controldefs.h` file discussed in Sec. 4.16.2 is included so that the code from the `MASTERVECTORS` section has access to all definitions from all master files. The file `control.h` (Sec. 4.16.4) is included in order to define the global variables within the waveform server that are initialized by installation-specific code. This file also utilizes the function prototypes from the `SERVERPROTOS` master file section (Sec. 4.10).

```
/*
An excerpt from wavemain.c.
*/
/*
All of the installation-specific definitions from the CONTROLDEF
section of the master files.
*/
#include "controldefs.h"
/*
All installation-specific function prototypes from the master files
that describe functions compiled into the waveform server.
*/
#define SERVERPROTOS 1
#include "categories.h"
#include "cpus.h"
#undef SERVERPROTOS
/*
The global variables that contain data about the various
installation-specific algorithms.
*/
#include "control.h"
/*
```

```
The main function of the waveform server.
*/
int main(int argc, char **argv)
{
/*
code that parses the command line and then calls the function
containing the bulk of the waveform server infrastructure code.
*/
}
/*
Here, all of the installation-specific sections from the master files
that contain complete functions that are intended to be used in the
waveform server are compiled. Note that all of the header-type
definitions from the CONTROLDEF sections of all master files are
defined above for use by this code when the controldefs.h is included.
Also, all of the function prototypes were included above from the
master files.
*/
#define MASTERVECTORS
#include "categories.h"
#include "cpus.h"
#undef MASTERVECTORS
```

4.16.6 hostmain.c

The file containing the `main()` function for the `host_cpu` processes is the infrastructure file `hostmain.c`. This single file is used to build the executables for the `host_cpu` processes for each of the real time CPUs.

The `HOSTCATCODE` section of the algorithm master file (Sec. 4.13) contains the code that is compiled with `hostmain.c`. This master file section can contain the code for multiple real time CPUs if that is appropriate for the algorithm (as discussed in Sec. 4.13 and Sec. 2.9.1). PCS configuration macros are used to determine which parts of the algorithm master file are actually compiled. When `hostmain.c` is compiled the `CPU_NUM` macro is defined on the build command line to indicate the physical CPU number for which the `host_cpu` process is being built. Then, this macro is tested in the PCS configuration file and used to determine which set of the `category_CPUx` (Sec. 4.13) virtual CPU macros should be defined.

Below is an excerpt from the `hostmain.c` file. The following are some features to note.

- The file `controldefs.h` is included so that all of the installation-specific header-file-type definitions are available when the `host_cpu` code from the algorithm master file is compiled.

- The function prototypes declared for the `host_cpu` process are included.
- The `HOST_CONTROL` master file section (Sec. 4.6.2) is included to obtain the list of algorithm descriptors.
- There are several functions that execute code on behalf of the installation-specific algorithms either before or after a shot (Sec.2.6). These functions are:
  - `run_host_init`: executes application-specific preshot initialization code.
  - `run_host_end`: executes application-specific postshot cleanup code.
  - `run_host_calc`: executes application-specific analysis immediately after the shot before the postshot cleanup and archive code.
  - `run_host_archive`: executes application-specific code that archives data. This is data archival that is in addition to the standard data archival performed by the infrastructure code.

  These functions have the same general structure. First, there is inline code that comes from the category master file. (In retrospect, the use of inline code here rather than calling complete functions can be viewed as rather unusual. In practice, there usually isn't any code provided here that is category-wide. Initialization and cleanup code is generally provided as part of an algorithm.) Then, there is a function called that looks at the list of algorithm descriptors that was compiled as part of the `HOST_CONTROL` section. The algorithm descriptor lists the initialization or cleanup function that should be called. The appropriate function is called for each phase of each category that is defined in the shot setup. Note that the initialization and cleanup functions are optional and if none of these functions need to be specified for an algorithm, there doesn't need to be an algorithm descriptor compiled into the `host_cpu` process.
- The `HOSTCATCODE` section of each category and algorithm master file is included in order to compile all of the functions provided by categories that should be part of the `host_cpu` process.
- There is a file called `hostinstall.h` that provides the code to interface the `host_cpu` process to the specific real time processor used in the PCS installation. This file is included to provide definitions and it has a `HOSTCODE` section containing complete functions that are compiled with the `host_cpu process`.

```
/*
An excerpt from hostmain.c.
*/
/*
All of the installation-specific definitions from the CONTROLDEF
```

```
section of the master files.
*/
#include "controldefs.h"
/*
All installation-specific function prototypes from the master files
that describe functions compiled into the host_cpu processes.
*/
#define HOSTPROTOS 1
#include "categories.h"
#include "cpus.h"
#undef HOSTPROTOS
/*
The list of descriptors of control algorithms.  It is optional
for an algorithm to provide a descriptor to be placed here.
*/
#define HOST_CONTROL 1
struct host_cat_alg categories_algorithms[] =
   {
#include "categories.h"
      {0},/* terminated with a blank structure */
   };
#undef HOST_CONTROL
/*
The main routine for the host_cpu process that parses the
command line and then calls the code that handles the main
job of the host_cpu process which is to set up the
real time processor for executing shots.
*/
int main(int argc,char **argv)
{
/* The infrastructure code for the host_cpu process. */
}
/*
Preshot and postshot initialization, cleanup, calculation and
archiving functions.
*/
int run_host_init(struct rt_heap_misc *hssc_rtheap)
{
#define HOSTINIT 1
#include "categories.h"
#undef HOSTINIT
```

```
   return run_host_algorithms("init",hssc_rtheap);/* algorithm specific code */
}
int run_host_end(struct rt_heap_misc *hssc_rtheap)
{
   int error;

   error = run_host_algorithms("end",hssc_rtheap);     /* algorithm specific */

#define HOSTEND 1
#include "categories.h"
#undef HOSTEND
   return error;
}

int run_host_calc(struct rt_heap_misc *hssc_rtheap)
{
   int error;

   error = run_host_algorithms("calc",hssc_rtheap);     /* algorithm specific */

#define HOSTCALC 1
#include "categories.h"
#include "cpus.h"              /* cpu specific code */
#undef HOSTCALC
   return error;
}
int run_host_archive(struct rt_heap_misc *hssc_rtheap)
{
   int error;

   error = run_host_algorithms("archive",hssc_rtheap); /* algorithm specific */

#define HOSTARCHIVE 1
#include "categories.h"
#include "cpus.h"              /* cpu specific code */
#undef HOSTARCHIVE
   return error;
}
/*
Complete functions provided in category and algorithm master files.
```

```
*/
#define HOSTCATCODE 1
#include "categories.h"
#undef HOSTCATCODE
/*
Installation-specific code that the host_cpu process to
the real time processors.
*/
#define HOSTCODE 1
#include "hostinstall.h"
#undef HOSTCODE
```

4.16.7 realmain.c

The file containing the `main()` function for the `realtime` processes is the infrastructure file `realmain.c`. This single file is used to build the executables for the `realtime` processes for each of the real time CPUs.

The `REALTIME` section of the algorithm master file (Sec. 4.15) contains the code that is compiled with `realmain.c`. This master file section can contain the code for multiple real time CPUs if that is appropriate for the algorithm (as discussed in Sec. 4.15 and Sec. 2.9.1). PCS configuration macros are used to determine which parts of the algorithm master file are actually compiled. When `realmain.c` is compiled the `CPU_NUM` macro is defined on the build command line to indicate the physical CPU number for which the `real time` process is being built. Then, this macro is tested in the PCS configuration file and used to determine which set of the `category_CPUx` (Sec. 4.15) virtual CPU macros should be defined.

Below is an excerpt from the `realmain.c` file. The following are some features to note.

- The file `controldefs.h` is included so that all of the installation-specific header-file-type definitions are available when the `real time` code from the algorithm master file is compiled.
- The function prototypes declared for the `real time` process are included.
- The `RT_CONTROL` master file section (Sec. 4.6.3) is included to obtain the list of algorithm descriptors.
- There are several functions that execute code on behalf of the installation-specific algorithms either before or after a shot (Sec.2.6). These functions are:
  - `run_realtime_init`: executes application-specific preshot initialization code.
  - `run_realtime_end`: executes application-specific postshot cleanup code.

These functions have the same general structure. First, there is inline code that comes from the category master file. (In retrospect, the use of inline code here rather than calling complete functions can be viewed as rather unusual. In practice, there usually isn't any code provided here that is category-wide. Initialization and cleanup code is generally provided as part of an algorithm.) Then, there is a function called that looks at the list of algorithm descriptors that was compiled as part of the `RT_CONTROL` section. The algorithm descriptor lists the initialization or cleanup function that should be called. The appropriate function is called for each phase of each category that is defined in the shot setup. Note that the initialization and cleanup functions are optional and if none of these functions need to be specified for an algorithm, there doesn't need to be an algorithm descriptor compiled into the `real time` process.

- The `REALTIME` section of each category and algorithm master file is included in order to compile all of the functions provided by categories that should be part of the `real time` process.

- There is a file called `realinstall.h` that provides the code to interface the `real time` process to the specific real time processor used in the PCS installation. This file is included to provide definitions and it has a `REALTIME` section containing complete functions that are compiled with the `real time process`.

```
/*
An excerpt from realmain.c.
*/
/*
All of the installation-specific definitions from the CONTROLDEF
section of the master files.
*/
#include "controldefs.h"
/*
All installation-specific function prototypes from the master files
that describe functions compiled into the real time processes.
*/
#define REALTIMEPROTOS 1
#include "realinstall.h"
#include "categories.h"
#include "cpus.h"
#undef REALTIMEPROTOS
/*
The list of descriptors of control algorithms.  It is optional
for an algorithm to provide a descriptor to be placed here.
*/
```

```
#define RT_CONTROL 1
struct rt_cat_alg rt_categories_algorithms[] =
   {
#include "categories.h"
      {0},/* terminated with a blank structure */
   };
#undef RT_CONTROL
/*
The main routine for the real time process that parses the
command line and then calls the code that handles the main
job of the real time process which is to do feedback control
during a shot.
*/
int main(int argc,char **argv)
{
/* The infrastructure code for the real time process. */
}
/*
Preshot and postshot initialization and cleanup functions.
*/
int run_realtime_init(struct rt_heap_misc *rtheap)
{
#define REALTIMEINIT 1
#include "cpus.h"
#include "categories.h"
#undef REALTIMEINIT

   return run_realtime_algorithms("init",rtheap);/* algorithm specific code */
}
int run_realtime_end(struct rt_heap_misc *rtheap)
{
#define REALTIMEEND 1
#include "categories.h"
#include "cpus.h"
#undef REALTIMEEND

   run_realtime_algorithms("end",rtheap);     /* algorithm specific code */
}

/*
Complete functions provided in category and algorithm master files.
```

```
*/
#define REALTIME 1
#include "realinstall.h"
#include "categories.h"
#include "cpus.h"
#undef REALTIME
```

Chapter 5

Support for PID calculations

A feedback control application often computes a "proportional, integral, differential" (or PID) feedback command. This command is proportional to

$$P = G_p * V + G_d * dV/dt + G_i * \int V dt$$

where G_p, G_d and G_i are gain constants. In the standard PID calculation in the PCS, a low pass filter is applied to the error value before it is integrated and differentiated. So, $V = \text{filter}(E)$ where E is the error value and "filter" indicates a low pass filter. The PCS uses the PID calculation in many algorithms so there is built-in support for it.

The standard filter, derivative and integral calculations emulate the effect of a low or high pass RC filter. For each calculation, an RC time constant must be specified. The RC time constant comes into the calculation in a ratio to the time delay since the last feedback cycle (ΔT).

There are four different PID functions which can be used in the real time code. All of the PID related code can be found in the file `pid_includes.h` found in the common directory.

Older versions, `pid_calc` and `pidv2_calc` use PID lookup tables. These versions should no longer be used. See Sec. 5.0.3.

The two newer versions are `pidv3_calc` and `pidv4_calc`. These functions do not use the PID lookup tables so they are more accurate. Also, these functions have more arguments which allow for more flexibility in the PID calculations. Either of these two functions can be used. See Sec. 5.0.1 for more information about pidv4_calc. See Sec. 5.0.2 for more information about pidv3_calc.

In all cases, the PID calculation functions require a scratch area used to store results from the previous cycle. A scratch parameter data block should be used for this purpose. There are simple functions for each version which can be used to create this scratch parameter data block in the `alg_parameters` function.

5.0.1 Example of pidv4_ code segments

Assuming that the PID is calculated on CPUA of the category, the following code segments would be needed. Here we define two error elements, six continuous target elements for the gains, and six floating step target elements for the taus. Since the gains and taus are separate, version 4 of the PID functions must be used since the gains argument and the taus argument are separated for the function `pidv4_calc`.

The taus could be made into continuous targets if desired. In that case, version 3 of the PID functions must be used since the gains and the taus would be grouped together and passed as a single argument to the `pidv3_calc` function.

```
#ifdef CONTROLDEF
#define NUM_XXX_ERRORS 2

#define TA_XXX_CONTROL1_GP      CTA_XXX(1)
#define TA_XXX_CONTROL1_GD      CTA_XXX(2)
#define TA_XXX_CONTROL1_GI      CTA_XXX(3)
#define TA_XXX_CONTROL2_GP      CTA_XXX(4)
#define TA_XXX_CONTROL2_GD      CTA_XXX(5)
#define TA_XXX_CONTROL2_GI      CTA_XXX(6)

#define TA_XXX_CONTROL1_TAUP    FSTA_XXX(1)
#define TA_XXX_CONTROL1_TAUD    FSTA_XXX(2)
#define TA_XXX_CONTROL1_TAUI    FSTA_XXX(3)
#define TA_XXX_CONTROL2_TAUP    FSTA_XXX(4)
#define TA_XXX_CONTROL2_TAUD    FSTA_XXX(5)
#define TA_XXX_CONTROL2_TAUI    FSTA_XXX(6)

#define ISTA_XXX_FEEDBACK1_ON_OFF   ISTA_XXX(1)
#define ISTA_XXX_FEEDBACK2_ON_OFF   ISTA_XXX(2)

#define EA_XXX_ERROR1           EA_XXX(1)
#define EA_XXX_ERROR2           EA_XXX(2)

#define BA_XXX_PIDSTORAGE 1

#define SAVEROUTINE_XXX_CONTROL1      SAVEROUTINE_INDEX(1)
#define SAVEROUTINE_XXX_CONTROL2      SAVEROUTINE_INDEX(2)

#endif

#ifdef SAVE_CODE_INCLUDES
```

```
/*
Allow for saving of values from all cycles in fast data arrays.
args: algname,ptname prefix,part of ptname,saveroutine index,cpu
The first name in the user interface would be XXX_pids_C1,
the pointnames would be XXAPC1, XXAEC1, XXAVC1, XXADC1, XXAIC1
for the pvector, error, pout, dout, and iout.
*/
pidv4_saveroutines(XXX,XX,C1,SAVEROUTINE_XXX_CONTROL1,CPUA)
pidv4_saveroutines(XXX,XX,C2,SAVEROUTINE_XXX_CONTROL2,CPUA)
#endif

#ifdef WAVEFORMS
...
pidv4_waveglobals(XX,CONTROL1,TARNAME,CPU_CODE)
...
#endif

#ifdef WAVEFORMS
/*
setup a feedback on/off waveform (duplicate for second error)
setup gains as continuous target waveforms
setup taus as floating step target waveforms
TAUP and TAUD are usually in msec and TAUI is usually in secs
*/
{
 0,                          /* set to zero (message length) */
 {XMIN, XMAX, 0.0, 2.0},     /* plot scales */
 "FeedbackOnOff1",           /* waveform name, 20 chars max */
 "time (seconds)",           /* x axis label, 20 chars max */
 "                            Off        On", /* Y axis, 20 char label,
                                                plus discrete value labels of
                                                exactly 10 chars each */
 " ",                        /* restore pointname, 13 chars max */
 1.0,                        /* restore scale factor */
 0.0,                        /* restore offset */
 " ",                        /* archive pointname, 10 chars max */
 "Control on/off switch",    /* description, 24 chars max */
 C_CODE,                     /* control category index */
 A_CODE,                     /* control algorithm index */
 {TA_XXX(ISTA_XXX_FEEDBACK1_ON_OFF),0}, /* target vectors affected */
 1,                          /* 1 = step waveform, 0 = not a step waveform */
 2,       /* -1 = y axis grid, 0 = yaxis continuous, > 0 = y axis discrete */
 {0.0,1.0},                /* grid definition or discrete yaxis values */
 0.0,                      /* minimum y value allowed. */
 1.0,                      /* maximum y value allowed. */
 1,                        /* subset number */
 0                         /* access control flag */
},

pidv4_waveform(CONTROL1,TA_XXX_CONTROL1,TA_XXX)
pidv4_waveform(CONTROL2,TA_XXX_CONTROL2,TA_XXX)
#endif

#ifdef MASTERVECTORS
xxx_vectors(struct shotphase *phase)
{
/*
gains are dimensionless and the gain waveform is multiplied
by the feedbackonoff waveform so that gains are zero
when feedback is off
tau waveforms are all converted to seconds in real time
the second set would be the same
*/
    case_wv2mp(TA_XXX(TA_XXX_CONTROL1_GP),TA_XXX_CONTROL1_GP,
           CPUA,"Control1_GP","FeedbackOnOff1",0.0f,1.0f)

    case_wv2mp(TA_XXX(TA_XXX_CONTROL1_GD),TA_XXX_CONTROL1_GD,
           CPUA,"Control1_GD","FeedbackOnOff1",0.0f,1.0f)

    case_wv2mp(TA_XXX(TA_XXX_CONTROL1_GI),TA_XXX_CONTROL1_GI,
           CPUA,"Control1_GI","FeedbackOnOff1",0.0f,1.0f)

    case_fst_wvmp(TA_IP(TA_XXX_CONTROL1_TAUP),TA_XXX_CONTROL1_TAUP,
           CPUA,"Control1_TAUP",0.0f,0.001f)

    case_fst_wvmp(TA_IP(TA_XXX_CONTROL1_TAUD),TA_XXX_CONTROL1_TAUD,
           CPUA,"Control1_TAUD",0.0f,0.001f)

    case_fst_wvmp(TA_IP(TA_XXX_CONTROL1_TAUI),TA_XXX_CONTROL1_TAUI,
           CPUA,"Control1_TAUI",0.0f,1.0f)
}
```

```
xxx_parameters(struct shotphase *phase)
{
   pidv4_parameters(phase,
                    NUM_XXX_ERRORS,
                    FORHOSTVCPUA_MASK,
                    BA_XXX_PIDSTORAGE);
}
#endif

#if defined(REALTIME)
/*
This global variable is needed in order to
smoothly transition from one phase to another.
The last phase to call this function will be
the last phase to set this pointer, and thus,
the parameter data block in that phase will
be used throughout the shot.
*/
struct pidstore_v4 *xxx_pidstorage;
#endif

#if defined(REALTIME)
void xxx_init(struct rtshotphase *phase,struct rt_heap_misc *rtheap)
{
    pidv4_init(phase, NUM_XXX_ERRORS, BA_XXX_PIDSTORAGE, &xxx_pidstorage);
}

void xxx(struct rt_heap_misc *rtheap)
{
int i;
int calculate[NUM_XXX_ERRORS];
int use_gi[NUM_XXX_ERRORS];

   for(i=0; i<NUM_XXX_ERRORS; i++) calculate[i] = 1;
   for(i=0; i<NUM_XXX_ERRORS; i++) use_gi[i] = 1;
   pidv4_calc(
           NUM_XXX_ERRORS,  /* number of p vector elements to be calculated */
           rtheap->currenttime,
               /* time stamp to use */
           &calculate,
               /* 0: set pvector value to 0, zero scratch,
                  1: do calculation */
           &use_gi,
               /* 3: use integral term if sign is same as vout,
                  2: freeze the integral term,
                  1: integrate, 0: integral term set to zero */
           (rtheap->saveroutine - (SAVEROUTINE_XXX_CONTROL1)),
               /* saveroutine index of first element */
           &rtheap->pvector[EA_XXX_ERROR1-1],
               /* first element of pvector to be calculated */
           (float *)NULL,
               /* first element of shape vector to store results - not used */
           rtheap->alldata,
               /* pointer to the high frequency structure */
           &rtheap->errorvector[EA_XXX_ERROR1-1],
               /* first element of error vector used as input */
           &rtheap->adtarget[TA_XXX_CONTROL1_GP-1],
               /* pointer to the first of the gain values */
           &rtheap->adtarget[TA_XXX_CONTROL1_TAUP-1],
               /* pointer to the first of the tau values */
           xxx_pidstorage
               /* pointer to the pidstorage arrays for this calculation group */
           );
}
#endif
```

5.0.2 Using pidv3_ code segments

Version 3 of the PID functions are exactly the same as version 4 except that the three gain and three tau vector elements are all in the continuous section of the target vector. The six required target elements must be grouped together for each error. The remaining code from the example for version 4 (Sec. 5.0.1) is then the same except pidv3 is substituted for pidv4 and there is no taus argument to pidv3_calc.

```
   pidv3_calc(
          NUM_XXX_ERRORS,  /* number of p vector elements to be calculated */
          rtheap->currenttime,
              /* time stamp to use */
          &calculate,
              /* 0: set pvector value to 0, zero scratch,
                 1: do calculation */
          &use_gi,
              /* 3: use integral term if sign is same as vout,
                 2: freeze the integral term,
                 1: integrate, 0: integral term set to zero */
          (rtheap->saveroutine - (SAVEROUTINE_XXX_CONTROL1)),
              /* saveroutine index of first element */
          &rtheap->pvector[EA_XXX_ERROR1-1],
              /* first element of pvector to be calculated */
          (float *)NULL,
              /* first element of shape vector to store results - not used */
          rtheap->alldata,
              /* pointer to the high frequency structure */
          &rtheap->errorvector[EA_XXX_ERROR1-1],
              /* first element of error vector used as input */
          &rtheap->adtarget[TA_XXX_CONTROL1_GP-1],
              /* pointer to the first of the gain & tau values */
          xxx_pidstorage
              /* pointer to the pidstorage arrays for this calculation group */
          );
```

5.0.3 Using old pid lookup tables

Since the feedback cycle time is measured as an integral number of fine scale clock ticks, and because the cycle time is reproducible within some range, there is only a finite number of values that this $RC/\Delta T$ ratio can have. So, to speed the PID calculation it is convenient to use a lookup table for 6 values that depend on the $RC/\Delta T$ ratio.

This lookup table is calculated before the discharge and loaded into the real time process's memory. Raw input data from the PCS operator provides values for the RC time constants used to calculate the lookup table content. The address of the appropriate table to use during any given control cycle is held in the pidtau vector. This vector has one element for every PID calculation that must be made. The raw input data for the control algorithm can specify that the RC time constants should vary in time during the discharge. When a time constant changes, the lookup table must change. This is automatically handled by the PCS infrastructure code by changing the pointer value in the pidtau vector.

The standard procedure is that during each control cycle, the PID algorithm is applied to each element of the error vector and the result is written in the corresponding element of the P vector. Thus the error and P vectors are the same length. There is one element in the pidtau vector for each PID calculation, so the pidtau vector has the same length as the error and P vectors.

The error and P vectors can be broken into blocks with the PID calculation for each block performed at a different position in the control cycle relative to other calculations, as is required for the control applications.

For each PID operation there must be a set of three gain values, G_p, G_d and G_i. There is a section of the continuous target vector that is allocated to hold these gain values. This section of the target vector has three values for each element in the error vector.

In the real time code either one of two functions can be used, both are located in the file `pid_includes.h` found in the common directory.

- pid_calc

```
        pid_calc(rtheap,
                 NUM_XXX_ERRORS,
                 &rtheap->pvector[EA_XXX_ERROR1-1],
                 &rtheap->errorvector[EA_XXX_ERROR1-1],
                 &rtheap->adtarget[TA_XXX_CONTROL_GP1-1],
                 &rtheap->pidtauvector[EA_XXX_ERROR1-1],
                 xxx_pidstorage);
```

- pidv2_calc

```
        pid_calc(rtheap,
                 NUM_XXX_ERRORS,
                 SAVEROUTINE_XXX_CONTROL1,
                 &rtheap->pvector[EA_XXX_ERROR1-1],
                 &rtheap->errorvector[EA_XXX_ERROR1-1],
                 &rtheap->adtarget[TA_XXX_CONTROL1_GP-1],
                 &rtheap->pidtauvector[EA_XXX_ERROR1-1],
                 xxx_pidstorage);
```

In addition, the `alg_vectors` function requires the use of the case_make_pidtau (Sec. 14.1.10) macro.

```
    case_make_pidtau(PIDTAUA_XXX(EA_XXX_ERROR1),
                     EA_XXX_ERROR1,
                     CPUA,
                     "XXX_CONTROL1_TAUP",0.0f,0.001f,
                     "XXX_CONTROL1_TAUD",0.0f,0.001f,
                     "XXX_CONTROL1_TAUI",0.0f,1.0f)
```

The pid lookup tables require the setting of the following fields in the `cpu_specs` structure. This structure is initialized in the `cpus.h` file.

- piddtmin The minimum delta T value to use when constructing pid lookup tables for this cpu in units of fine scale clock ticks.

- piddtmax The maximum delta T value to use when constructing pid lookup tables for this cpu in units of fine scale clock ticks.
- piddtinc The increment in delta T between entries in a pid lookup table on this cpu in units of fine scale clock ticks. If this value is less than 20, then it is the power of 2 to be used as the increment, e.g., a value of 2 gives an increment of 4. This is the historical meaning of this field. If this value is 20 or more, then the value is the actual increment to use. If cycle times are fixed, then the increment should be set to the actual cycle time to get more accurate results.

Chapter 6

Periodic actions and raw data actions

There are two types of actions the PCS can perform by executing a function. One type of action can be done on a regular interval and/or at the time of first lockout. This is known as a `periodic action`. The other type of action allows raw data to be changed when another piece of raw data is updated. This is called a `raw data action`.

6.1 Periodic actions

A periodic action is the execution of a function in the PCS code at a regular timed interval. An example would be the reading of a file that gets written by an external program or by another computer. Since the PCS does not know when this file would be written, it can periodically look for the file to see if it is present or has been updated.

Now, it would be possible for the program that writes the file to actually send a message to the waveform server to update a piece of raw data, but this requires more coding to be done. It may be easier for one reason or another for the PCS to simply keep checking if the file has been changed.

It also may be desirable to do an action at the time of first lockout. Using the same example above, the content of the file would be guaranteed to be read before the shot is locked out so any raw data changes can be made and any dependent processed data can be recalculated.

Another example would be the creation of a file with current settings in the PCS that needs to be read by the control computer. Since the settings in the PCS can be changed up to the first lockout, an action function executed at first lockout assures that accurate data are written. Since the user can perform an unlock, make a change to these settings, and relock the shot, an action executed at first lockout would be executed again at the time of relock. Then the control computer could read the file after it issues the final lockout to the PCS.

A periodic function is declared by calling the following function (usually in the `alg_parameters` function).

```
int add_periodic_action(
    struct shotphase *phase,
    int interval,
    int bitmask_flag,
    char *data_item,
    int data_entry,
    int (*function)(int data_entry,struct shotphase *phase,int call_flags))
```

Function Arguments:

- `phase`: The pointer to the descriptor of the shot phase.
- `interval`: The time interval in seconds (0 means the function will not be executed on a regular basis). Note that an interval of about 30 or 60 seconds works best.
- `bitmask_flag`: A bitmask giving various options.
  - `ACTION_DO_AT_LOCKOUT`: execute the action at first lockout.
  - `ACTION_DO_NOT_RECORD_CHANGE`: do not record the change in the setup's change history.
  - `ACTION_DO_ONLY_FOR_PROCESSED_COPY`: only execute the action for the set of processed data. Set this bit if the function to execute is a function that only creates processed data.
  - `ACTION_DO_ONLY_FOR_IO_COPY`: only execute the action for the IO copy of raw data. Set this bit if the action only needs to be done once like the writing of a file.
- `data_item`: A character string which has one or more data item names (separate the names with a vertical bar —). Any processed data associated with these data items will be recalculated if the return value from the action function is greater than zero. This argument can be NULL if the function does not change any raw data associated with any data items.
- `data_entry`: The index of a data entry. If greater than zero, then the `alg_vectors` function will be called after the change is made to the processed copy of the raw data. This value can be zero.
- `function`: The name of the function to execute.

Action functions take three arguments, the data_entry, the shot phase pointer, and a call_flags argument. The function MUST return a value. The return value signals whether a change was made. A value ¿ 0 indicates a change, any other value indicates no change.

When the return value indicates a change, then the change is recorded in the change history and a message indicating the change is sent to all users with a user interface that is active (unless the ACTION_DO_NOT_RECORD_CHANGE bit is set in the bitmask_flag argument).

Since a periodic action can change raw data, the action is executed for both copies of the raw data unless the bitmask_flag argument indicates otherwise.

Note that only one data entry is allowed per action. If multiple data entries are required, then a different function should be used for each one. The data entry is only useful for the recalculation of the processed data that is derived from the raw data that would be updated by the action function. Any processed data assoicated with the data item(s) will also be recalculated.

The call_flags argument indicates whether the function is being called at first lockout (a value of ACTION_CALLED_AT_LOCKOUT) or whether it is being called at a regular interval (a value of ACTION_CALLED_AT_INTERVAL).

6.2 Raw data actions

A raw data action is a special type of periodic action (Sec. 6.1). A raw data action function is executed whenever another piece of raw data changes. An example would be the name of a file that the user sets from the user interface. The content of the file is really the raw data that is needed in the PCS. So when the file name changes, the raw data action function will be executed so that the file can be read.

Another example would be if two pieces of raw data conflict with one another. Normally, a raw data message would have to be generated to indicate such a conflict. A raw data action function could attempt to fix the conflict by changing one or the other piece of raw data. However, be aware that the function cannot determine which of the two pieces actually changed. So one data item may need to be thought of as "read-only".

After the waveform server processes a change to any raw data, a check is made to see if the item or items changed is associated with any raw data actions. If so, then the raw data action function is executed. This is true when either copy of raw data is changed. Only after all raw data action functions have been executed is processed data then recalculated. Note that if more than one data item is changed at the same time (as in a restore), a raw data action function will only be called once instead of once per data item.

A raw data function is declared by calling the following function (usually in the `alg_parameters` function).

```
int add_raw_data_action(
    struct shotphase *phase,
    char *data_item,
    int (*function)(int data_entry,struct shotphase *phase,int call_flags))
```

Function Arguments:

- `phase`: The pointer to the descriptor of the shot phase.
- `data_item`: A character string which has one or more data item names (separate the names with a vertical bar —). Any processed data associated with these data items will be recalculated. This argument cannot be NULL.
- `function`: The name of the function to execute.

Raw data action functions take the same three arguments as a periodic action function: data_entry which is always zero, the shot phase pointer, and a call_flags argument which gets set to the value ACTION_CALLED_FOR_RAWDATA.

Chapter 7

The user interface

This section discusses the user interface.

On the main Plasma Control System window (the one with the PCS logo) there are three menu options:

- File: has two submenu options:
  - version: displays information about the version of the PCS.
  - quit: quits out of the user interface.
- Control: has three submenu options:
  - next shot: brings up a window that handles the "next shot" (Sec. 7.2).
  - future shot: brings up a window where a future shot can be set up (Sec. 7.3).
  - shared shot: brings up a window which will communicate with another person's waveform editor allowing shared access (Sec. 7.4).
- Utilities: has one submenu option:
  - view pcs log: bring up a window that displays the status of the PCS, its processes, and any messages logged to the msgserver (Sec. 7.1).

7.1 The view log

The view log window displays the current status of the PCS. There are two status lights on this window: one that gives the status of the PCS processes and another that indicates whether there is a raw data problem in the waveform server.

The status light comes from the log messages that are sent from the host_cpu processes, the lockserver process, and the waveserver process that go to the msgserver process. These programs all have a heartbeat. When the msgserver stops hearing from any of the others, the status light will turn red. This usually means that a process has died or gotten hung up. This light also turns yellow if there are any errors reported during a shot.

The color of the data light comes from the "raw data problem" list. If there is a fatal raw data problem, then the light is red and a shot won't run. If there is a raw data warning, then the light is yellow and a shot can still run. An example of a warning is when the number of samples for a cpu is too small to cover the whole shot - not a killer but something that may need attention.

The view log also displays the progress of the shot cycle in a bar graph. The bar graph will indicate the progress of the host real time processes in getting ready after first and final lockouts and the progress of the post-shot archiving. The current status of the shot is also displayed.

From the `Utilities` menu the "show pcs log messages" option will show all the latest messages from the message server. The "hide pcs log messages" will hide this part of the display. The "pcs process table" option will display a window with more detailed information about the PCS processes and the progress of the shot cycle. The "show realtime errors only" option will display only messages from the real time processes.

7.2 The "next shot" option

The "next shot" window is used to program the next plasma shot in "operations" mode, the user's shot setup if in "test" mode, and the user's shot setup for a simulation shot in "stand alone" mode. In all cases, this graphical user interface communicates to a waveform server process over a given port. In operations and test modes the port is a fixed port number. In stand alone mode the port number will vary for each user executing the runsa script.

The same two status lights that are on the view log window are also on this window.

Refer to sections on the waveform editor Sec. 7.5 for more information.

7.3 The "future shot" option

The "future shot" window is used to program a "future" shot. The user chooses a name to associate with the setup and the setup is saved in a file name that has the user's username prefixed to it. Each future shot window will have associated with it a dedicated instance of the waveform server.

There are several differences between a "next shot" window and a "future shot" window. For one there is no status light or data light. This is because there is no lockserver, msgserver, host_cpu, or realtime processes associated with a future shot. The only process is the waveserver. Another difference is that there is no "manual cycle control" under the File menu. And there is no access control choices under the File menu.

Refer to sections on the waveform editor Sec. 7.5 for more information.

7.4 The ”shared shot” option

The ”shared shot” option is mainly for testing purposes. It allows a user to attach to someone else’s waveform server dedicated to a future shot or stand alone next shot. After choosing this option, a window requesting the host name and access code of the waveform server pops up. The access code can be gotten from the ”access information” choice under the File menu of the other person’s waveform editor.

Once the access code is entered, a waveform editor will be created which is attached to the other person’s waveform server. Any changes made by one person will be available to the other person. This emulates the behavior of ”operations” mode.

7.5 The waveform editor

The waveform editor displays a great deal of information. The following sections describe the various buttons, checkboxes, menus, lists, and entry boxes found on the waveform editor window. The majority of the window, however, is the plot region that displays one or more waveforms.

7.5.1 Menu options

There are several menu options on the waveform editor.

7.5.1.1 File menu

- `version`: pops up a window with version information.
- `access information`: pops up a window with the access code for the waveform server. This is needed for the ”shared shot” option Sec. 7.4.
- `waveform properties`: pops up a window with the ”waveform properties” of waveforms in the current phase. The user can change the XMIN, XMAX, YMIN, and/or YMAX values, or reset these to their default values. There are also two checkboxes that indicate whether the global XMIN and XMAX values are used instead of the ones for the waveform.
- `waveform groups`: pops up a window that modifies waveform groups. See Sec. 7.5.2 for more information about waveform groups.
- `set global X scales`: pops up a window that allows the global XMIN and XMAX to be modified. These global values are usually used for most or all waveforms. Setting one of these values would take effect immediately for all waveforms that use these values. The ”waveform groups” menu choice has two checkboxes to specify whether the global XMIN and XMAX are used for each waveform.

- `preferences`:pops up a window that allows preference settings to be modified. These settings include the initial settings of the checkboxes on the waveform editor (Sec. 7.5.4), the number of plots to display in the X and Y directions for multiplot plot mode, etc.
- `manual cycle control`: pops up a window that allows the user to start a shot or unlock and lock the shot. For stand alone cases this option allows a shot to be taken in simulation mode. In operations mode this option allows a shot to be unlocked before the "final lockout" has been sent so changes can still be made.
- `change access control`: pops up a window to allow changes to the access control. This choice only exists in operations mode. See Sec. 7.5.9 for more information about access control and this window.
- `show access control`: pops up a window showing whether the user has write privilege to each category. This choice only exists in operations mode. See Sec. 7.5.9 for more information about access control.
- `quit`: this choice closes the waveform editor.

7.5.1.2 Show menu

This menu choice chooses the items that are displayed in the list in the upper left of the waveform editor. The choices are the same as the checkboxes above the list box. See Sec. 7.5.4 for more about these choices.

7.5.1.3 Category menu

This menu option shows all the category choices. Choose one to select that category. The first shot phase, first group or subset, and first data item will also be selected.

7.5.1.4 Phase menu

- `modify the selected shot phase`: pops up a window that allows modifying the attributes of the shot phase. This includes setting the algorithm, changing the phase name (if it is not a permanent phase), setting the "entry time", and setting the anchor time if the phase is "anchored".
- `add a new shot phase`: pops up a window that allows the adding of a new shot phase. The phase name, algorithm, and entry time all need to be specified. If the phase is anchored, then the anchor time must be set as well.
- `delete the selected shot phase`: pops up a window that asks whether or not the phase should be deleted. The phase cannot be deleted if it is permanent or if it is used on any phase sequences.

- `shift vertices for the selected shot phase`: pops up a window that requests the value of "X shift", the value to shift all waveforms in the phase.

7.5.1.5 Defaults menu

- `load default vertices for selected waveform`: this choice fetches the default vertices for the waveform and displays them on the plot after deleting any existing vertices. The user still needs to press the "apply" button to update the vertices in the waveform server.
- `load defaults for selected phase`: pops up a window that asks whether or not to actually do this. This choice will set the vertices for all waveforms to their default values and reset all parameter data to default values. This choice is the same as re-choosing the algorithm in the `modify the selected shot phase` choice in the `Phase` menu.
- `load defaults for the selected category`: pops up a window that asks whether or not to actually do this. This option deletes any temporary shot phases and reinitializes the permanent shot phases for the selected category. In addition, the vertices for the phase sequences are set to their default values.
- `load defaults for all categories`: pops up a window that asks whether or not to actually do this. This is the most extreme action that a user can take! This choice reinitializes ALL categories, phases, phase sequences, waveforms, and parameter data.

7.5.1.6 Edit menu

- select all vertices: all vertices for the currently selected waveform are selected in the vertices list box and each vertex box on the plot display is changed to black.
- deselect all vertices: all vertices for the currently selected waveform are deselected in the vertices list box and each vertex box on the plot display is changed back to blue.
- add selected vertices to clipboard: add selected vertices from the currently selected waveform to the clipboard. Any vertices already on the clipboard remain on the clipboard. If a vertex being added has the same X value as a vertex already on the clipboard, a popup window asks whether the vertex should be replaced.
- add all vertices to clipboard: add all vertices from the currently selected waveform to the clipboard. Any vertices already on the clipboard remain on the clipboard. If a vertex being added has the same X value as a vertex already on the clipboard, a popup window asks whether the vertex should be replaced.

- copy selected vertices to clipboard: copy the currently selected vertices to the clipboard. Any vertices on the clipboard are first erased.
- copy all vertices to clipboard: copy all vertices to the clipboard. Any vertices on the clipboard are first erased.
- cut selected vertices (copy and delete): cut the currently selected vertices to the clipboard. Any vertices on the clipboard are first erased. The selected vertices are then deleted from the current waveform.
- cut all (copy all and delete all): cut all vertices to the clipboard. Any vertices on the clipboard are first erased. All vertices are then deleted from the current waveform.
- paste vertices from clipboard: paste the vertices from the clipboard onto the current waveform. If a vertex from the clipboard has the same X value as a vertex on the current waveform, then a popup window asks whether the vertex on the waveform should be replaced. Note that existing vertices on the waveform that do not match any X value of vertices from the clipboard are not affected.
- merge (paste replacing duplicate X values): merge the vertices from the clipboard onto the current waveform. If a vertex from the clipboard has the same X value as a vertex on the current waveform, then the vertex on the waveform is replaced. Note that existing vertices on the waveform that do not match any X value of vertices from the clipboard are not affected.
- delete all vertices: delete all vertices from the currently selected waveform.
- replace all (delete all and paste): delete all vertices from the currently selected waveform then copy all vertices from the clipboard.
- adjust vertices...: bring up a window that allows vertices to be adjusted. Options allow shifting or scaling either or both of the X and Y value of the vertices selected or all vertices for the waveform.
- plot vertices from clipboard: plot the vertices that are on the clipboard on the currently selected waveform plot. This choice does not add or delete vertices, it simply displays the vertices on the clipboard.
- clear clipboard: erase vertices on the clipboard.

7.5.1.7 Bookmarks menu

The bookmarks menu allow the user to bookmark the currently selected data item or to move to a data item marked by a previously saved bookmark. The first bookmark is usually "operating setup data".

- Add Bookmarks: add a new bookmark for the currently selected data item. This bookmark will go to the end of the list of bookmarks.
- Edit Bookmarks: brings up an editor to allow the bookmarks to be edited. The default editor can be set using the preferences choice under the File menu.

7.5.1.8 Restore menu

The restore menu option usually contains four items (the installation can add more if desired).

- reference plot: add a new bookmark for the currently selected data item. This bookmark will go to the end of the list of bookmarks.
- from archive: bring up a window that allows the user to restore from an archived setup for a shot. See Sec. 7.5.3.1 for more about the shot restore interface.
- from a prepared setup: bring up a window that allows the user to restore from a prepared setup or future shot. See Sec. 7.5.3.2 for more about the prepared setup restore interface.
- flush restore cache: this choice flushes the restore information held in the waveform server. The installation can configure the number of restore archives that are buffered (the default is two). This choice is mainly used to flush a prepared setup that was previously cached and then modified. After choosing this option, the prepared setup can then be brought up again.

7.5.1.9 Commands menu

The Commands menu is optional for an installation. The menu option will automatically be created if there are files in the data directory that end in ".com". A choice will be created for each file found using the name of the file minus the ".com" suffix. For example, if a file called "turn_gas_off.com" is found in the data directory, a Commands menu choice called "turn_gas_off" will be created.

These files may be created for the installation to do tasks that may be complicated or require multiple steps. The lines in one of these files can start with "IDL¿" to indicate that the remainder of the string will be executed using the IDL "EXECUTE" function. For example, variables can be set and procedures or functions can be called. Since each line is executed from the infrastructure procedure "modwave_event", any IDL variables in this procedure can be used. This includes the two structures given by the names "wavest" and "dx". Care must be taken not to redefine any of the IDL variables in the modwave_event procedure. Note that the "EXECUTE" function is disabled for the IDL virtual machine.

Note that any of the lines with the "IDL¿" prefix must adhere to IDL rules for non-compiled commands. So there are no if blocks or do loops allowed. However, a single if-then-else statement or for-do statement is allowed.

A command can be added that runs an IDL procedure that handles an installation specific action. For example, an installation procedure to do a morning checklist can be created that could be added as a Commands menu choice. The ".com" file would be called "checklist.com" and it would be placed in the data directory.

```
skip message  ;skip popup confirming selection of this command
IDL>resolve_routine,'checklist
IDL>checklist,dx,error
```

The special line "skip message" is a request to skip the popup message that normally is created which asks the user whether the actions should actually be done or not.

Other strings not beginning with "IDL¿" are passed through to the waveform server to be parsed as a change command. The syntax of these lines is the same as listed in the "list changes in current setup" choice under the Information menu (Sec. 7.5.1.10).

So simple commands can be created that would turn the gas on and off by switching the phase sequence in the Gas category.

```
;This turns gas on by selecting the ShotStart phase in
;the Primary phase sequence of the Gas category.
skip message  ;skip popup confirming selection of this command
SET_VERTICES,'Gas','','Primary',[-9.0],['ShotStart'],1,-9

;This turns gas off by selecting the NoGas phase in
;the Primary phase sequence of the Gas category.
skip message  ;skip popup confirming selection of this command
SET_VERTICES,'Gas','','Primary',[-9.0],['NoGas'],1,-9
```

The above example passes the "SET_VERTICES" string to the "execute_command" procedure found in the infrastructure which passes the entire string to the waveform server.

All lines in the file, whether they get executed by IDL or passed to the waveform server, are checked for an error. If an error occurs, then a message window will pop up and the rest of the command file will be aborted. The error can be ignored by simply setting a variable.

```
IDL> IGNORE=1
```

7.5.1.10 Information menu

The following are choices under the Information menu.

- `raw data problems`: pops up a window displaying any raw data problems or warnings from the waveform server. If there are any raw data problems, the "data" light on the waveform editor and the "data" light on the view log will both turn red. Warnings turn the lights yellow. Green lights indicate no raw data problems.

- `pcs pointnames dictionary`: pops up a window displaying the list of pointnames for all algorithms in all categories. Select the category, then the algorithm, then the type of pointname (targets, errors, shapes, etc.). Additionally, the cpu can be selected to filter only those pointnames that are created on that cpu.

- `list waveforms' vertices`: pops up a window requesting two times to use as a range to find vertices. Checkboxes allow the search to be for the currently selected phase, all used phases in the category, or all phases in the category. The "show" button will pop up a window displaying only those waveforms which have vertices between the two given times.

- `list changes in current setup`: pops up a window showing the changes made to the setup since the last time all categories were loaded or defaults for all categories were loaded. This window has several features.

  - `Find`: enter a string to find in the list. Those lines that match the string are highlighted. More than one word or phrase can be specified and lines that match any of the words or phrases will be highlighted. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.
  - `Filter`: enter a string to filter the lines in the list. Only those lines that match the string are displayed. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.
  - `dismiss button`: dismisses the window.
  - `save to file button`: saves the lines displayed to a file. A window will pop up requesting the name of the file to create.
  - `reload button`: resends the request for the information to the waveform server.
  - `execute button`: send the selected line to the waveform server to be "executed". This feature can be used, in some cases, to "undo" a change. The syntax of these lines should be followed if a command file is created that needs to change a waveform's vertices or a data item, etc. needs to be loaded from an archive or prepared setup. See Sec. 7.5.1.9 for more information about commands.
  - `brief checkbox`: the lines in the display may be difficult to see since most will be rather long. The "brief" option reduces the lines to the most important details.

The submenu `Diagnostics` is the last choice under the Information menu and has the following choices.

- `phase change lists`: pops up a window displaying the phase change lists for the selected phase. This includes the time (relative to the start of the discharge if the phase is used or relative to the start of the phase if the phase is not used), the type of change, the element number in the target vector (starting at 1), the element's pointname, and the values (e.g., slope and intercept) associated with that change.

  This window has several features.

  – `Find`: enter a string to find in the list. Those lines that match the string are highlighted. More than one word or phrase can be specified and lines that match any of the words or phrases will be highlighted. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.

  – `Filter`: enter a string to filter the lines in the list. Only those lines that match the string are displayed. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.

  – `dismiss button`: dismisses the window.

  – `save to file button`: saves the lines displayed to a file. A window will pop up requesting the name of the file to create.

  – `reload button`: resends the request for the information to the waveform server.

  – `brief checkbox`: each change list takes up multiple lines in the display. Use the "brief" option to show each change list on a single line.

- `phase sequence change lists`: pops up a window displaying the phase sequence change lists for the selected category. This includes the time (relative to the start of the discharge), the Y value of each vertex on each phase sequence (starting at 1), and the name of the phase that the Y value corresponds to.

  This window has several features.

  – `Find`: enter a string to find in the list. Those lines that match the string are highlighted. More than one word or phrase can be specified and lines that match any of the words or phrases will be highlighted. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.

  – `Filter`: enter a string to filter the lines in the list. Only those lines that match the string are displayed. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.

  - `dismiss button`: dismisses the window.
  - `save to file button`: saves the lines displayed to a file. A window will pop up requesting the name of the file to create.
  - `reload button`: resends the request for the information to the waveform server.

- `phase parameter data summary`: pops up a window that displays the full parameter data information for each block found in the currently selected phase. The block names are listed in ASCII order.

  This window has several features.

  - `Find`: enter a string to find in the list. Those lines that match the string are highlighted. More than one word or phrase can be specified and lines that match any of the words or phrases will be highlighted. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.
  - `Filter`: enter a string to filter the lines in the list. Only those lines that match the string are displayed. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.
  - `dismiss button`: dismisses the window.
  - `save to file button`: saves the lines displayed to a file. A window will pop up requesting the name of the file to create.
  - `reload button`: resends the request for the information to the waveform server.
  - `brief checkbox`: each parameter data block has multiple lines of information. Use the "brief" option to show each block on a single line.

- `phase parameter data (IO queue)`: pops up a window that contains a list of just the names of the parameter data blocks found in the currently selected phase in the IO queue (raw data only).

  One of the blocks can be selected by clicking on its name and then when the `show` button is clicked a new window displays the values in the parameter data block. The data are interpreted using the data object descriptor for the parameter data block. The descriptor contains only information about the data types and quantity so the data values are visible but not much information is available about the data organization from this window.

The parameter data blocks displayed by this menu choice are those stored in the portion of the waveform server that contains the most recent data. This list contains only raw parameter data blocks.

This window has several features.

- `Find`: enter a string to find in the list. Those lines that match the string are highlighted. More than one word or phrase can be specified and lines that match any of the words or phrases will be highlighted. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.
- `Filter`: enter a string to filter the lines in the list. Only those lines that match the string are displayed. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.
- `dismiss button`: dismisses the window.
- `save to file button`: saves the lines displayed to a file. A window will pop up requesting the name of the file to create.
- `reload button`: resends the request for the information to the waveform server.

• `phase parameter data (work queue)`: pops up a window that contains a list of just the names of the parameter data blocks found in the currently selected phase in the work queue (raw data and processed data).

  One of the blocks can be selected by clicking on its name and then when the `show` button is clicked a new window displays the values in the parameter data block. The data are interpreted using the data object descriptor for the parameter data block. The descriptor contains only information about the data types and quantity so the data values are visible but not much information is available about the data organization from this window.

  The parameter data blocks displayed by this menu choice are those stored in the portion of the waveform server that contains the most recent data. This list contains both raw parameter data blocks and processed parameter data blocks.

  This window has several features.

  - `Find`: enter a string to find in the list. Those lines that match the string are highlighted. More than one word or phrase can be specified and lines that match any of the words or phrases will be highlighted. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.

  - `Filter`: enter a string to filter the lines in the list. Only those lines that match the string are displayed. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.
  - `dismiss button`: dismisses the window.
  - `save to file button`: saves the lines displayed to a file. A window will pop up requesting the name of the file to create.
  - `reload button`: resends the request for the information to the waveform server.

- `memory usage summary`: pops up a window that displays how much memory is used on each real time computer and how that memory is distributed in several major sections of the real time heap.

  This window has several features.

  - `dismiss button`: dismisses the window.
  - `save to file button`: saves the lines displayed to a file. A window will pop up requesting the name of the file to create.
  - `reload button`: resends the request for the information to the waveform server.

- `RT processor memory layout`: pops up a window that displays how the memory is laid out on each real time computer. Use this diagnostic to see exactly what is in the real time memory and any additional memory regions.

  This window has several features.

  - `Find`: enter a string to find in the list. Those lines that match the string are highlighted. More than one word or phrase can be specified and lines that match any of the words or phrases will be highlighted. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.
  - `Filter`: enter a string to filter the lines in the list. Only those lines that match the string are displayed. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.

  - `dismiss button`: dismisses the window.
  - `save to file button`: saves the lines displayed to a file. A window will pop up requesting the name of the file to create.
  - `reload button`: resends the request for the information to the waveform server.

- `real time functions list`: pops up a window that displays each function that will be executed on each of the real time computers. The display shows the absolute time that a phase will execute, the function vector element, the name of the function, the category, the phase, and the algorithm. The display is sorted for each real time computer by the time.

  This window has several features.

  - `Find`: enter a string to find in the list. Those lines that match the string are highlighted. More than one word or phrase can be specified and lines that match any of the words or phrases will be highlighted. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.
  - `Filter`: enter a string to filter the lines in the list. Only those lines that match the string are displayed. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.
  - `dismiss button`: dismisses the window.
  - `save to file button`: saves the lines displayed to a file. A window will pop up requesting the name of the file to create.
  - `reload button`: resends the request for the information to the waveform server.

- `cpu info`: pops up a window that displays the virtual cpu for each category that is found on each physical real time cpu.

  This window has several features.

  - `Find`: enter a string to find in the list. Those lines that match the string are highlighted. More than one word or phrase can be specified and lines that match any of the words or phrases will be highlighted. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.

- `Filter`: enter a string to filter the lines in the list. Only those lines that match the string are displayed. The "+" character can be used to require that a word or phrase be in the line, and the "-" character can be used to ignore lines with that word or phrase. Surround phrases with quotes. Check the "match case" checkbox to match upper and lower case characters.
- `dismiss button`: dismisses the window.
- `save to file button`: saves the lines displayed to a file. A window will pop up requesting the name of the file to create.
- `reload button`: resends the request for the information to the waveform server.

• `users logged in`: pops up a window that displays the users who are currently logged in to the waveform server. The information includes the user id, the username, the host name, and the display name used for the user interface.

  This window has several features.

  - `dismiss button`: dismisses the window.
  - `save to file button`: saves the lines displayed to a file. A window will pop up requesting the name of the file to create.
  - `reload button`: resends the request for the information to the waveform server.

7.5.2 Waveform groups

One of the choices on the File menu of the waveform editor is "waveform groups". A waveform group is a set of data items that serve two purposes.

1. Vertices for all waveforms in a group can be modified if the vertices for any of the waveforms are modified. After changing the vertices for one of the waveforms and pressing the "apply" button, a popup window requests whether the vertices for all the other waveforms in the group should also be changed. If the user replies "yes", then all the waveforms in the group will end up with exactly the same vertices.
2. A set of data items can be grouped together so they are displayed as a subset. Waveform groups are listed first before the normal subsets of an algorithm so the first group will be the default when a phase is first displayed. Any group names are followed by three asterisks, "***". The grouping of these data items makes it easier to view and change the most commonly used data items in the algorithm.

The "waveform groups" window has the following features.

- `File menu`: choices allow for the reading of the waveform groups file, the saving of the current settings to a new file, the editing of the current file, and a choice to pick a file to edit. The final choice is to close the window. The name of the current wavegroups file is displayed below the menu bar.
- `Edit menu`: choices allow a group to be added, a group to be copied, a group name to be changed, and a group to be deleted.
- `display of groups`: the list of groups shows for each group the group name, whether it is shown in the subset list, whether it is used for changing vertices, the algorithm identifier, and the category identifier. Select a line to fill in the widgets below the list.
- `Data items in group`: lists the data items in the currently selected group. Select one or more and press the "remove from group" button below this list to remove data items from the currently selected group.
- `Data items in algorithm`: lists all data items in all subsets in the algorithm. Select one or more and press the "add to group" button below this list to add to the currently selected group.
- `GROUP`: shows the name of the group in a box. The name may not be edited. To change the name choose the "Rename group" choice under the Edit menu.
- `Show as subset`: check this box if the group should appear as a subset for that algorithm.
- `Use for changes`: check this box if the group should be enabled for changes. When the vertices for one waveform in the group are changed, the vertices for the other waveforms in the group can also be changed. The user will be prompted whether to do this or not.
- `remove from group`: button to remove selected data items from the currently selected group.
- `add to group`: button to add selected data items to the currently selected group.
- `Help`: this menu choice brings up a window describing how to use the interface.

7.5.3 Restoring data

This section further describes the choices for restoring from a shot, restoring from a prepared setup, and the restore graphical user interface.

7.5.3.1 Restore from a shot

The interface that restores from an archived shot consists of two windows. The first window requires the user to input a shot number. The user can then press one of three buttons.

- select data to load: this button will bring up a large interface that allows individual categories, phase sequences, phases, subsets, or data items to be restored. See Sec. 7.5.3.3 for more information.
- load sel category: this button will restore the selected category in the list above the button. The category is then loaded without the bigger interface being needed.
- load all categories: this button will restore all categories from the shot without the bigger interface being needed.

7.5.3.2 Restore from a prepared setup

The interface that restores from an prepared setup consists of two windows. The first window requires the user to select an existing prepared setup or specify a new name to create a new setup. Features of this interface include the following.

- list of user's setups: on the left is a list of the prepared setups that the user has already created. One can be selected by simply clicking on the setup name. The name of the setup is followed by its owner's name in parentheses.
- setup description: this box allows the user to add one or more labels to the selected setup. The labels help identify what is in the prepared setup.
- list user setups: a drop down list of usernames allows the user to select the prepared setups owned by someone else. The first item in the drop down list is "all users" which lists all prepared setups found.
- file filter: allows for the filtering of the setup names. An asterisk is used as a wildcard (match anything) character.
- delete setup: allows the user to delete one of his setups. The currently selected setup is deleted if this button is pressed.
- delete all backup files: setups are backed up every time a future shot waveform editor is created. This is done in case the setup file gets corrupted and the original version is needed. The number of backup files can grow quite large so this button can be used to delete these backup files.
- ignore backup files: this checkbox can be unchecked to show the backup files in the list.

- select data to load: this button will bring up a large interface that allows individual categories, phase sequences, phases, subsets, or data items to be restored. See Sec. 7.5.3.3 for more information.
- dismiss: deletes the window.

7.5.3.3 Restore graphical user interface

The restore graphical user interface has many features.

At the top are the following menus.

- File menu: a single choice to dismiss the window.
- Show menu: this menu has several choices.
  - show version: pops up a window giving the PCS version information stored with the archive.
  - show changes: pops up a window giving the change list information stored with the archive.
  - show phase history: (only for shot archives) pops up a window giving the phase history for the given shot. This information is stored in the archive in pointnames like PHASESTK1, PHASESTK2, etc., one for each cpu.
  - show phase seq history: (only for shot archives) pops up a window giving the phase sequence history for the given shot. This information is stored in the archive in pointnames like SEQSTK1, SEQSTK2, etc., one for each cpu.
- Load menu: this menu has several choices.
  - load algorithm: loads the algorithm used in the selected phase. (NOTE: this initializes the phase, it does not restore the phase).
  - load entry time: loads the entry time used in the selected phase. (NOTE: this does not restore the phase).
  - load anchor time: loads the anchor time used in the selected phase. (NOTE: this does not restore the phase).
  - load parameters: loads all the parameter data from the phase. (NOTE: this will initialize any data items that are static data items). Most parameter data is associated with a data item, but sometimes a parameter data block is not. This choice allows those types of blocks to be restored from the archive without restoring the entire phase. Use with caution!

Below the menu options there are five list boxes that display the following items.

- source categories: a list of categories in the source archive.
- phase sequences: a list of phase sequences in the source archive.
- phases: a list of phases in the source archive.
- data item subsets: a list of subsets in the selected source phase.
- data items: a list of data items in the selected source subset.

Below each list box are two buttons. In all but the last set, these buttons load a single item (category, phase sequence, etc.), or all the items. In the last set found under the "data items" list box the first button loads the selected data item. The second button allows the user to show the vertices for a waveform on the plot display so the vertices can be viewed before loading them.

Below the list boxes and buttons there is a region which has a dual purpose depending on the two checkboxes "show messages" and "show targets". If the "show messages" box is checked, then this region displays the messages from the load operations.

If the "show targets" box is checked, then this region displays five list boxes corresponding to the same items for the target, i.e., what is currently in the waveform server. In addition, there is a box where a different phase name can be entered when loading a phase.

The "show targets" region is useful when a user wants to load an item *into* another item with a different name. For example, the user can select the source phase and a different target phase, then press the "load selected phase" button. Or the vertices for the selected data item can be loaded into a different target data item.

By default, when a source item is selected, the same target item is selected if it exists. If it does not exist, then no item is selected in the matching list box, plus any others that would be affected, i.e., no subset or data item if the phase does not exist.

At the bottom of the restore user interface are several other regions.

To the left is a list of the currently chosen items in the source.

- Source category: the currently selected source category.
- Phase: the currently selected source phase.
- Algorithm: the algorithm used in the source phase.
- Entry time: the entry time used in the source phase.
- Data item descr: the descriptor for the currently selected data item for the source.

In the center are the two checkboxes that determines what gets displayed above (messages or target information), a colored status light that indicates the status of the last load operation, and a button to dismiss the window.

To the right is a list of the currently chosen items in the target.

- Target category: the currently selected target category.
- Phase: the currently selected target phase.
- Algorithm: the algorithm used in the target phase.
- Entry time: the entry time used in the target phase.
- Data item descr: the descriptor for the currently selected data item for the target.

7.5.4 Checkboxes on the user interface

There are two sets of checkboxes found on the user interface. The first set indicates what is displayed in the upper left list box.

- Ctgy: lists the categories.
- Seq: lists the phase sequences.
- Phs: lists the shot phases.
- Time: lists the time reference used in the plot. There is one time reference for each phase sequence in the category followed by a time reference for the phase itself. If the phase is used on the first phase sequence (usually called "Primary"), then the time reference is always with respect to the discharge or t =0. If the phase is used on another phase sequence, then the time reference is with respect to the start of that phase sequence which is usually unknown (caused by some event). If the phase is not used at all, then the time reference is respect to the beginning of the phase.
- Sub: lists the subsets. If any waveform groups have been defined that are set to "show as subset", then these groups will be listed first, before the algorithm's subsets.
- Data: lists the data items in the waveform group or subset.

Note that only one of these checkboxes can be set at one time.

The second set of checkboxes are mutually exclusive and appear above the "cancel" and "apply" buttons. The initial settings of these checkboxes can be changed with the preferences choice under the File menu.

- multiplot: Switch between single plot mode and multiplot mode. Use the preferences menu choice to set the number of plots in the X and Y direction.
- select: If this checkbox is selected, then a left mouse click near a vertex will select it, or a box can be drawn around vertices to select more than one. If the checkbox is not selected, then a left mouse click will begin to blow up the plot.
- add-drag: If this checkbox is selected, then a right mouse button press can be used to *drag* a vertex to another location. A plot line moves with the vertex as it is dragged.

7.5.5 Buttons on the user interface

The following buttons are found on the user interface.

- add: add selected vertices from the currently selected waveform to the clipboard. Any vertices already on the clipboard remain on the clipboard. If a vertex being added has the same X value as a vertex already on the clipboard, a popup window asks whether the vertex should be replaced.
- copy: copy the currently selected vertices to the clipboard. Any vertices on the clipboard are first erased.
- cut: cut the currently selected vertices to the clipboard. Any vertices on the clipboard are first erased. The selected vertices are then deleted from the current waveform.
- paste: paste the vertices from the clipboard onto the current waveform. If a vertex from the clipboard has the same X value as a vertex on the current waveform, then a popup window asks whether the vertex on the waveform should be replaced. Note that existing vertices on the waveform that do not match any X value of vertices from the clipboard are not affected.
- merge: merge the vertices from the clipboard onto the current waveform. If a vertex from the clipboard has the same X value as a vertex on the current waveform, then the vertex on the waveform is replaced. Note that existing vertices on the waveform that do not match any X value of vertices from the clipboard are not affected.
- replace all: replace all the vertices from the current waveform with the vertices that are on the clipboard.
- copy all: copy all the vertices from the current waveform onto the clipboard, erasing the clipboard prior to the copy.
- adjust...: bring up a window that allows vertices to be adjusted. Options allow shifting or scaling either or both of the X and Y value of the vertices selected or all vertices for the waveform.
- trace: after adding or deleting vertices, the trace button allows the plot to be redrawn in a different color before applying the changes.
- refresh: this button will refresh the plot display. This differs from `replot` in that it will not change a plot that has been blown up.
- replot: this button redraws the plot with its original scales. Use this button after the plot has been blown up.

- blowup: this button will blow up the plot. Pressing this button brings up a window that indicates to click the mouse in the first location, then a second window pops up that requests a second mouse click. The box formed by the two clicks becomes the new plot area that the waveform is plotted in.
- cancel: This button is used to cancel any changes to the waveform. Those changes are thrown away and the plot is redrawn.
- apply: This button sends the new list of vertices for the waveform to the waveform server. The vertices and plot are then redisplayed.
- get new: This button only appears when a change is made by someone else to the currently selected waveform. It is displayed in a red background that says "data unsynch'ed". This button will cause the new vertices to be reread from the waveform server.

7.5.6 The vertex region of the user interface

The lower left corner of the user interface is a region that displays the vertices for the currently displayed waveform and allows the user to make changes to those vertices.

At the top of this region is a droplist which has the following choices.

- `Numerics mode`: this is the default mode where the "new x:" and "new y:" entry boxes allow the user to enter a new vertex. Below these boxes are three buttons: add the vertex whose X and Y values are given, delete the selected vertex or vertices, or replace the selected vertex with the new X and Y values that are given.

  Vertices to be deleted can be selected by highlighting them in the vertex display box or by using the mouse to select the vertices (Sec. 7.5.8). Only one vertex can be replaced by a new vertex.

  One feature that is useful is the double-clicking of a vertex line in the vertex display box. This will load the vertex into the "new x" and "new y" boxes so the user can make a change.

  Also, math can be performed inside the "new x" and "new y" boxes. This is possible because the input string entered is passed into the IDL "EXECUTE" function and the result is what gets entered as the value. Note that the "EXECUTE" function is disabled for the IDL virtual machine.
- `Clipboard`: this mode displays the vertices that are on the clipboard. In addition, the "new x" and "new y" boxes turn into "adjust x" and "adjust y" boxes and the buttons turn into four arithmetic operations and a delete button. These functions allow mathematical operations on one or more vertices. The operations are performed on any vertices on the clipboard that are selected.

- `Sine mode`: this mode creates a sine wave of vertices. All the necessary values to create the sine wave need to be given.

  - start time (sec): the start time of the sine wave.
  - period (sec): the value of the period.
  - offset: the Y offset (the default is 0).
  - # periods: the number of periods.
  - p-p amplitude: the peak to peak amplitude. The square wave is centered at the offset value.
  - start phase (deg): the starting phase.
  - vertex interval (sec): the interval between vertices. Note that this value cannot be smaller than the shot phase tick time which is an installation specific value.

  To add the vertices for the sine wave choose "vertices" from the "options" droplist, then "add" from the submenu. To delete these same vertices, choose "vertices", then "delete" from the submenu.

  The number of vertices added is the "period" divided by the "vertex interval".

  Multiple "components" can also be set up. Choose "components" from the "options" droplist, then "add" from the submenu to add a new component. The new component will be a duplicate of the component that is currently visible. To delete a component, choose the component to delete using the "u" and "d" buttons, then choose "components" from the "options" droplist and "delete" from the submenu.

- `Pulse burst`: this is the same as "Sine mode" except that the wave created is a square wave instead of a sine wave. All the necessary values to create the square wave need to be given.

  - start time (sec): the start time of the square wave.
  - period (sec): the value of the period.
  - offset: the Y offset (the default is 0).
  - # periods: the number of periods.
  - p-p amplitude: the peak to peak amplitude. The square wave bottom is at the value of the offset.
  - pulse width: the width of the pulse.

  To add the vertices for the square wave choose "vertices" from the "options" droplist, then "add" from the submenu. To delete these same vertices, choose "vertices", then "delete" from the submenu.

Each cycle will have 5 vertices. Note that the shot phase tick is an installation dependent value which is the minimum resolution of a given time value.

- X = the start time of the cycle - 1 shot phase tick (Y value = offset)
- X = the start time (Y value = offset)
- X = the start time + pulse width - 1 shot phase tick (Y value = offset + p-p amplitude)
- X = the start time + pulse width (Y value = offset)
- X = the start time + period - 1 shot phase tick (Y value = offset).

The next cycle would begin at the last vertex.

Multiple "components" can also be set up. Choose "components" from the "options" droplist, then "add" from the submenu to add a new component. The new component will be a duplicate of the component that is currently visible. To delete a component, choose the component to delete using the "u" and "d" buttons, then choose "components" from the "options" droplist and "delete" from the submenu.

- `Triangle wave`: this is the same as "Sine mode" except that the wave created is a triangle wave instead of a sine wave. All the necessary values to create the triangle wave need to be given.

  - start time (sec): the start time of the triangle wave.
  - period (sec): the value of the period.
  - offset: the Y offset (the default is 0).
  - # periods: the number of periods.
  - p-p amplitude: the peak to peak amplitude. The triangle wave is centered at the offset value.
  - pause (sec): the pause between each cycle when there is more than 1 period.

  To add the vertices for the triangle wave choose "vertices" from the "options" droplist, then "add" from the submenu. To delete these same vertices, choose "vertices", then "delete" from the submenu.

  Each cycle will have 4 vertices.

  - X = the start time of the cycle (Y value = offset)
  - X = one quarter of the cycle (Y value = offset + 0.5 * p-p amplitude)
  - X = three quarters of the cycle (Y value = offset - 0.5 * p-p amplitude)
  - X = the end of the cycle (Y value = offset).

The next cycle begins at the end time + the pause.

Note that multiple components are not allowed for triangle waves.

- `File input`: this choice allows for the reading of a file that contains one or more vertices. The format of the file is simply two values per line, the X and Y values. Two entry boxes exist: one for the directory path (the user's login directory is the default) and one for the file name ("waveform.txt" is the default name).

  To read the vertices from the input file choose "vertices" from the "options" droplist, then "add" from the submenu. To delete these same vertices, choose "vertices", then "delete" from the submenu.

  Multiple "components" can also be set up. Choose "components" from the "options" droplist, then "add" from the submenu to add a new component. The new component will be a duplicate of the component that is currently visible. To delete a component, choose the component to delete using the "u" and "d" buttons, then choose "components" from the "options" droplist and "delete" from the submenu.

7.5.7 The plot region of the user interface

The plot region is used to display one or more waveforms. This section describes what is displayed. Though more than one plot can be displayed in `multiplot` mode, this section will describe the contents of the plot region for a single plot.

The plot region uses the information specified in the waveform structure described in Sec. 4.7. The plot region consists of the following.

- `title`: The name of the waveform followed by a description, e.g., "Rp: Major Radius". The waveform name and the description are part of the waveform structure.
- `Y axis label`: This label is on the left side of the plot and corresponds to the `yunits` value of the waveform structure.
- `X axis label`: This label is on the bottom of the plot and corresponds to the `xunits` value of the waveform structure.
- `plot scales`: The four plot scales are given in the waveform structure. Note that IDL may use these four values only as hints.
- `blue hashed area`: If the selected shot phase starts later or ends earlier than the XMIN and XMAX values of the plot, then the area before the start of the phase and the area after the end of the phase are plotted with a blue hash to indicate that these regions are not part of the currently selected shot phase.

- `yellow hashed area`: Each waveform can specify a minimum Y value in the waveform structure and a maximum Y value. If the waveform plot extends beyond either of these two values, then the lower or upper area is drawn with a yellow hash to indicate that values in these regions are not allowed.
- vertices: Each vertex is displayed with a box centered around its X,Y location.
- plot line: A line is drawn through the center of each vertex. If the first vertex is displayed on the plot, then the line extends to the left edge at the same Y value as the first vertex. Likewise, if the last vertex is displayed, then the line extends to the right edge at the same Y value as the last vertex.
- crosshairs: As the mouse enters the plot region, red crosshairs are drawn at the mouse location. The X,Y plot coordinates are displayed above the button rows in the `x,y:` box. When the mouse leaves the plot region, the crosshairs are erased.

7.5.8 Using the mouse buttons in the user interface

The mouse buttons in the user interface can be used to perform various actions. The actions performed by clicking a mouse button may depend on whether the `select` checkbox is checked or the `add-drag` checkbox is checked.

- If the `add-drag` checkbox *is not* checked, then right clicking on a waveform plot adds a new vertex at the location where the mouse button is released. If the `add-drag` checkbox *is* checked, then the plot will automatically be redrawn when the mouse button is pressed. If the mouse is moved, then the plot is redrawn as the mouse moves. A new vertex will be added when the mouse button is released. Note that if the mouse button is pressed and the mouse location is over an existing vertex, that vertex will be deleted. This results in the existing vertex being `dragged` to where the mouse button is released.
- Clicking the middle mouse button near an existing vertex will delete the vertex but only if the mouse button is released without moving the mouse. If the mouse is moved before releasing the button, then the plot will be blown up in the X direction from the location will the mouse button is pressed to the location where the mouse button is released. If the `add-drag` checkbox is checked, the plot between the first vertex on the left of the deleted vertex and the first vertex on the right of the deleted vertex is redrawn.
- If the `select` checkbox *is not* checked, clicking the left mouse button will blow up the waveform plot from the location where the mouse button is pressed to the location where the mouse button is released. If the `select` checkbox *is* checked, then any vertices found within the box created between the pressing and releasing of the mouse button will be selected, i.e., the vertices in the vertex list box will

be highlighted and the vertex squares will be redrawn in black. Selected vertices can then be cut, copied, or deleted.

– Double-clicking the left mouse button on a vertex in the list of vertices will put that vertex into the `new x` and `new y` boxes so either value can be easily modified.

7.5.9 Changing access control

The installation has the option of setting up access control for the operations version of the PCS by creating a file called "access_control.data" in the data directory. Please see the explanation of the contents of this file in the infrastructure file "waveserver.c" in the subroutine "create_access_control_param_blocks".

The access control user interface allows a privileged user to add or remove write access to any or all categories for a user. The user interface has the following features.

– category list: the display on the left shows the groups or users who are allowed to make changes for each category. Selecting one or more categories allows access to the two buttons below this list that allow a group or username to be typed in so it can be added to the list or removed from the list for the selected categories.

– groups/users: the display on the right shows all the usernames listed in the input access control file. These usernames may be put into groups. Selecting a group in the list shows on the left which categories that group has write access to. Selecting a username in the list shows which categories the user has write access to.

– add group/user to category: this button becomes enabled when a privileged user selects one or more categories in the category list. When this button is enabled, a group or username may be typed into the box to the button's right. Pressing the button will then add the group/user to the selected categories.

– remove group/user from category: this button becomes enabled when a privileged user selects one or more categories in the category list. When this button is enabled, a group or username may be typed into the box to the button's right. Pressing the button will then remove the group/user from the selected categories.

– Set current "leader": the text of this string depends on the definition of "LEADER_NAME" provided in the input access control file. To the right is a droplist of all the possible users who can make themselves the "leader". The leader automatically gets added to the first group given in the input file. This first group should always be given write access to all categories.

– Add user to group "group": the text of this string depends on the definition of "CURRENT_GROUP" provided in the input access control file. To the right is a droplist of all usernames found in the input file. Choosing a username and pressing this button will add the username to the first group which should allow write access to all categories.

– add user to group: selecting a group in the groups/users list will enable this button. A new username may be entered in the box to the right of this button. When the button is pressed, that username is added to the selected group. Note that this change is not permanent. If the waveform server exits, this change will be lost. Permanent changes must be made to the input access control file.

– remove user from group: selecting a username in the groups/users list will enable this button. When the button is pressed, that username is removed from the selected group. Note that this change is not permanent. If the waveform server exits, this change will be lost. Permanent changes must be made to the input access control file.

– add new group: a new group may be added to the groups/users list by entering the new name to the right of this button. Pressing the button will add the group. Note that this change is not permanent. If the waveform server exits, this change will be lost. Permanent changes must be made to the input access control file.

– load defaults: this button will cause the waveform server to reload the default settings from the input access control file. Whenever changes are made to the file, press this button.

7.6 Routines for the user interface

The application code author may need to create a custom user editor to provide the ability to edit the data contained in one or more parameter data blocks. The basic procedure is described in Sec. 2.11.8. This section provides definitions of the available utility routines.

7.6.1 alterparamdata

This procedure sends a set of parameter data blocks to the waveform server. The data in these blocks and their structure are used to replace the data and structure of the parameter data blocks with the same names held in the waveform server in a specified phase in a specified category.

The parameter data blocks are provided in an array of bytes in the format originally obtained from the waveform server (using `param_getdata`, Sec. 7.6.4). The array of bytes may contain blocks in addition to those that should be sent to the waveform server. Those that should actually be sent to the waveform server can either be specified by their complete names or by specifying the prefix name of a group of associated blocks (Sec. 2.11.4.3) (or a combination of the two). In the latter case all of the blocks in the group are sent.

Note that it is important that only parameter blocks containing data that have been modified should be sent to the waveform server (see Sec. 2.11.8.11).

Calling format:

```
alterparamdata, category, phase, host, port,$
 send_data, error,$
 groups=groups, names=names
```

Procedure arguments:

- `category`: On input, the integer category code for the category to which the data should be sent. Normally obtained from the `dx` structure that contains the current user interface context (Sec. 2.11.8.1).
- `phase`: On input, the integer identifier for the shot phase, within the category specified by `category`, to which the data should be sent. Normally obtained from the `dx` structure that contains the current user interface context (Sec. 2.11.8.1).
- `host`: On input, the name of the computer where the waveform server is executing. Normally obtained from the `dx` structure that contains the current user interface context (Sec. 2.11.8.1).
- `port`: On input, the integer port number where the waveform server is listening for input messages. Normally obtained from the `dx` structure that contains the current user interface context (Sec. 2.11.8.1).
- On input, `send_data` is the parameter data byte array.
- On output, `error` is set to: 0 if the data were sent successfully to the waveform server, and no keywords were specified or all of the required parameter data blocks were located, 1 if there was a problem communicating with the waveform server, or 2 if any of the specified data blocks could not be located. It is expected that the programmer has specified a set of parameter blocks which is consistent with the design of the algorithm and if any of those blocks is not present this indicates a programming error.

Keywords:

- `groups`: A string array specifying one or more name prefixes (Sec. 2.11.4.3) for groups of associated blocks to be sent. That is, any block that has a name beginning with a character string provided in this array is sent. Case insensitive comparisons are made.

- `names`: A string array specifying one or more complete block names. Any block with a name that completely matches one of the strings in the array is sent. Case insensitive comparisons are made.

Neither, one or both of the keywords `groups` and `names` can be specified. If neither the keyword `groups` nor the keyword `names` is provided in the procedure call, all of the parameter data blocks present in the input byte array are sent.

Note that since it is required that only parameter blocks containing data that have been modified be sent to the waveform server, it is unlikely that a complete group of blocks will need to be sent. Usually, the `names` keyword will be used to specify the list of modified parameter data blocks to be sent to the waveform server.

7.6.2 extract_paramdata

Given an array of bytes containing one or more parameter data blocks obtained from the waveform server (using `param_getdata`, Sec. 7.6.4), this procedure extracts the specified parameter data block and returns it in an IDL variable. The IDL variable is returned with the proper format to hold the data from the parameter block. A combination of the information in the parameter data block descriptor (Sec. 2.11.4.5) and an optional model variable provided to the `extract_paramdata` procedure is used to format the IDL variable.

Calling Sequence:

```
err = extract_paramdata(blockdata, name, outdata)
```

Input variables:

- `blockdata`: A byte array containing the parameter data (obtained from the waveform server using `param_getdata`, Sec. 7.6.4).
- `name`: A string giving the name of the parameter block to return. Case insensitive.

Output variables:

- `outdata`: the output variable to receive the returned data. Any value this variable had on input to the procedure is overwritten. (Note that `outdata` cannot be specified to be an element of a structure because in this case the variable is passed by value and cannot be overwritten by the procedure.)
- `err`: 0 if successful, 1 if the named block cannot be located, 2 if there is an unknown data type encountered in the parameter data block descriptor (this indicates a programming error), 3 if there is an inappropriate combination of keywords and arguments, 4 if the model provided doesn't match the data in the parameter block.

Optional keywords:

- `model`: specifies a variable (use the form `model=variable` in the procedure call) that should be used as a model for formatting the parameter data.

  The parameter data block content is expected to have the type and size to match an array of variables of the model type. For instance, the model could be a 2 dimensional array that is repeated more than once in the parameter block. The returned data will be a 3 dimensional array with the third dimension sized to match the data found in the parameter block. Or the input model could be a structure with tag names appropriate for the particular block. If the structure is repeated in the parameter block more than once, an array of structures will be returned.

  The parameter data descriptor is tested against the input model to make certain that the parameter block really matches it and to determine the number of times the model is repeated in the block.

  If the model is not repeated in the parameter block more than once, the dimensions of the output variable are the same as the dimensions of the input model.

- `/structure`: specifies that if `model` is not specified, the output variable is returned as a structure even if there is only one descriptor for the parameter data block. Normally, a single descriptor will yield a scalar or array.

- `strsize`: integer, specifies the size of a string. See the discussion about strings below for more information.

There are several useful ways to use this routine to return an IDL variable containing the parameter block data in a format easily useful in IDL code.

1. If the parameter data block is expected to contain a one dimensional array of values of a single data type, the procedure can be called without any keywords to obtain a variable defined to be a one dimensional array of the appropriate type containing the data.

2. If the block contains more than one array of data, each specified with its own descriptor, the procedure can be called without any keywords to obtain a variable defined to be a structure with one element per descriptor. Normally the IDL syntax for referencing a structure element by its index, `structure.(i)`, would be used with the resulting variable to access the ith element of the structure. (The tag names are arbitrarily set to the structure element index beginning with 1.)

   If there is only one descriptor for the parameter block, add the `/structure` keyword to obtain a structure with just one element.

3. If it is known that the block contains an array of structures (or a single structure) and it is desired to reference the structure elements with particular tag names, first define an example of the structure containing dummy data using the desired tag names to serve as a model for the procedure to follow. Then, pass this structure to the procedure with the `model` keyword to obtain an array of structures with the desired tag names containing the data from the parameter block. (If the parameter block contains only one instance of the structure, a scalar variable will be returned.)

   Note that the variable type of the model is compared to the descriptors for the parameter block, and if the parameter block cannot be made to match an array of the model variable, the `err` return value is 4 to indicate an error.

   Also note that if a model input variable structure is provided, in each element of the structure which is an array (independent of the variable type), the number of elements in the array must be known before the model structure is created.

   It is not possible to obtain a structure that has another structure as one of its elements.

4. If the parameter data block is expected to contain a multi-dimensional array of values of a single type, define a model variable with the correct dimensions and pass it as a model to the procedure using the `model` keyword. The parameter data descriptor will be compared to this model array, and if they match, the parameter block data will be copied into the input array. In this way the one dimensional array stored in the parameter block can be converted easily to the actual multi-dimensional array that is the format for the data that matches the intended application.

   Alternatively, the model can contain one dimension fewer than the expected dimension of the parameter data array and the procedure will determine the final dimension by counting the number of times the input model array is repeated in the parameter data block.

Strings and byte arrays

An array of characters contained within a parameter data block could be either an array of strings or an array of byte data.

- By default, an array of characters is assumed to be an array of strings.
- To force an array of characters to be returned as an array of bytes, provide a model variable that has byte type.

Given an array of characters, there are 2 possible ways that strings could be stored within the array.

1. As a concatenated array of null terminated strings. In this case, a descriptor specifies an array of characters and it is expected that the character array is composed of one or more strings, each terminated with a null character.

To extract this data as an IDL string array, there are three options.

- The procedure will determine the number of strings present and return a string array if no keywords are provided.
- A string array model can be provided. If the model array is replicated more than once in the parameter data block, then an array with an additional dimension of the correct size is returned.
- If an input model structure is provided, specify the appropriate element of the model structure as a string array containing the correct number of elements. This means that information about the number of strings must be stored in another parameter data block or must be known in some other way.

If the number of null terminated strings located is less than the number of elements in the string array provided in the model `err` is returned equal to 4 because the parameter data and the model do not match. If there are more null terminated strings in the parameter data block than in the array in the model, an array with an additional dimension is returned if possible, otherwise `err` is returned equal to 4 because the parameter data and the model do not match.

2. As a concatenated array of strings each with the same length.

   To extract this data as an IDL string array, do the following.

   - If a model structure is provided, specify the appropriate element of the model structure as a string array containing the correct number of elements. This means that information about the number of strings must be stored in another parameter data block or must be known in some other way.
   - Specify the string size in one of these two ways.
     - Specify `STRSIZE=-1`. In this case a model must be provided. The number of characters in the array belonging to each string is determined from the descriptor size divided by the number of elements in the string array provided in the model.
     - Specify `STRSIZE` equal to the number of characters in each string. No model is necessary.

   A null character, if present, ends each string. If a null character is present, it is included in the count of characters in each string. If a given string is terminated by a null character, the remaining characters in the array belonging to that string are ignored.

If more than one string array is contained within the parameter data block, all of the arrays must be specified using the same technique. In the model structure, provide an element for each expected string array.

7.6.3 param_finddata

Given an array of bytes containing one or more parameter data blocks (usually obtained from the waveform server using `param_getdata`, Sec. 7.6.4), this procedure extracts the specified set of parameter data blocks. The required parameter data blocks can either be specified by their complete names or by specifying the prefix name of a group of associated blocks (Sec. 2.11.4.3). In the latter case all of the blocks in the group that are present in the array of bytes provided as input to the routine are returned.

Calling format:

```
param_finddata, block_data, error, groups=groups, names=names
```

Procedure arguments:

- On input, `block_data` is the parameter data byte array from which the specified data blocks should be extracted. On output, the input data will have been replaced with an array of bytes containing only the specified data blocks.
- On output, `error` is set to: 0 if no keywords are given or if all of the required parameter data blocks are located, or 1 if any of the specified data blocks could not be located. It is expected that the programmer has requested a set of parameter blocks which is consistent with the design of the algorithm and if any of those blocks is not present this indicates a programming error.

Keywords:

- `groups`: A string array specifying one or more name prefixes (Sec. 2.11.4.3) for groups of associated blocks to be returned. That is, any block that has a name beginning with a character string provided in this array is returned. Case insensitive comparisons are made.
- `names`: A string array specifying one or more complete block names. Any block with a name that completely matches one of the strings in the array is returned. Case insensitive comparisons are made.

This routine replaces the input parameter data byte array with a new byte array that only contains the blocks that match the group(s) or name(s) specified in the keywords. Name matching is case insensitive. An associated group of blocks (Sec. 2.11.4.3) is specified using the common set of characters beginning each block name. For example, if an argument to the keyword such as `group = "M Matrix:"` would match any block name that begins with "M Matrix:".

If neither the keyword `groups` nor the keyword `names` is provided in the procedure call, the input parameter data byte array is returned unchanged.

7.6.4 param_getdata

Obtain from the waveform server a specified set of parameter data blocks from a specified category and phase. The required parameter data blocks can either be specified by their complete names or by specifying the prefix name of a group of associated blocks (Sec. 2.11.4.3) (or a combination of the two). In the latter case all of the blocks in the group are returned. The only parameter data blocks that are returned are those that have specified `"USER"` in the options argument when the block was created (Sec. 2.11.4.2).

Calling format:

```
param_getdata, category, phase, host, port,$
 reply_data, error,$
 groups=groups, names=names
```

Procedure arguments:

- `category`: On input, the integer category code for the category from which the data should be returned. Normally obtained from the `dx` structure that contains the current user interface context (Sec. 2.11.8.1).
- `phase`: On input, the integer identifier for the shot phase, within the category specified by `category`, from which the data should be obtained. Normally obtained from the `dx` structure that contains the current user interface context (Sec. 2.11.8.1).
- `host`: On input, the name of the computer where the waveform server is executing. Normally obtained from the `dx` structure that contains the current user interface context (Sec. 2.11.8.1).
- `port`: On input, the integer port number where the waveform server is listening for input messages. Normally obtained from the `dx` structure that contains the current user interface context (Sec. 2.11.8.1).
- On output, `reply_data` is the parameter data byte array obtained from the waveform server.
- On output, `error` is set to: 0 if the data were obtained successfully from the waveform server, and no keywords were specified or all of the required parameter data blocks were located, 1 if there was a problem communicating with the waveform server, or 2 if any of the specified data blocks could not be located. It is expected that the programmer has requested a set of parameter blocks which is consistent with the design of the algorithm and if any of those blocks is not present this indicates a programming error.

Keywords:

- `groups`: A string array specifying one or more name prefixes (Sec. 2.11.4.3) for groups of associated blocks to be returned. That is, any block that has a name beginning with a character string provided in this array is returned. Case insensitive comparisons are made.
- `names`: A string array specifying one or more complete block names. Any block with a name that completely matches one of the strings in the array is returned. Case insensitive comparisons are made.
- `workqueue`: By default the parameter data are retrieved from the waveform server's input/output handler's copy. Set this keyword to 1 (or use the format `/workqueue`) to request data from the waveform server's queue handler database. For an explanation of these two data sources see Sec. 2.11.8.9.

Neither, one or both of the keywords `groups` and `names` can be specified. If neither the keyword `groups` nor the keyword `names` is provided in the procedure call, all of the parameter data blocks for the specified phase that had `"USER"` in the options argument when the block was created (Sec. 2.11.4.2).

7.6.5 replace_paramdata

Given an array of bytes containing one or more parameter data blocks obtained from the waveform server (using `param_getdata`, Sec. 7.6.4) this procedure replaces the data for a specified parameter block. New descriptors are created using the data type information from the IDL input variable, but the other information describing the block (flag values, alignment, and storage order index, is unchanged.

This routine provides the capability to radically change the structure of the parameter data from what was there when the data were received by the user interface. However, normally the only change would be to change the number of elements in an array (of structures or arrays). The structure of an individual element of the array normally wouldn't change.

Calling Sequence:

```
err = replace_paramdata(blockdata, name, indata)
```

Input variables:

- `blockdata`: A byte array containing the parameter data (obtained from the waveform server using `param_getdata`, Sec. 7.6.4).
- `name`: A string giving the name of the parameter block to replace. Case insensitive. The specified block must exist already.

- `indata`: The data to be inserted into the specified block. This is an IDL variable. The type and format of the data in the variable are recorded in the descriptor written into the parameter data block.

Output variables:

- `err`: integer, 0 if successful, 1 if the specified parameter block cannot be located.

Optional keywords:

- `strsize`: integer, specifies the size of a string. See the section on strings below for more information.

The input data variable (`indata`) can be a single scalar value, an array, a structure or even an array of structures. (A structure cannot have another structure as one of its elements.)

Replacing strings

Both an array of byte values and an array of strings are stored as an array of C language character values.

An array of strings can be stored in 2 ways.

1. As a set of null terminated strings concatenated into a single character array. This is the default.

2. As a concatenated array of strings of identical length. To choose this option specify the `strsize` keyword. If `strsize` is specified as a value greater than 0 then that value is used as the length of each of the strings. Strings larger than this size will be truncated. Strings shorter than this size will be padded with null characters.

   If `strsize=-1` is specified, then the common string length is set to the length of the longest string in the array plus 1. The extra character is filled with a null character to terminate the string.

Chapter 8

Building the PCS code

This section discusses the procedure for building the PCS code.

8.1 The first build of the PCS

This section describes how to get set up to do PCS code development and how to build the PCS code. The simplest procedure is to try the example in this section. However, this example procedure assumes that all of the default settings for the build procedure are appropriate. If this isn't the case, consult Sec. 8.3 for a description of the ways to set options for building the PCS code.

A developer of PCS code will typically have a directory tree in which the development work is done. The exact organization of this area will depend on the type of source code control system that is in use. However, something in common with all setups will be that there is a separate directory where the infrastructure code is built and a directory where the installation-specific code is built.

Here is an example set of commands to create the necessary directories when SCCS is used as the source code control system (as is the case at General Atomics for DIII-D).

In this example, there is a central PCS directory tree at `/pcshome/pcs` that is the master storage location for PCS code. The directory `/pcshome/pcs/src` includes subdirectories `install` and `infra`. Under each of these directories is a `SCCS` directory which contains the source code control files for the installation-specific code and infrastructure code, respectively.

To get started, create a master working directory (e.g. `pcs` in your home area) and create a soft link to the master SCCS area for the installation-specific code.

```
cd
mkdir pcs
cd pcs
ln -s /pcshome/pcs/src/install/SCCS ./SCCS
```

In in the `pcs` directory create the directory in which the infrastructure will be built. Then create a link to the master SCCS area for the infrastructure code.

```
mkdir infra
cd infra
ln -s /pcshome/pcs/src/infra/SCCS ./SCCS
```

Alternatively, the infrastructure directory can be located elsewhere and a link to it placed in the installation-specific area.

```
cd $HOME
mkdir infra
cd infra
ln -s /pcshome/pcs/src/infra/SCCS ./SCCS
cd $HOME/pcs
ln -s $HOME/infra ./infra
```

(This example assumes that the environment variable `HOME` is defined to be equal to the developer's home directory.)

Set the default directory to the infrastructure area and build the infrastructure code using the command `make`.

```
cd $HOME/pcs/infra
make
```

Then, go to the installation-specific area and make the installation-specific portion of the PCS.

```
cd $HOME/pcs
make
```

Note that the files created by the build of the infrastructure are used as inputs during the build of the installation-specific code. So, the build must be done first in the infrastructure area. Also, after the first time the infrastructure code is built, whenever the installation-specific code is rebuilt, the infrastructure code will also be automatically built in order to acquire any changes that were made.

The files that are the product of the build procedure are placed in a subdirectory named after the computer architecture on which the build takes place, e.g. `SUN_DIR`, `INTEL_DIR`, `ALPHA_DIR`, etc. There are separate subdirectories in the infrastructure and the installation-specific areas. It is the set of files in the subdirectory in the installation-specific area that is the final product of the PCS build.

8.2 Overview of the make procedure

The `Makefile` found in the installation directory is the same for all installations. It is an outline of the build procedure and simply includes other files, some from the infrastructure and some from the installation. The installation only has to provide three files. In the infrastructure: `makedefs_HOST` (described in Sec. 8.3). In the installation: a second version of `makedefs_HOST` and `makeinstall` (described in Sec. 8.5). This allows the build for the default programs of the PCS to be specified in the infrastructure, and it allows general changes to be made without all PCS installations having to update their installation files after getting a new version of the infrastructure.

8.3 Make procedure options

The default settings for the PCS build procedure are normally configured so that the build simply requires typing `make`. The default settings are located in a file called `makedefs_HOST` in both the infrastructure and the installation areas. Here `HOST` is the name of the computer where the build is being performed (as determined from the `hostname` command). This file is the place where the system-specific definitions required by the PCS build procedure are located. This method of configuring the `make` procedure allows the proper definitions to be automatically located on any given computer without the developer having to specify options on the `make` command line.

It is not essential that the `makedefs` file be named after the output of the `hostname` command. The extension of the `makedefs` file can be any arbitrary string. In this case, the extension of the file must be specified on the `make` command line as in this example.

```
make HOST=example_configuration
```

The `make` procedure will use the file called `makedefs_example_configuration` to obtain its default set of definitions.

Any of the definitions in the `makedefs` file can be overridden by specifying another definition on the `make` command line.

```
make OPTION1=definition1 OPTION2=definition2
```

Here are some of the most commonly used command line options.

- `INFRADIR`: The directory where the infrastructure files are located. By convention, the infrastructure directory (or a link to the directory) is located within the installation-specific directory and is called `infra`.

- `NCPU`: Set this option to indicate the number of real time processors configured into the PCS. This tells the `make` procedure how many executables to build for the real time processors.

- `MODEDEF="-Doption1 -Doption2"`: This option is used to provide preprocessor definitions for the compiler command line. This option is useful if, for example, some non-standard `#ifdef` blocks are used to include or exclude some particular code.
- `OUTDIR`: This option indicates the name of the subdirectory into which the files that are the product of the build procedure should be placed. Normally this is a directory named after the computer architecture, such as `SUN_DIR`, `INTEL_DIR`, `ALPHA_DIR`, etc. The default value is `PCS_DIR`.
- `HOST`: Use this option to specify the extension of the `makedefs` file from which the `make` procedure should obtain the necessary system-specific definitions, as described earlier in this section.

For a complete list of the possible options and the default values, examine the `Makefile` and `makedefs` files in the installation-specific and infrastructure directories.

8.4 The installation Makefile

This section describes the contents of the installation version of the `Makefile` which is the same for all installations. After setting the value for `INFRADIR`, the following files are included in this order:

- `makedefs_HOST` includes definitions for the specific `HOST` as described in Sec. 8.3. This file could include, for example, the definition for `OUTDIR` whose default value is `PCS_DIR`, or the definition for CC whose default value is `gcc`. It can also change the definition of `INFRADIR` if desired. It must contain a definition for `PCSLOCATION`. No make targets are allowed in this file.
- `$(INFRADIR)/../makeinstall_part1` includes all default definitions. See this file for more information. Any of these definitions can be overridden in either the `makedefs_HOST` file or the `makeinstall` file.
- `makedefs_HOST` is included again to allow the installation to override the defaults that were just set in the `makeinstall_part1` file.
- `$(INFRADIR)/../makeinstall_part2` includes several targets including the `all` target which is the default and targets like `host_only` and `real_only` to build only parts of the PCS. These targets are lists of definitions, which, in turn, are lists of targets. These definitions all have default values that were set in `makeinstall_part1` and can be added to or reset by the installation in the `makedefs_HOST` and `makeinstall` files.
- `makeinstall` includes all the definitions which are not host or platform dependent and any additional installation targets. See Sec. 8.5 for more information.

- `$(INFRADIR)/../makeinstall_part3` includes all the default targets for all the default PCS programs.

8.5 Adding to the make: modify makeinstall

The only file that needs to be updated to add a target to the PCS is the `makeinstall` file. Simply include the make code to handle the target, then add the name of the target to one of the "install" targets listed below. These targets must be included and can be empty.

- `make_setups_install`: targets that are related to the setup of the PCS
- `required_install`: required targets for the installation
- `idl_install`: idl related targets
- `host_install`: targets relating to the host_cpu programs
- `realtime_install`: targets relating to the real time programs
- `realtime_lo_install`: targets relating to the loadable realtime programs which are built differently than the realtime programs
- `realtime_other_install`: other targets needed for realtime
- `other_install`: all other targets that are required by the installation
- `clean_install`: files to delete when the clean target is chosen
- `user_install`: targets which are not part of the general build

The definitions for the target dependencies need to be appended to in this file. For example, the definition `INSTALL_INCLUDES` needs to include installation files for categories and algorithms other than the default acquisition and system categories which were added in `makeinstall_part1`. Append to the definition using `+=` instead of `=`. For example, to add a category called `coils` and an algorithm called `coils1` which uses an IDL procedure to edit a parameter data block, append to the proper definition the names of the files:

```
INSTALL_INCLUDES += coilscategory_master.h coilsalgorithms.h coils1_master.h
PARAMIDL_FILES += coils_editor.pro
```

Installation-specific definitions can also be put here when they differ from the default values given in the infrastructure. For example, the default value for `NCPU` is 1. If the installation wants to build a 3 cpu version by default, then add the following line:

```
NCPU = 3
```

8.6 Making a production build of the PCS code

Chapter 9

Starting the PCS for testing or normal operations

There are several ways in which the PCS can be started. The method to use depends on the intended type of PCS operation. In all cases one of a group of available scripts is used to start the PCS.

1. "Stand alone mode" allows testing of the PCS code on a computer platform other than the actual PCS real time processors. To start the PCS in this mode use the script `runsa`. The script starts all of the PCS processes and a user interface. When the user interface is closed, the script stops all of the PCS processes and exits. See Sec. 9.1 for more information.

2. To test execution of the PCS in future shot mode use the script `runtest`. The script asks whether "next shot" processes should be started. Answer "n" for future-shot-only execution. . The script starts just the user interface. Then, the user can use the "control" menu to start a future shot session. This starts a waveform server process. When the user interface is closed, the script exits.

3. To test execution of the PCS on the real time processors use the script `runtest`. The script asks whether "next shot" processes should be started. Answer "y" for operation on the real time processors. Usually this script must be executed on the host processor for the real time system. The script starts all of the PCS processes and a user interface. When the user interface is closed, the script stops all of the PCS processes and exits. See Sec. 9.3 for more information.

4. To start the version of the PCS code that has been installed for normal operations use the script `startpcs`. To stop the normal operations execution use the script `stoppcs`. These scripts start or stop all PCS processes, but do not start a user interface. The PCS operator uses the `run_wave` script to start a user interface that connects to the

already executing PCS processes used for normal operations. To check the status of the PCS processes the `checkpcs` script can be used.

The scripts and the associated PCS operation are discussed in this section.

9.1 Stand alone mode

9.1.1 Overview

Much of the PCS control algorithm code can be tested on a platform other than the actual real time processors since most code is involved with calculating control commands rather than interacting directly with the tokamak hardware. This type of testing is done in conjunction with the simulation server that provides test input data to the PCS. The simulation server also receives the command output from the PCS so that it can, if desired, calculate simulated plasma response based on the control commands.

The principal code that cannot be tested without executing on the real time hardware is anything that deals specifically with the PCS hardware such as data acquisition and command output. Other code, though, that calculates control commands can be executed and the results can be saved to determine whether the control code is working properly.

This testing method is convenient when the real time processors are in use for other testing or for tokamak operations. This mode of operation that allows the PCS to be executed on any computer for testing is called "stand alone mode."

In stand alone mode, the complete PCS executes. However, the code that would normally run on a real time processor is executed in a process on the host computer. If the PCS is configured to have multiple real time processors, the code for each "real time" processor is executed in a separate process.

Normally, in stand alone mode all of the PCS processes would execute on a single computer. A single large shared memory region is established that contains all of the memory that would exist on the real time processors. Each CPU master file for a real time processor indicates the maximum about of memory that is available on that processor. The sum of all of these memory sizes for all real time CPUs configured into the PCS is the amount of memory allocated into this shared region. Since the memory is shared, the communication between CPUs can be simulated by one "real time process" writing directly into the memory assigned to another CPU. This emulates the usual method for communication between the real time processors (see Sec. 2.9.2).

In stand alone mode, the idea is that the exact code that would run in real time is executed with the exception of a few small modules.

- Code associated with acquiring input data from digitizers etc.
- Code that sends commands to output hardware.

- Code that implements communication between real time processors (Sec. 2.9.3).

These modules would normally be rarely changed so that frequent testing wouldn't be necessary. Thus, stand alone mode allows most necessary testing of PCS code to be done without access to the actual PCS real time hardware.

Because the "real time code" is run on a different computer system from that of the PCS real time hardware, a special executable is required. When the PCS code is built, then, the real time code is compiled into both the file containing the code to be loaded into the real time processor and the executable for the host processor for stand alone mode.

Usually, for stand alone mode there would be substitute code for the operations that depend on execution on the actual PCS real time processors, such as the three types of operations listed above. Special code for stand alone mode would be implemented by using preprocessor `ifdef` statements to distinguish the code that should be compiled. Code that must distinguish between whether the build is for the real time processor or the stand alone mode executable can test the preprocessor macro `STANDALONE` to see if it is defined.

```
#ifdef STANDALONE
/* Code for stand alone mode. */
#else
/* Code for the real time processor. */
#endif
```

9.1.2 Host setup for stand alone mode

In order to use a given computer for PCS testing in stand alone mode a few customizations are required.

- The operating system must usually be adjusted to allow for a fairly large amount of memory to be used as shareable memory areas. The amount of memory required for each user executing a stand alone mode test is the total memory that would be available in all of the real time processors (see Sec. 9.1.1). Normally it is desirable to provide enough memory so that several users can be testing simultaneously.

- The standard PCS directories must be configured. See Sec. 2.18 for a description.

- Any input data files required by the PCS must be available.

- The standard method to archive PCS data after a shot must be available. Ideally, the user could specify one of a special range of shot numbers that would indicate a test shot that doesn't need permanent archival (e.g. for DIII-D, any shot numbers larger than 900000 are considered by the data archival system to be test shots). Also, the standard tools used to view archived data must have access to the results from the stand alone mode test shot.

- The PCS uses a number of network ports for communication between its various processes. Most operating systems do not quickly delete these ports (or mark them as unused) once the PCS processes exit. There is normally a timeout period first. During PCS testing the PCS might be started and stopped often. The port timeout period often inhibits the rate that this can be done. It is usually desirable to determine how to shorten the operating system's timeout period for making unused ports available again.

9.1.3 The startup script

To execute the PCS in stand alone mode, use the script `runsa`. Normally the script is executed without any options. To find the available options, read the comments at the beginning of the script file.

The script is located in the directory into which the installation-specific executable code is placed when it is built. For instance, following the example in Sec. 8.1, the script would be `/pcshome/pcs/SUN_DIR/runsa` on a Sun Solaris system. The script uses the file `pcs_config_info_empty` to obtain default configuration information.

The `runsa` script calls the `runtest` script (Sec. 9.3) to start many of the PCS processes so the options for that script are also relevant. See the comments at the beginning of the `runtest` script.

The `runsa` script can be executed with the default directory set to any directory. The PCS executables used are always those in the same directory where the `runsa` script is located. Thus, any build of the PCS can be executed in stand alone mode. For instance, any user can execute in his own directory area the development version of another user.

The `runsa` script creates a directory called `./runsa_dir` to contain files associated with its execution. Within this directory, there is a subdirectory for each PCS process. Each process is executed with its subdirectory as the default directory so that any core file for that process is written to its subdirectory so that the core file can easily be associated with the process creating it.

The waveform server process stores its `Nextshot.wa10` file in its subdirectory rather than using the central PCS archive area (see Sec. 2.18). However, any future shot files used by the `runsa` execution are still created or read from the central PCS archive area.

9.2 Testing in future shot mode

The `runtest` script is used to start the PCS in order to test a development build of the code in future shot mode. When adding a new algorithm, for instance, future shot mode is usually the best place to start because future shot mode doesn't require the real time processors (and so is quicker to execute) and it allows testing of the algorithm code that executes in the waveform server and the user interface. This code in the waveform server and the user interface is usually a large part of the implementation of a control algorithm and so requires testing. The real time code cannot be successfully tested until the waveform server code is working properly.

The `runtest` script is located in the directory into which the installation-specific executable code is placed when it is built. For instance, following the example in Sec. 8.1, the script would be `/pcshome/pcs/SUN_DIR/runtest` on a Sun Solaris system. The PCS executables used are always those in the same directory where the `runtest` script is located.

Normally the script is executed without arguments. However, there are options that can be used to customize the script execution. See the comments at the beginning of the `runtest` script for descriptions of the options. Among the available options, there is a very fast shortcut to starting up a future shot user interface. Use the command `runtest -f <name_of_archive>` to bring up the future shot interface without the main pcs window. When testing changes to the waveserver section of algorithms, this is all that is needed. Here, `<name_of_archive>` is the future shot setup name.

When the `runtest` script is started, it asks the following question:

```
Would you like to start up your own NEXT SHOT server processes?
(y or n)
```

Answering “no” will cause the script to start only the user interface. Then, the “control” menu is used to choose “future shot.”

9.3 Testing on the real time hardware

The `runtest` script starts the PCS in the mode of operation in which the real time code is executed on the actual real time processors.

This script is meant to execute the PCS temporarily when a developer is testing a build of the PCS software. The PCS processes are executed as children of the process in which the `runtest` script is executed. When the `runtest` script exits, all of the PCS processes exit also. This differs from the way the PCS processes are started by the `startpcs` script (Sec. 9.4) where the PCS processes are executed in the background and remain executing after the startup script completes. Also, the `runtest` script starts the user interface and when the user interface exits, the script exits also. In contrast, when the `startpcs` script is used, the `run_wave` script is used to start the user interface and connect to the already executing PCS processes.

The `runtest` script is located in the directory into which the installation-specific executable code is placed when it is built. For instance, following the example in Sec. 8.1, the script would be `/pcshome/pcs/SUN_DIR/runtest` on a Sun Solaris system. The script uses the file `pcs_config_info_test` to obtain default configuration information.

Normally the script is executed without arguments. However, there are options that can be used to customize the script execution. For instance, options are specified to control on which computer the PCS processes are executed. See the comments at the beginning of the `runtest` script for descriptions of the options. Also, the pcs_config_info_test can be used to configure the test environment.

The `runtest` script can be executed with the default directory set to any directory. The PCS executables used are always those in the same directory where the `runtest` script is located. Thus, any build of the PCS can be executed. For instance, any user can execute in his own directory area the development version of another user.

The `runtest` script creates a directory called `./runtest_dir` to contain files associated with its execution. Within this directory, there is a subdirectory for each PCS process. Each process is executed with its subdirectory as the default directory so that any core file for that process is written to its subdirectory so that the core file can easily be associated with the process creating it.

The waveform server process stores its `Nextshot.wa10` file in its subdirectory rather than using the central PCS archive area (see Sec. 2.18). However, any future shot files used by the `runtest` execution are still created or read from the central PCS archive area.

When the `runtest` script is started, it asks the following question:

```
Would you like to start up your own NEXT SHOT server processes?
(y or n)
```

Answering "yes" will cause the script to start up all the processes (user interface, waveserver, lockserver, msgserver, and host_cpu(s)). Answering "no" will allow you to bring up a future shot interface or a next shot interface to an existing set of processes.

9.4 Starting the PCS for normal operation

For normal operation, when the PCS is in use for routine tokamak operations, the PCS processes (waveserver, lockserver, msgserver, and host_cpu(s)) are executed in the background. These processes are allowed to execute continuously. The PCS operator only needs to start the user interface in order to communicate with the already executing PCS. In this way the operator doesn't need to be concerned with details of the PCS operation. In addition, multiple operators can start the user interface and communicate with the PCS simultaneously.

The script used to start the PCS for normal operations is `startpcs`. The script is intended to start the production version of the PCS executables (see Sec. 8.6). The version of the PCS that is started is actually specified in in the configuration script `pcs_config_info` which is read by `startpcs`. There are some optional arguments to the `startpcs` script, though, that can be used to modify this behavior. See the documentation at the beginning of the script for details. After the script is executed the PCS processes waveserver, lockserver, msgserver, and the host_cpu process for each real time CPU are executing in the background and ready to run a tokamak shot.

The script used by the operators to start the user interface is `run_wave`. The script takes no arguments. This script simply starts up the main user interface window which has available menu options that allow communication with the waveform server containing the setup for tokamak shots ("control/next shot") or the user can choose to edit a future shot setup ("control/future shot").

To stop the PCS processes executing in the background, the script `stoppcs` is used. This script takes no arguments.

To check which PCS processes are running, use the script `checkpcs`. This script takes no arguments.

These scripts are all normally made available in the standard path for all PCS users so that anyone can execute the scripts. For example, normally soft links to the copies of the scripts located in the directory with the production version (Sec. 8.6) of the PCS executables are placed in `/usr/local/bin`.

Note that the user interfaces started with `run_wave` execute independently of the processes started with `startpcs` and stopped with `stoppcs`. So, if it is necessary to stop the PCS processes and then restart them to clear up some sort of problem, the PCS operators can usually leave their user interface processes executing during this PCS restart. The user interfaces do not normally need to be restarted.

Chapter 10

Control Algorithm Testing

The control system infrastructure provides the facilities to extensively test control algorithms off-line. This section contains a description of these capabilities and how to use them.

Both the functionality of the algorithm and the correct implementation of the algorithm in the control system can be tested. Functionality of the algorithm refers to whether the feedback control technique implemented in the algorithm will actually do the proper control of the quantities of interest. Implementation refers to whether the code in the algorithm master file does the job correctly.

There are three test modes in the PCS - software test mode, simulation test mode, and hardware test mode. There is also the ability to interactively debug supercard program execution, special IDL routines designed to help analyze supercard memory dump files (s-files), and a routine to help identify the target, error, P, command and shape vector indices used by your algorithm when executing.

Software test mode simply executes all software linked into the PCS with input values set to 0. Functions which require communication with hardware are bypassed. As a result, software test mode will execute on any platform with a SuperCard, whether or not it is the actual control system platform that includes the input and output hardware.

Hardware test mode exercises the entire system, including input and output hardware. However, in this test mode there is no synchronization with the DIII-D shot cycle, so there is no meaningful data obtained from the digitizers. Data input is whatever signal is entering PCS digitizers during execution - typically bit noise. In this mode the software must be executed on the control system hardware because it executes all software including that which communicates with hardware.

Simulation test mode is similar to software test mode in that it can execute on any platform with a SuperCard but it provides a more comprehensive test because input data can be made representative of real tokamak operation. The input data is provided by a "simulation server" process and the control system response is provided to the server. There is a standard server that can simply provide the data from an old shot or a custom server that models some portion of the experiment can be created to test the closed loop feedback control cycle.

In software development, testing usually proceeds from testing individual functions (component tests) through testing of various integrated versions of the software up to full integration testing (testing the entire system at once). In the following, we describe the recommended sequence of testing for implementing a control algorithm within the PCS. As part of this description, we also discuss how to use the testing tools available.

Note that any testing of the PCS which requires execution of the real time code on the supercard must be done using the "next shot" option under "plasma control". During operations, this type of testing is restricted to a platform other than the actual control system so as not to conflict with DIII-D plasma control.

10.1 Setting up for Algorithm Testing

To obtain and compile your own copy of the source code, do the following:

```
cd ~
mkdir pcs
cd pcs
ln -s /home/pcs/SCCS ./SCCS
sccs get *
```

`/home/pcs` here indicates the directory where the control system code is stored. On some systems, a different path name may be necessary (e.g. `/pchome/pcs`). This sequence of steps creates a subdirectory `pcs` of your main directory, creates a link to the source code in `SCCS`, and gets all the source needed. Before compiling and linking, you will need to do a "chmod u+w" on each of the following files, then edit:

- Makefile: add the name of your algorithm master file to the list of dependencies defining ALGORITHM_FILES

- shapealgorithms.h: add the following:

```
            #define A_CODE 5
            #include "alg_master.h"
            #undef A_CODE
```

where alg_master.h is replaced by the name of your algorithm master file, and the algorithm number 5 is replaced by the next available algorithm number.

To compile and link your test version of the PCS with your new algorithm, type

```
make onecpu MODEDEF=-DTEST
```

This makes an executable version which executes on only one supercard. Often only one cpu is available when testing algorithms.

10.2 Component Testing

First component test individual functions. This consists primarily of the real-time algorithm function plus routines which execute in the waveserver (e.g. *_vectors, *_parameters, and *_vertices), plus any additional functions which these call. Methods for testing correct execution of individual functions will usually consist of embedded print statements in user developed code. Interactive debuggers are available however to assist with particularly hard-to-find bugs.

Interactive or so-called symbolic debuggers allow a code developer to watch his code being executed and thus identify more easily problems with execution. Code which executes on the SUN may be debugged using the dbx debugger, which is fairly easy to use. A less user-friendly debugger called sdbcs is available for debugging code which executes on the supercard. We have learned through experience that using both debuggers simultaneously does not work well.

Information on the dbx debugger is available through the man pages on the SUN workstations. (This debugger is generally not available on the HP workstations, although another debugger called xdb is.) We will not discuss the use of dbx here because user developed code which executes on the host is typically fairly simple and usually does not warrant the time investment required to set up to use dbx on user-developed algorithms in the PCS software.

To set up for sdbcs, you will need to edit the Makefile:

1. add -DDEBUG1 to the HDBG_OPTS options, e.g.
   HDBG_OPTS = -DDEBUG1
2. add -g to the SCDBG_OPTS options, e.g.
   SCDBG_OPTS = -g
3. change the Host Support Library option to -ldbg rather than -lcs, e.g.
   HSL_LIB = -ldbg

The host_cpu and real time programs need to be recompiled after these changes.

To execute using sdbcs, start up the waveserver from IDL. When the waveform window has made itself available for modification, open another window, kill the host_cpu1 process (occasionally it will have died by itself), and start up a new host_cpu1 process, e.g.

```
ps -aux | grep host
kill (PID number of host_cpu1)
host_cpu1
```

This will give an executing version of host_cpu1 which will talk to you and allow you to interactively debug the code executing on the supercard. Because you have "disconnected" this process from the rest of the PCS, it will first ask for the host and port number of the waveserver so that it may reestablish the connection. This information is available from the waveserver by pressing the "access information" button under "options".

After providing this information, the supercard debugger will take over:

```
 SuperCard  i860(TM) DEBUGGER, Release 1.2
 Copyright (C) 1989,1990 Intel Corporation
 Copyright (C) 1991 CSP, Inc

cs0>
```

The first input should be name of the entry point in the real time code at which to stop, e.g.

```
cs0> _snull:b
cs0> :r
```

The first command sets a breakpoint at the function snull. The command :r stands for "run" and initiates execution. At this point, the real-time code is ready to execute. Execution is initiated using the "lock", "final lock" mechanism in the waveserver after programming your waveforms. The PCS can be in any of the three test modes. The executing supercard code will stop at the specified breakpoint snull. Then do

```
:i
:i
```

to step into the routine snull. (Sometimes you will see cs0> prompt, sometimes not.) At this point the source code of the snull routine is shown, and interactive execution of the snull routine can begin. To debug your algorithm instead of snull, replace snull everywhere above with the name of your real time function. See the Supercard Debugger User Manual (MP-SC-019-03) for information on debugger commands.

Choosing options/quit from the waveserver will terminate both the waveserver and the host_cpu1 process.

10.3 Small Scale Integration Testing

An integrated test of waveserver functions is provided by generating the supercard memory dump file (s-file) and examining it using IDL. To do this, use any of test modes under "options", "global parameter data", "Standard operator data", "Operation mode". Enter a test shot number larger than 990000 in "diagnostic setup data" under "global parameter data". Use the "lock"/"final lock" mechanism to initiate the test shot. An s-file for the shot number specified will be created by the PCS. You may wish to replace your algorithm name in the *_functionnames array in your algorithm master file with "donothing" for this test. This ensures that the test shot will run to completion even if there is an error in your real-time code. At this point, only correct execution of the waveserver routines is of interest.

When execution is complete (message = "shot completed" from the host_cpu routine), exit the PCS program and start a new IDL session. (Alternatively, start another IDL session in a second window). Type

```
IDL> getrtdata,######,1
```

(Here, ###### is the test shot number.) This will get all pointers to data in the s-file. Type

```
IDL> listget
```

to list the set of available functions for examining various data objects in the s-file. For example,

```
IDL> getshapev,shape
```

will fetch all shape vectors calculated during the test shot and place them in the variable "shape". The shapes calculated by your algorithm can then be examined, provided you know the appropriate offset within the shape array of shapes which your algorithm has calculated. This information is available through the program get_vector_maps. Consider the following example, where the test shot number used was 993002.

```
IDL> getrtdata,993002,1
IDL> gettime,t
IDL> gettarget,target
IDL> $get_vector_maps -t | more
DNTZPP     0   cat: 1, alg 1: Double Null Divertor
DNTRPP     1   cat: 1, alg 1: Double Null Divertor
DNTPREBIP  2   cat: 1, alg 1: Double Null Divertor
DNTBIAS7P  3   cat: 1, alg 1: Double Null Divertor
DNTWVSP1P  4   cat: 1, alg 1: Double Null Divertor
DNTZXP     5   cat: 1, alg 1: Double Null Divertor
DNTGAPINP  6   cat: 1, alg 1: Double Null Divertor
DNTSHAPEP  7   cat: 1, alg 1: Double Null Divertor
DNTELP     8   cat: 1, alg 1: Double Null Divertor
DNT8ATO5AP 9   cat: 1, alg 1: Double Null Divertor
DNT4ATO5AP 10  cat: 1, alg 1: Double Null Divertor
DNT2ATO5AP 11  cat: 1, alg 1: Double Null Divertor
DNT8BTO5BP 12  cat: 1, alg 1: Double Null Divertor
DNT4BTO5BP 13  cat: 1, alg 1: Double Null Divertor
DNT2BTO5BP 14  cat: 1, alg 1: Double Null Divertor
DNTWHICHM  277 cat: 1, alg 1: Double Null Divertor
DNTWHICHR  278 cat: 1, alg 1: Double Null Divertor
```

```
ANTZPP     0   cat: 1, alg 2: Single Null Divertor
ANTRPP     1   cat: 1, alg 2: Single Null Divertor
ANTPREBIP  2   cat: 1, alg 2: Single Null Divertor
ANTBIAS7P  3   cat: 1, alg 2: Single Null Divertor
    .
    .
    .
IDL> plot,t,target(0,*)
```

This example shows how to determine the form of the ZPP target signal. The function getrtdata was used first to obtain all data pointers. Then gettime was used to obtain the time array which corresponds to all target, shape, and error vector data. The function gettarget obtained all target signals for the shot. The UNIX function get_vector_maps ($ escapes to the operating system) with the option -t displays the offsets in the target array of each target signal by name. These names are specified by each algorithm author in the *_targetnames array in the algorithm master file. For example, the index 0 corresponds to DNTZPP which comes from the dnull_targetnames array in dnull_master.h. It is the target signal giving the plasma z-position program. Finally, the zpp target signal is plotted versus time for inspection.

The capability of analyzing the s-file is also useful for later integration testing of the real-time algorithm because it allows you to examine shape and error vectors produced by real-time code. Where to find info on using get_vector_maps??

10.4 Large Scale Integration Testing

A set of software known as the simulation server (simserver) has been developed to aid in full integration testing of user developed control algorithms. The simulation server provides the capability to emulate in software the execution of the entire shot sequence. The same PCS software executes in simulation as will run in an actual shot. The only difference is in the origin of the data that the control algorithm processes and in the fact that resulting commands are not actually written out to the D/A converters. (See Figure 4-1.) This test capability is obtained by running using the PCS "simulation test" mode.

Figure 4-1. The simulation server provides an interactive simulation of DIII-D and plasma for testing of algorithm implementation.

The simserver program emulates DIII-D operation by providing simulated digitized "diagnostic data" to the PCS as a function of time. The data provided might be as simple as diagnostic data read from the shot database, or it might be calculated in "real-time" using a simulation of the actual physical process being controlled. The algorithm programmer is responsible for providing the software which generates the data to send to the PCS. However, all communications to and from the PCS are taken care of by the existing simserver program. In addition, an example set of software is available which reads shot data for transmission to the PCS. See the file sim.readme in SCCS for the name of the example file and instructions on how to compile and link the resulting simserver simulation.

The user specific portion of the simserver code consists of the three required functions sim_inits, sim_ends, and return_simulated_data, plus any functions these use to prepare data. These functions are linked with the simserver main routine as described in sim.readme. The return_simulated_data function actually does the work of generating data to pass to the PCS. The simserver main routine calls this function each time more data is requested by the PCS. The sim_inits function does any initialization required for the return_simulated_data function to work properly. The sim_ends function does any clean-up, including in some cases writing out data. The example code described in sim.readme provides data from an old shot file read in using ptdata. The sim_ends function in this example code does nothing.

A special test program is also provided to allow you to easily test a new simserver program without having to immediately run with the entire PCS. This program takes the place of the plasma control system and serves as a test driver of the simulation server. This test program can be modified to perform any desired input/output testing of the simulation server. This test setup also allows the dbx debugger to be used easily in debugging your simulation server software. (The difficulty with using dbx in debugging software embedded in the PCS is because standard output from the programs to be debugged has been redirected, so there is no "interactive" for the interactive debugger. This difficulty can be circumvented but is usually not worth the trouble.) See the file sim.readme in SCCS for a description of the example simserver test routine.

After you are confident that the simserver program is working correctly, run it with the plasma control system to test correct execution of your algorithm. First start up the plasma control program and select "next shot" under "plasma control". Change the mode to simulation test mode by pressing "options", then "global parameter data", then "standard operator data". Press "Operation Mode" and select "simulation test".

Start up the simserver program and supply it with the host and port number which are available from the waveserver by pressing "options", then "access information". Once you have provided this information correctly to the simserver, the program is now ready to interact with the PCS program and provide it with the simulated data it needs during a test shot. Proceed with normal use of the PCS program, supplying it the inputs needed to run your test shot. Force a lock and final lock to startup the real time processes after you have finished programming your waveforms by using the "lock/unlock" function under "options". This will run a simulated shot using inputs provided by the simulation server. Results of this test may be obtained by analyzing the s-file written by the PCS and any output files written by the sim_ends function in the simserver.

10.5 Evaluating Test Results

There are two questions which algorithm testing can be designed to answer: (1) Does the code correctly do the calculations you told it to? (2) Will the results of these calculations have the intended effect in controlling DIII-D? Most tests of software execution assume that the correct result of software execution is known and can simply be compared with the output from the actual software execution to complete the test. In those cases where the "correct" answer is available, e.g. shapes calculated by EFIT or from previous versions of the algorithm code, this is the case. These tests provide an affirmative answer to (1) and, if the previous implementation of the algorithm provided effective control, also to (2). The second question will usually be addressed independently of the process of PCS algorithm implementation however.

A method which has been used previously in developing software for the PCS involves building a parallel implementation of the control algorithm (in this case, in IDL) which should give the same results as the algorithm executing in C code within the PCS. This approach seeks to ensure correct calculation (question (1)) by using the fact that the likelihood of the same implementation error occurring in both the C and IDL versions is small. Example IDL implementations are available from the PCS administrator.

If the "correct" answer is not known, another option is to use the simserver code to simulate the system being controlled. In this case, the return_simulated_data function provides the results of one data acquisition time step in a software simulation of the physical system to be controlled. If the simulation is effectively controlled by the PCS, then the real time code is apparently working correctly. This method has been used with the simserver implementing a software simulation of tokamak plasma vertical motion. In this case, the closed-loop effectiveness of the algorithm is tested simultaneously with correctness of code execution. For this purpose, tools have been developed which link the PCS simserver software with simulation code which is automatically generated by the MATLAB software package.

Although each of these methods have been used previously, they are not necessarily the only methods for ensuring correct execution.

10.6 Testing hints

Note: this section is under construction. Presently, it is just a collection of notes.

When running in "standalone" mode with the `runsa` script, system resources such as shared memory and semaphores are used. At the end of the standalone session, these resources are released. Occasionally this doesn't happen properly if the `runsa` script is interrupted prematurely. In this case, following `runsa` sessions might not run because the needed resources are not available. If this is suspected, enter the command `ipcs` at the unix shell prompt. If there is a long list of resources printed with names attached other than "root", it may be necessary to clean up by hand. The appropriate commands are `ipcrm -m` and `ipcrm`

-s. See the man page for ipcrm for more details.

Chapter 11

Installing Your Algorithm in the Plasma Control System

When your algorithm has been fully documented and tested, it is ready to be installed into the PCS. This section discusses the requirements and procedure for this installation. The installed algorithm should consist of only one file, the algorithm master file, including all code and documentation. Data files necessary for the algorithm may be maintained separately. The file names and locations should be emailed to the PCS administrator for installation. The algorithm master file must contain the algorithm documentation/specification described in section 3, a test summary as described below, and the algorithm code as described in section ??. Each individual function in the algorithm code must incorporate standard header documentation as described below.

The remainder of this section describes the required test summary and individual function header documentation. The purpose of the test summary is to provide the minimum documentation necessary to convince the PCS administrator that the algorithm code has been sufficiently tested prior to installation. It should summarize an input/output integration test of the principal functions contained in the algorithm master file: (1) the waveform server functions, and (2) the real time functions. For each set of functions, the summary should consist of

1. The test procedure. For the waveform server functions, the input/output test should consist of testing correct transformation of input waveforms to target vectors using an analysis of the resulting s-file. For the real time functions, the input/output test should use the simserver to provide input to the PCS execution. The waveform server functions should be tested and validated before conducting the real time functions test.

2. A description of test scenarios and input data for each. The algorithm should be tested under a number of different anticipated operational conditions.

3. Expected result of each scenario test.

4. Actual result of each scenario test.
5. A brief discussion of how you know the algorithm is executing correctly. (See the discussion in section 10.5.)

Once an algorithm has been installed in the PCS, read access to the source code is unrestricted but write access is controlled by the PCS administrator.

Standardized header documentation is required for each function in the algorithm master file. A documentation template (Figure 4-2) is available from the PCS administrator. The purpose of standardizing this documentation is to facilitate ongoing maintenance of algorithm code, especially if this responsibility is passed from the algorithm author to someone else. An example of a filled in header template is shown in Figure 4-3.

```
/*
*******************************************************
SUBROUTINE: your_routine
PARTOF: which_process (etc. waveserver, lockserver, etc)

PURPOSE: Describe here in one or two sentences what this routine does.

INPUTS:
in_arg1 - one line description of what INPUT arg1 is.
in_arg2 - one line description of what INPUT arg2 is.
global_arg1 - one line description of global_arg1.

OUTPUTS:
out_arg1 - one line description of what OUTPUT arg1 is.
out_arg2 - one line description of what OUTPUT arg2 is.
global_arg2 - one line description of global_arg2.

RESTRICTIONS:
Describe here any restrictions and requirements on this program.
For example,
this program requires that subroutine_x be run first.

NOTES:
Describe here any special notes which might be useful in maintaining this
routine in the future.
********************************************************
*/
void your_routine(in_arg1,in_arg2,out_arg1,out_arg2)
{
}
```

Figure 4-2. Header template for documenting individual C functions in algorithm master files.

```
/*
*******************************************************
SUBROUTINE: zcontrol

PURPOSE: Calculate input command voltage to choppers connected
to the F-coils 2a, 2b, 7a, and 7b.  Commands to choppers are specified by
directing commands to individual F-coils.
This is the real-time C code implementation of vertical position control.
It is also the code used in the simulations of vertical control.

INPUTS:
Everything contained in rtheap = The real time data structure of
(structure rt_heap_misc in realtime_data.h).  Data used is:
rtheap->adtarget = address of target vectors
    (units defined by zcontrol_vectors)
rtheap->datavector = input data after offset removed (struct input_data *)
    (units = digitizer bits)
NCNTLRS = number of linear controller matrices
NCNTLR_INPUTS = number of inputs to controller
                (includes measurements/target)
NCNTLR_OUTPUTS = number of outputs of controller
NCNTLR_ROWS = number of rows in controller (= states + outputs)
NCNTLR_COLS = number of columns in controller (= states + inputs)

OUTPUTS:
rtheap->intcommand = pointer to vector containing outputs of commands to
     choppers (chopper commands go from 0->17 = F1A..F9A, F1B..F9B)
rtheap->shapevector[S_ZCONTROL_STATES-1] = pointer to location to keep
    calculated states of controller
rtheap->alldata = used to record "fast" data

RESTRICTIONS:
Controllers must be scaled to accept measurement in
acquired bits rather than
physical units or acquired voltage and to return commands in bits for D/A.

NOTES:
None.

*******************************************************
*/
void zcontrol(struct rt_heap_misc *rtheap)
{
```

```
(etc.)
}
```

Figure 4-3. Example of a filled in header template.

Chapter 12

The simulation server

The plasma control system can be run in a mode in which the data that are normally obtained in real time from tokamak diagnostics are instead obtained from a computational simulation of a tokamak. This provides the capability to test both the implementation of a control algorithm and whether the algorithm will actually control the plasma properly. In this testing mode of the PCS, the same code is executed that would normally be used in real time. The difference is the source of the input data obtained by the PCS. In this test mode the data are provided by the "simulation server." The simulation server can be used when the PCS is operated in stand alone mode or when the PCS is executed on the real time processors, as described in Sec. 9.

The simulation server is a process that executes on any desired computer. This process communicates with the PCS to provide simulated diagnostic data. The simulation server also receives from the PCS the calculated actuator command values. This two way communication allows the simulation server to execute a simulation of the response of the plasma to the actuator commands generated by the control system algorithms.

The tokamak simulation implemented in the simulation server process will probably be unique to the PCS implementation. Also, there may be more than one possible type of tokamak simulation that can be used. For this reason the simulation server software is structured so that a standard set of utility routines can be combined with a set of installation-specific functions to create a customized simulation server.

The simplest simulation server process simply reads the diagnostic data from the tokamak database for a discharge that has previously been run. These data are simply passed to the PCS for analysis and the command values returned from the PCS are ignored. This simple simulation server is extremely useful for testing of algorithms that can be done by simply providing reasonable data values.

This section describes how to execute the simulation server and how to create a customized server process. The first sections provide information useful for the user of a previously written simulation server application. Section 12.3 provides information on how to write the code for a custom simulation server.

12.1 Overview of simulation server operation

The simulation server runs a relatively simple cycle that is repeated until the shot completes.

1. After the simulation server is initialized, it waits to hear from the PCS. During the initialization for a shot, the PCS tells the simulation server the time at which the shot should end. This ending time is specified by the user on the operating setup data window.

2. The simulation server then waits until it receives a request for data from each of the PCS real time CPUs. Each of the real time CPUs uses its `host_cpu` process to provide the facilities for communication with the simulation server.

3. Once it has been established that all of the real time CPUs are ready and waiting for data, each CPU is provided with data in turn. The first CPU provided with data is the "master" CPU. Then, the simulation server cycles through the other CPUs starting at CPU 1 (skipping the master CPU).

4. When the simulation server provides a set of data to a real time CPU, it waits to receive a new request for data from that CPU. This new request is generated after the CPU finishes its control cycle using the set of data that was provided. Only after receiving the new data request, the simulation server moves on to the next CPU. This ensures that each time a simulated shot is executed, the control cycles and any necessary passing of data between the real time CPUs are executed in exactly the same order.

5. When each CPU has been given a set of data, the cycle starts again with the master CPU.

6. The last set of data provided to each CPU is the first set of data that is labeled with a time value greater than or equal to the "time to stop" that was provided by the PCS at the start of the shot sequence. After this last set of data is transmitted to all of the real time CPUs, the simulation server stops sending data. The assumption is made that the real time CPUs will also recognize this last set of data as marking the end of the shot sequence.

12.2 Using the simulation server with the PCS

To run the simulation server with the plasma control system, first start up the server process. A typical name for the executable is `simserver`. The executable is found in the PCS build subdirectory appropriate for the computer architecture where the simulation server process will be executing (see Secs. 8.1 and 2.18). Arguments to the simulation server are provided either on the command line or, if not provided there, the simulation server prompts for the necessary information. See Sec. 12.2.2 for a description of the command line arguments.

After the simulation server finishes initializing itself, it will list the number of its communication port. This number will be provided to the PCS so that the simulation process and the PCS can communicate.

Next, start up the PCS. See Sec. 9 for a description of the various testing modes of the PCS and how to start the PCS in the appropriate mode of operation. Once the PCS has started, bring up a "next shot" navigation/waveform editor window. Find the "operating setup data" window (usually in the Data Acquisition category) and choose the simulation test mode. Enter the name of the host computer where the simulation server is executing, the number of its communication port (the value that the simulation server printed after completing initialization), and a test shot number.

Once you have provided this information to the PCS, the simulation server is ready to interact with the PCS and provide it with the simulated data it needs during a test shot. Start a shot by choosing "manual cycle control" from the `File` menu and then "start test shot". This will run a complete shot sequence. The archived data (Sec. 2.7) and the S files (Sec. 13.4) can be examined for the results.

12.2.1 Information required by the simulation server

The simulation server typically requires some parameters.

- The simulation server can provide data to multiple real time CPUs configured into the PCS. Because the purpose of the simulation server is to replace the data acquisition hardware during testing, data wouldn't normally be passed to any CPUs that do not actually have data acquisition hardware connected. So, the simulation server usually must be told the number of real time CPUs from which it should expect requests for data. This is not necessarily the same as the number of CPUs configured into the PCS.

- The simulation server provides data first to a "master" CPU (Sec. 12.1). The number of the master CPU must be provided by the user.

- Each set of data passed to the PCS is labeled with the corresponding time during the simulated shot. The simulation server must be told the time interval between the simulated sets of data passed to the PCS. This time interval might be determined by the custom portions of the simulation server. The shorter the time interval and the longer the duration of the shot, the longer the simulated shot will take to run. However, in order to simulate fine time scale control it may be necessary to set a short interval between data sets. The data interval can be different for each of the real time CPUs to simulate the situation where the cycle time varies among the CPUs.

12.2.2 Simulation server command line arguments

Here is a summary of the typical set of command line arguments for the simulation server. Because each simulation server executable is installation-dependent, the set of command line arguments is likely to vary.

- `-n numrtcpus`: number of real time cpus to provide data for, default is 1.
- `-m mastercpu`: number (1,2:.) of the real time cpu serving as the master, the default is cpu 1.
- `-b begin_time`: begin simulation time in seconds.
- `-s shot`: the shot number from which the input data should be obtained. This value is appropriate for a simulation server that is defined to simply pass to the PCS the data that were acquired on a previous shot. The default is shot 0 which means to use dummy data.
- `-t which data`: When the simulation server passes data that were acquired on a previous shot, it may have a choice of more than one archive location from which to read the data. For instance, both the data acquired by the PCS and the data acquired by the standard data acquisition digitizers might be available. This argument specifies the archive location from which the data should be taken.
- `-d delta time`: the time interval between data samples provided to the PCS, in microseconds; e.g. `-d 1000` which provide data every millisecond.
- `-D level`: turn on debug messages, level=1 is best, 2 gives more.
- `-E cpu:delay,`: specify a computational delay for one or more cpus. The simserver will delay giving data to a cpu to emulate the computational time for that cpu.
- `-e`: ask question about cpu computational delay times.
- `-C cpu:delay,`: specify a command delay for one or more cpus. The simserver will set the time that commands from a cpu are valid using this value; the value must be greater than 0.
- `-c`: ask question about cpu command delay times.
- `-o channel`: channel whose data is output during shot. (1 to N) on given cpu to output, cpu is 1 if not given.
- `-i interval`: interval (in usec) to output data (0 to not output).

12.3 Creating a custom simulation server

1. Obtain all of the necessary files (Sec. 12.3.1) from the PCS source code control system.
2. Make any necessary changes for the customized simulation server in the file `simspecific.c`. See the example `simspecific.c` file for more details and Sec. 12.3.3 for a description of the required functions.
3. The simulation server can be built by entering `make simserver` at the unix command prompt. It may be necessary to alter the `make` procedure files for a custom executable as described in Sec. 8.2.

12.3.1 Simulation server source files

The source files for the simulation server can be found primarily in the installation-specific source directory. The simulation server files are the following.

- `sim.readme`: a description of the simulation server containing information similar to what is in this document. This file may contain more up to date information.
- `simpcs.h`: include file for the simserver set of programs (an infrastructure file).
- `simserver.c`: main source for the simserver program. This file contains the generic functions that provide the framework within which a customized simulation server can be written.
- `simspecific.c`: user customizable code for the simserver program.
- `simserver`: the simserver executable (a product of the PCS build procedure (Sec. 8).
- `simhosttest.c`: source for the routine to test the simserver program (Sec. 12.3.2).
- `simhosttest_routines.c`: user customizable code for `simhosttest` program.
- `simhosttest`: routine used to test the simserver program (a product of the PCS build procedure (Sec. 8)..

12.3.2 Testing a simulation server by running in test mode

A special program has been provided to allow a new simulation server to be tested without having to operate with the PCS. This program is called `simhosttest`, and it serves to take the place of the plasma control system by communicating with the simulation server to provide it with some dummy data and print the results that are sent back to it.

To use this `simhosttest` program, first run the simulation server for one real time CPU and answer its questions. Next run the `simhosttest` program and supply the requested information about the simulation server process. After the information is supplied to the `simhosttest` program, this program should then begin interacting with the simulation server and it should print out some results.

`simhosttest` takes the place a single CPU. To test a multiple CPU configuration, more than one instance of the `simhosttest` program must be run. The only way to change the CPU number is to provide all the input information on the command line, as follows.

```
simhosttest node_name port_number channel_number_to_list cpu_number
```

12.3.3 Functions needed for a custom simulation server

In `simspecific.c` the following routines must be provided.

- `sim_parse_command_line`
- `sim_inits`
- `sim_ready_messages`
- `sim_return_simulated_data`
- `sim_increment_sim_time`
- `sim_ends`

These functions are described in this section.

12.3.3.1 sim_parse_command_line

Calling format:

```
sim_parse_command_line(argc,argv,optiontags)
```

This function parses the command line and is called first by the simserver main routine. The `optiontags` is input with a string of default options that are used in the main program. This string can be appended to with other tags that are installation dependent. The `switch_default_options` function can be called to handle the default options.

Example:

```
int SHOTNUMBER=-1;
int WHICHPOINTNAMES=-1;
int DELTATIME=-1;
extern int SIMDEBUG;
```

```
int sim_parse_command_line(int argc,char **argv,char *optiontags)
{
extern char *optarg;
extern int optind;
int c;
char user_optiontags[100];
char combined_optiontags[100];

    strcpy(user_optiontags,"s:t:d:D:");

    sprintf(combined_optiontags,"%s%s",optiontags,user_optiontags);
    while((c = getopt(argc,argv,combined_optiontags))!= -1)
    {
            switch_default_options(c,optarg);

            switch(c) {
                    case 's':
                            sscanf(optarg,"%d",&SHOTNUMBER);
                    break;

                    case 't':
                            sscanf(optarg,"%d",&WHICHPOINTNAMES);
                    break;

                    case 'd':
                            sscanf(optarg,"%d",&DELTATIME);
                    break;

                    case 'D':
                            sscanf(optarg,"%d",&SIMDEBUG);
                    break;
            }
    }
    return 0;
}
```

12.3.3.2 sim_inits

Calling format:

```
int sim_inits(int *cpu_cycle_times,int sim_cpu_count,int master_cpu_num)
```

This function is called after the command line is parsed. It needs to do any program initializations. The `cpu_cycle_times` is an integer array dimensioned to `MAXCPUS` and should be filled with `sim_cpu_count` cycle times in microseconds, `sim_cpu_count` is the count of cpus, and `master_cpu_num` is the number designated as the master cpu. Note that the cycle times for each cpu can be different and that the master cpu must be the one that has the fastest cycle time.

This function also needs to set two trigger times which are needed in the main program. These must be declared globally as variables of type `int`. These are values used to calculate baseline data at DIII-D, but could be used for other purposes. They are used in the function `default_increment_sim_time` found in `simserver.c` (see Sec. 12.3.3.5). To set these to values of -2 and -1 seconds, add the following to your `sim_inits` function:

```
    time_for_68 = (-2000000 + CLOCK_CLEAR_TIME); /* -2 seconds */
    time_for_70 = (-1000000 + CLOCK_CLEAR_TIME); /* -1 seconds */
```

This function might also ask if data for a shot should be used. Data could then be cached so that it can be provided later as quickly as possible.

12.3.3.3 sim_ready_messages

Calling format:

```
int sim_ready_messages(cpu_cycle_times,sim_cpu_count,master_cpu_num)
int *cpu_cycle_times;
int sim_cpu_count,master_cpu_num;
```

`sim_ready_messages` is a function that is called by the simulation server main routine when the program is ready to interact with the PCS. It can print out items like the cycle times, number of cpus being used, the master cpu, the shot whose data was read, what data, etc.

12.3.3.4 sim_return_simulated_data

Calling format:

```
int sim_return_simulated_data(sim_time,host_data,dig_return)
int sim_time;
struct sim_exchange_info_in **host_data;
struct sim_exchange_info_out *dig_return;
```

This function returns a set of digitizer values, one per channel, for the given time step. `sim_time` is the simulation time in microseconds. `host_data` is the data sent to the simserver from the PCS and can be used to simulate a "real-time" simserver. It contains all the important vectors (Target, Errors, Shape, etc..) and the vector lengths. `dig_return` is the output digitizer data that is sent to the `host_cpu` process. See the infrastructure file `simserver.h` for the definitions of these structures.

Note that the data array in the `sim_exchange_info_out` structure is of type `float` in order to provide for data that is either integer or real.

This routine must check for the beginning of the simulation by checking the time value against the initial start time of `PRIMARY_START_TIME_FS` set in the main simserver routine if any variables need resetting.

Here is a simple example that provides a constant value at every time.

```
int sim_return_simulated_data(sim_time,host_data,dig_return)
int sim_time;
struct sim_exchange_info_in **host_data;
struct sim_exchange_info_out *dig_return;
{
    double time_seconds;
    int chnl;
/*
-------------------------------------------------------------------------------
If we are at the start of the simulation then reset global values.
-------------------------------------------------------------------------------
*/
    if(sim_time == PRIMARY_START_TIME_FS)
    {
    }

    time_seconds = (double)((sim_time - CLOCK_CLEAR_TIME) * 1.0e-6);

    /*
    for each CHANNEL simply give a constant value
    */
    for(chnl=0;chnl<NUMCHANNELS;chnl++)
            (dig_return->data)[chnl] = 1000.0;
/*
-------------------------------------------------------------------------------
Else, get data for each channel at time sim_time from an archive.
Or, if this is a "real-time" simulation, then use the values that
the pcs gives in the sim_exchange_info_in structure (variable host_data)
to calculate the next set of raw data.
```

```
-------------------------------------------------------------------------------
*/
/*
-------------------------------------------------------------------------------
print out some diagnostics
-------------------------------------------------------------------------------
*/
    {
    int channel;

      channel = 1;

      printf(
      "%d,%f: %s = %f, trg = %x step = %d\n",
        sim_time,
        time_seconds,
        rawdata_defaults[channel-1].pointname,
        (dig_return->data)[channel-1],
        dig_return->simtrigger,
    }
    return 0;
}
```

12.3.3.5 sim_increment_sim_time

Calling format:

```
int sim_increment_sim_time(int *sim_time)
```

This function is called at the end of each loop in the simserver. It needs to increment the master simulation time. `sim_time` is given in microseconds. The increment would be one of those values that the user input earlier in the `sim_inits` function or it could be given on the command line. So this function could simply contain the following line:

```
    *sim_time = *sim_time + increment;
```

where `increment` would be a global variable.

Because this increment could be very short (100 microseconds, for example), and a shot might begin at -3 seconds, stepping through those 30000 cycles can take a good deal of time (DIII-D starts at -9 seconds). Therefore, a function can be found in `simserver.c` called `default_increment_sim_time` which will jump around before a shot and use increments that are much larger than the user given value. This function can be called directly or it can be used as an example.

12.3.3.6 sim_ends

Calling format:

```
int sim_ends()
```

`sim_ends` is a function that is called at the end of a simulated shot. This function can do any post-shot cleanup like closing of files, etc.

12.3.4 Simulation server time increments

Each time the simulation server provides a set of data to the PCS, it increments the simulated shot time, usually by the interval specified by the `-d` command line argument (Sec. 12.2.2). Depending on the standard timing for the PCS installation, there might be some long portions of the shot cycle where nothing happens, such as prior to $t = 0$ and before digitizer baseline values are determined. The simulation server can be configured to skip quickly through these uninteresting time intervals in order to minimize the time required to execute a shot simulation. The way the simulation server determines how to set the time interval is configured into the code. Here is a brief description.

```
default simserver timing found in default_increment_sim_time in
simserver.c and called from sim_increment_sim_time in simspecific.c:

D3D:
PRIMARY_START_TIME to -8 by 10 msec intervals (recently changed from -8.9 by 1)
-5 to -4.98 by 2 msec intervals (10 iterations)
hex68 trigger (-2) to hex68-0.02 (-1.98) by 2 msec intervals (10 iterations)
hex70 trigger (-1) to hex70-0.02 (-0.98) by 2 msec intervals (10 iterations)

if user_begin_time > CLOCK_CLEAR_TIME then
   use 100 msec intervals until 0 (goes to 0.02 from -0.98)
   skip to user_begin_time (default is 0.1)
   use user increment
else (user_begin_time <= CLOCK_CLEAR_TIME)
   use 100 msec intervals to user_begin_time
   use user increment

NSTX:
PRIMARY_START_TIME is -3.0
hex68 is -2
hex70 is -1
```

```
We should probably change this to use an installation array of times
and increments either defined in installdefs.h or have it read in a
user text file.
```

Chapter 13

Understanding the organization of the real time computer memory

13.1 The real time heap

The data memory of the real time computer is treated by the PCS infrastructure as one large "heap" of memory. That is, the memory organization is not predetermined by the source code but rather the allocation of memory is performed as necessary for each shot. However, the memory is not treated as it would be in a conventional C language program in which the `malloc` function would be used at run time by the application code to allocate new memory. Instead, all memory allocation is performed during setup for a shot. The infrastructure code arranges all of the standard data structures in the address space of the real time computer, determining the sizes of the various standard structures as needed for the next shot. Thus, use of memory is optimized by assigning only the required memory for each data structure. However, the list of data structures for which memory can be allocated is predetermined by the design of the control system. Application code should not create new data structures at run time.

This method of memory utilization ensures that there cannot be run time faults caused by lack of memory. The control system determines before a shot that there will be enough memory available for the algorithms assigned by the operator to the next shot. The necessary flexibility to add data structures for use by a control algorithm is provided by the parameter data facility (Sec. 2.11).

13.2 Memory regions other than the real time heap

Normally, all real time structures are allocated in one area of memory called the "real time heap" (see Sec. 13.1). This memory is preloaded with all the data and stuctures necessary in real time. But this memory is not normally shareable between processors in a multi-processor computer or between computers.

It may be desirable to share some of the data structures between processors in one computer using system shareable memory or between computers using a hardware device like reflective memory which automatically copies data from one computer to another. By sharing a data structure there is no need to copy the data from one real time processor to another.

For example, the datavector and/or the physicsvector could be shared between real time processors. If put into a memory region which is shareable, then the pointers to these vectors could be made to point to the same memory location (in other words, the vector would be "overlayed"). Then only one processor would need to do the work of filling this vector. The other real time processors would need to check a flag that indicates the data have been written (usually, this flag is set to the value of the current time and this flag is put in the install_buffer structure).

If reflective memory is used, then this data would be passed to other computers automatically. The current time value would need to be read by the other computers until it changes, signaling a new set of data are available.

In addition, individual parameter data blocks can be put into a memory region other than the real time heap. These parameter data blocks can contain data that are shared between real time processors, or these blocks could contain data which one processor writes and another reads. Messages can also be written to a memory region by using the real time message facility (Sec. 2.9.4).

13.2.1 Creating a memory region

To create a new memory region, the waveform server needs to be told that a new memory region is desired. This is usually done in the `alg_parameters` function or in the `alg_vectors` function of an algorithm. A region number must be specified. So it is best to set macros in the installation file `config.h` that would specify the region numbers (in case more than one is desired).

To specify that a memory region is to be created on a particular real time processor, call the following function:

To use an installation specific memory region, you must create at least two functions and, optionally, a third:

1. install_create_memory in realinstall.h which actually creates the memory region. The address for the real time needs to be set. This might open the reflective memory device, for example.

2. install_delete_memory in realinstall.h which deletes the memory region (or closes the reflective memory).

3. host_install_create_memory_regions in hostinstall.h which maps the memory region. This function gets the address of the memory region from the real time process (see version of this function in hostmain.c). This is needed if the real time and host real time are separate programs (like in runsa).

The type of MEMORY_SHAREABLE is already built into the infrastructure. No need to write any code for that case. MEMORY_GENERIC is also built into the infrastructure and might be used to store data which is too large for the real time heap. Care must be taken that memory is actually available on the real time processor since the space allocated for these memory regions does not count against the maximum size limits imposed on the real time heap.

13.3 The real time heap structure

There is a single, standard data structure in the real time computer's memory that provides all of the pointers to the other data structures. This key data structure is normally referred to in the code as `rtheap` (for "real time heap"). A pointer to this structure is a standard argument for most functions in the real time code. The real time code can then locate other pointers within the `rtheap` structure to locate the required data.

`rtheap` is a structure of type `rt_heap_misc`. This structure is defined in the file `realtime_data.h`. The reader should refer to this file for details of the organization of the structure.

The different parts of the `rtheap` memory are put in two sections: `cacheable` and `noncacheable`. See the discussion of the communications buffer 2.9.2 for why there needs to be two sections. If the default section does not work for some reason, then it can be changed by calling the function `set_cache_preference` (Sec. 14.6.4) usually in the system algorithm's parameters function.

13.4 The real time computer memory dump file

At the completion of a discharge, the content of each real time computer's memory is written to a file, referred to as the "s file." This section describes this file and routines that are used to access the content of the file.

The s file contains both the data provided to the real time computer as part of the the pre-shot setup and any data computed during real time that remains at the end of the discharge. There is a separate file for each of the real time computers. The s file is primarily useful for diagnosing the behavior of the control system. It is not used for long term data archival. Only the s files for the most recent few shots are retained.

In addition to the memory dump, the s file contains some extra information that is determined after the shot. In particular, because the values of the target vector are computed in real time and are not saved, the target vector is recomputed and the values that were present during each of the control cycles for which the control system response was saved are stored in the s file.

The key data structure in the real time computer's memory is referred to as `rtheap` (for "real time heap") (Sec. 13.3). This structure contains pointers to all other data structures in the real time computer's memory. Routines that access the s file first read the `rtheap` structure and then use it to locate other data in the file.

The routines that access the s file are written in the IDL language. Normally, then, when examining the s file an interactive IDL session is used, as shown below (here `%` refers to the prompt from the UNIX shell).

First, set your default directory to a directory where PCS executables are located and start IDL. For example:

```
% cd  my-pcs-development-area/SUN_DIR
% idl
```

or

```
% cd  my-pcs-development-area/INTEL_DIR
% idl
```

The processor architecture corresponding to the executables in the directory chosen doesn't matter. The IDL routines are architecture-independent. Also, the IDL routines for accessing the S file can be executed on any machine architecture and will properly read an S file independent of the processor architecture on which the file was actually created. For instance, if the S file was written on a little-endian machine, it can be read using IDL on a big-endian machine.

Then, to access an s file some internal data structures must be initialized with a function call that finds the file and copies the `rtheap` structure.

```
IDL> getrtdata,shot_number,cpu_number
```

Here, `shot_number` is the number of the shot for which the file was created, and `cpu_number` is the number of the physical real time computer for which the file was created. `getrtdata` must be called again in order to change the s file being accessed. Each of the s file access procedures accesses the file specified in the most recent call to `getrtdata`.

The call to `getrtdata` also loads the pre-compiled set of s file access routines from the file `s_file_access`. A summary list of all of the s file access routines can be printed.

```
IDL> listget
```

After the call to `getrtdata`, any of the routines listed by the `listget` procedure can be called in order to return some data from the s file. For instance, the following returns the `rtheap` structure and displays its content.

```
IDL> getrtheap,rtheap
IDL> help,/structure,rtheap
```

Or, to return the entire content of the list of buffers that were used to receive direct memory access data from the data acquisition circuits:

```
IDL> getdmabuffer,dma
IDL> help,dma
IDL> help,/structure,dma(0)
```

13.5 Routines for probing the S file

The following sections describe the s file access procedures.

Some of these procedures access data structures that are useful and recognizable to the application programmer. However, some of these procedures access obscure data structures that are useful primarily for understanding the behavior of the infrastructure code. For each of the routines below, a note is added indicating whether the routine is likely to be useful for an application programmer.

This section is far from complete. The `listget` procedure can be used to view the most current list of procedures and a summary of the procedure arguments, but it doesn't provide the detailed information that would be here if this section was complete.

13.5.1 getalldata

This procedure is useful for an application programmer. Usage:

```
IDL> getalldata,s
```

The `alldata` data structure is used to hold "fast sampled data." This is a set of data that is saved on every control cycle. On each cycle, the absolute time of that cycle is saved, plus there is space for storage of 7 `float` values. The real time code that is allowed to write into the `float` values is specified by the operator as a "system-wide parameter." This data structure is used primarily for diagnosing an application code function in cases where the data from every control cycle in a sequence is required.

The procedure returns in `s` an array of structures, each of which holds the sample time and the 7 `float` values.

13.5.2 getbaseli

This procedure is not likely to be useful for an application programmer. Usage:

```
IDL> getbaseli,s
```

Returns in `s` the content of the `baseline_input` data structure, an array of `float` values. This array is used by a data acquisition slave processor to receive baseline data that was acquired by the data acquisition master processor.

13.5.3 getbaseline

This procedure is useful for an application programmer. Usage:

```
IDL> getbaseline,s
```

Returns in `s` the content of the `baseline` data structure, an array of `float` values. This array holds the baseline data values that are subtracted from the raw input data on each control cycle.

13.5.4 getchca

This procedure is not likely to be useful for an application programmer. Usage:

```
IDL> getchca,s,catnum
```

Returns in `s` the content of the "change list cache" for the category specified by the category index `catnum`. Note that `catnum` is the index on the physical real time computer of the specified category, not the category code number (see Sec. 2.9).

The change list cache is a data structure used by the infrastructure code to hold the next few values to be processed in the "change list" for the active shot phase of the specified category. Note that the values returned are what was left in the buffer at the end of the last control cycle.

13.5.5 getcombuf

This procedure is not likely to be useful for an application programmer. Usage:

```
IDL> getcombuf,s
```

Returns in `s` the content of the first-level input communication buffer. This is the buffer into which another real time computer would write data in order to perform some computer-to-computer communication. Note that the values returned are what was left in the buffer at the end of the last control cycle.

The buffer accessed by this procedure is not the buffer that is referenced by application code. The content of the first-level input communication buffer is copied to a second-level communication buffer during the data acquisition procedure for each control cycle. It is this second-level communication buffer that is referenced by application code. The first-level input communication buffer must never be reference by application code since this could bring the buffer into cache.

13.5.6 getcombufl

This procedure might be useful to the application programmer. Usage:

```
IDL> getcombufl,s
```

This procedure returns in `s` the content of the second-level communication buffer, the buffer that is referenced by the application code (see Sec. 13.5.5 for more detail about the communication buffer). Note that the values returned are what was left in the buffer at the end of the last control cycle.

Chapter 14

Utility routines used in the waveform server

This section contains the details of how to use the utility routines provided by the PCS infrastructure code in writing code to implement a control algorithm.

14.1 Macros used in the alg_vectors function

Here are the standard macros that can be used to define a `case` for the `switch` statement in an `alg_vectors` function. These macros are defined in the file `serverdefs.h`. The macro definition includes the proper syntax for the `switch` statement and the proper call to a function to do the work.

14.1.1 case_wvmp

Calling format:

```
\ntt{case\_wvmp(THE\_DE,THE\_T,CPU,WAVEFORM,OFFSET,FACTOR)}
```

Macro arguments:

- `THE_DE`: data entry index
- `THE_T`: target vector index
- `TCPU`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.)
- `WAVEFORM`: string giving the waveform name. Must be used by the algorithm used in the shot phase specified by the `phase` argument to the `alg_vectors` routine.
- `OFFSET`: the value to add to the waveform vertices

- `FACTOR`: the value to scale the waveform vertices

This macro causes the data for the target vector element specified by `THE_T` on the virtual cpu indicated by `CPU` to be generated by multiplying the waveform vertices by `FACTOR` and then adding `OFFSET`. This target vector element is identified as data entry `THE_DE`.

The target vector element is assumed to be part of the "continuous" target vector segment.

If `WAVEFORM` is a null pointer, a waveform is not used and `FACTOR` is ignored. Instead, a single vertex at $t = 0$ (relative to the beginning of the shot phase) is generated with value equal to `OFFSET`. This can be used to generate a constant (in time) continuous target vector element.

14.1.2 case_fst_wvmp

Calling format:

```
\ntt{case\_fst\_wvmp(THE\_DE,THE\_T,CPU,WAVEFORM,OFFSET,FACTOR)}
```

Macro arguments:

- `THE_DE`: data entry index
- `THE_T`: target vector index
- `CPU`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.)
- `WAVEFORM`: string giving the waveform name. Must be used by the algorithm used in `phase`.
- `OFFSET`: the value to add to the waveform vertices.
- `FACTOR`: the value to scale the waveform vertices.

This macro causes the data for the specified element in the floating step segment of the target vector to be generated by multiplying the waveform by `FACTOR` and then adding `OFFSET`.

If `WAVEFORM` is a null pointer, a waveform is not used and `FACTOR` is ignored. Instead, a single value at $t = 0$ (relative to the beginning of the shot phase) is generated with value equal to `OFFSET`. This can be used to generate a constant (in time) floating step target vector element.

14.1.3 case_ist_wvmp

Calling format:

```
\ntt{case\_ist\_wvmp(THE\_DE,THE\_T,CPU,WAVEFORM,OFFSET,FACTOR)}
```

Macro arguments:

- `THE_DE`: data entry index
- `THE_T`: target vector index
- `CPU`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.)
- `WAVEFORM`: string giving the waveform name. Must be used by the algorithm used in `phase`.
- `OFFSET`: the value to add to the waveform vertices.
- `FACTOR`: the value to scale the waveform vertices.

This macro causes the data for the specified element in the integer step segment of the target vector to be generated by multiplying the waveform by `FACTOR` and then adding `OFFSET`.

If `WAVEFORM` is a null pointer, a waveform is not used and `FACTOR` is ignored. Instead, a single value at $t = 0$ (relative to the beginning of the shot phase) is generated with integer value equal to `OFFSET`. This can be used to generate a constant (in time) integer step target vector element.

14.1.4 case_category_offset

Calling format:

```
\ntt{case\_category\_offset(THE\_DE,THE\_T,CPU,CATIDENT,CATCPU)}
```

Macro arguments:

- `THE_DE`: data entry index.
- `THE_T`: target vector index.
- `CPU`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.) on which the target vector element to be filled exists.
- `CATIDENT`: a string giving identifier of the category for which the offset should be computed.

- `CATCPU`: the virtual cpu of the category for which the offset is being computed.

This macro generates an integer step target vector value. The value is the index (0 based) (in an array of dimension `cat_count`) of the specified category on the category's specified virtual cpu.

14.1.5 case_ptrt_paramblock_wvmp

Calling format:

```
case_ptrt_paramblock_wvmp(THE_DE,THE_T,CPU,WAVEFORM,
                          OFFSET,FACTOR,BLOCK)
```

Macro arguments:

- `THE_DE`: data entry index.
- `THE_T`: pointer target vector index.
- `CPU`: virtual cpu number (e.g. `CPUA`, `CPUB`, `CPUC` etc.).
- `WAVEFORM`: string giving the waveform name. Must be used by the algorithm used in `phase`.
- `OFFSET`: a pointer to a data object descriptor (Sec. 2.11.4.5). The size in bytes of the data object is used as the value to add to the waveform vertices. If the pointer is `NULL`, no descriptor is expected and the size value is set to 0.
- `FACTOR`: a pointer to a data object descriptor (Sec. 2.11.4.5). The size in bytes of the data object is used as the value to scale the waveform vertices. If the pointer is `NULL`, no descriptor is expected and the size value is set to 0.
- `BLOCK`: the block order index assigned to the required parameter data block.

This macro causes the data for a specified element in the pointer target vector to be a pointer to a location at a specified offset into a particular parameter data block. The offset is generated by mulitiplying the waveform by the byte size given by `FACTOR`, then adding the number of bytes given by `OFFSET`. The waveform vertices are converted to integers by truncation before performing this calculation.

Here, `OFFSET` and `FACTOR` are pointers to data object descriptors (or `NULL`). The `convert_size` function (Sec. A) is used along with the processor type of the real time processor indicated by `CPU` to determine the size in bytes of the data object described by each of these descriptors. If the pointer is `NULL`, the corresponding value (offset or factor) is set to 0 (i.e. the pointer is to the beginning of the parameter data block).

If `WAVEFORM` is a null pointer, a waveform is not used and `FACTOR` is ignored. Instead, a single value assigned to $t = 0$ is generated with integer value equal to the size determined from `OFFSET`. This can be used to generate a constant (in time) pointer into the parameter data block. If `WAVEFORM` is a null pointer, the pointer to the parameter data block does not depend on any specific piece of raw data. Therefore the data entry number should be placed in the `categories_algorithms` structure (Sec. 4.6.1) in the list of processed data that doesn't depend on any specific piece of raw data.

The change list entry is generated as type `PTRT_PARAMBLOCK_CHANGE` to indicate that the integer data is an offset into a specified block of parameter data to which the address of the parameter data block should be added to get the real time value of the pointer target vector element.

The pointer to the parameter data block can be accessed as in the following example where the parameter data block to be accessed is an array of floats and `PTR_ELEMENT_NUMBER` is the macro for the pointer target vector element number.

```
float *mydata;
float **pointertargets;

pointertargets = (float **)rtheap->pointer_target;
mydata = (float *)pointertargets[PTR_ELEMENT_NUMBER - 1];
```

For more background information on the use of the `case_ptrt_paramblock_wvmp` macro see Sec. 2.11.7.2.

14.1.6 case_wvdp

Calling format:

```
case\_wvdp(THE\_DE,THE\_T,CPU,WAVEFORM,OFFSET,FACTOR)
```

Macro arguments:

- `THE_DE`: data entry index
- `THE_T`: target vector index
- `TCPU`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.)
- `WAVEFORM`: string giving the waveform name. Must be used by the algorithm used in the shot phase specified by the `phase` argument to the `alg_vectors` routine.
- `OFFSET`: the value to add to the waveform vertices
- `FACTOR`: the value used to divide the waveform vertices into

This macro causes the data for the target vector element specified by `THE_T` on the virtual cpu indicated by `CPU` to be generated by dividing the waveform into `FACTOR` and then adding `OFFSET`. This target vector element is identified as data entry `THE_DE`.

The target vector element is assumed to be part of the "continuous" target vector segment.

14.1.7 case_fst_wvdp

Calling format:

```
case\_fst\_wvdp(THE\_DE,THE\_T,CPU,WAVEFORM,OFFSET,FACTOR)
```

Macro arguments:

- `THE_DE`: data entry index.
- `THE_T`: target vector index.
- `CPU`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.).
- `WAVEFORM`: string giving the waveform name. Must be used by the algorithm used in `phase`.
- `OFFSET`: the value to add to the waveform vertices.
- `FACTOR`: the value used to divide the waveform vertices into.

This macro causes the data for the element in the floating step segment of the target vector to be generated by dividing the waveform into `FACTOR` and then adding `OFFSET`.

14.1.8 case_wv2mp

Calling format:

```
\ntt{case\_wv2mp(THE\_DE,THE\_T,CPU,WV1,WV2,OFFSET,FACTOR)}
```

Macro arguments:

- `THE_DE`: data entry index.
- `THE_T`: target vector index.
- `CPU`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.).
- `WV1`: string giving the name of waveform #1. Must be used by the algorithm used in `phase`.

- `WV2`: string giving the name of waveform #2. Must be used by the algorithm used in `phase`.
- `OFFSET`: the value to add to the waveform vertices.
- `FACTOR`: the value to scale the waveform vertices.

This macro causes the data for the element of the continuous segment of the target vector specified by `THE_T` to be generated by multiplying together the two waveforms and then multiplying the result by `FACTOR` and then adding `OFFSET`. The waveforms are specified by the waveform name. Both waveforms must be in the algorithm used by `phase`.

When the actual waveform argument is specified it must include the quotes normally provided for a string constant.

The waveforms can be either continuous waveforms or step waveforms, or a combination.

Note that the multiply, scale and offset operations are performed on the vertices of the waveforms. The result is a set of vertices that is used to define the time evolution of the target vector element by connecting these new vertices by straight lines. The target values obtained by following these lines are not what would be obtained by multiplying together the values on the line segments defining the waveforms. Multiplying two line segments would give a curve whereas the target values are generated with line segments.

14.1.9 case_fst_wv2mp

Calling format:

```
\ntt{case\_fst\_wv2mp(THE\_DE,THE\_T,CPU,WV1,WV2,OFFSET,FACTOR)}
```

Macro arguments:

- `THE_DE`: data entry index.
- `THE_T`: target vector index.
- `CPU`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.).
- `WV1`: string giving the name of waveform #1. Must be used by the algorithm used in `phase`.
- `WV2`: string giving the name of waveform #2. Must be used by the algorithm used in `phase`.
- `OFFSET`: the value to add to the waveform vertices.
- `FACTOR`: the value to scale the waveform vertices.

This macro causes the data for the element of the floating step segment of the target vector specified by `THE_T` to be generated by multiplying together two waveforms and then multiplying the result by `FACTOR` and then adding `OFFSET`. The waveforms are specified by the waveform name. Both waveforms must be in the algorithm used by `phase` and both waveforms must be of the "step" type.

When the actual waveform argument is specified it must include the quotes normally provided for a string constant.

14.1.10 case_make_pidtau

Calling format:

```
\ntt{case\_make\_pidtau(THE\_DE,THE\_T,CPU,WAVEFORM1,OFFSET1,FACTOR1,
              WAVEFORM2,OFFSET2,FACTOR2,WAVEFORM3,OFFSET3,FACTOR3)}
```

Macro arguments:

- `THE_DE`: data entry index.
- `THE_T`: target vector index.
- `CPU`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.)
- `WAVEFORM1- 3`: strings giving the waveform names. The waveforms must be used by the algorithm used in `phase`. `WAVEFORM1` is the time constant for the initial low pass filter applied to the error vector value, `WAVEFORM2` is the derivative time constant waveform, `WAVEFORM3` is the integral time constant waveform
- `OFFSET1 - 3`: the value to add to the waveform vertices.
- `FACTOR1 - 3`: the value to scale the waveform vertices.

This macro is used to provide the data that will cause the necessary lookup tables for the PID calculation algorithm to be generated. Pointers to the lookup tables will be placed in the correct locations in the pidtau vector.

The waveforms provide the three necessary time constants. After the waveforms are scaled and offset the waveform units must be seconds.

This macro causes the data for the specified element in the pidtau vector to be generated by multiplying the waveforms by the appropriate `FACTOR` and then adding the appropriate `OFFSET` and then using them to generate the necessary lookup tables and to have the pointers to the lookup tables placed in the pidtau vector at the correct times.

The waveforms must be of the step type. A vertex on any of the waveforms causes a new lookup table to be generated.

14.2 Tools for target vector elements

This section describes utility functions used for setting up the change list entries for a target vector element. Additional background material is in Sec. 2.10.7 and 2.10.8.

14.2.1 fetch_vertices

This function returns a copy of the vertices for the specified waveform. For "continuous" waveforms the copy returned is an exact duplicate of the set of vertices for the waveform. For a "step" waveform, extra vertices are optionally added so that the new set of vertices, when treated as a continuous waveform by the function `wv` (Sec. 14.2.6), has the correct values. To do this an additional vertex is added at one phase clock tick before every vertex except the first vertex. The Y axis value of this additional vertex is equal to the Y axis value of the next earlier vertex.

Calling format:

```
struct vertex *fetch_vertices(struct waveform *wavein,
                              struct waveform_variable *wavevarin,
                              int *numvertexout,
     int ADD_VERTICES)
```

Function Arguments:

- `wavein`: A pointer to the descriptor for the waveform. This would be typically:

  ```
  (struct waveform *)&waveforms[index]
  ```

  where `waveforms` is the global variable array of waveform descriptors.

- `wavevarin`: A pointer to the structure containing the data for the waveform that depends on the PCS setup. This includes the list of waveform vertices. This argument would typically be:

  ```
  (struct waveform_variable *)&phase->wavelist[index]
  ```

  where `phase` is the pointer to the phase descriptor.

- `numvertexout`: A pointer to the location to store the count of vertices in the returned array.

- `ADD_VERTICES`: An integer flag specifying whether or not extra vertices should be generated for integer step target waveforms to simulate a continuous waveform. A value of 1 for `ADD_VERTICES` indicates values should be added, 0 indicates do not add vertices.

Return values:

- The function returns an array of structures of type `vertex`. Each structure contains the X and Y values of one vertex. This structure is defined as follows.

  ```
  struct vertex {float x; float y;}
  ```

- The number of structures in the returned array is written to the location specified by the argument `numvertexout`.

14.2.2 fetch_external_vertices

This function returns a set of vertices for the given waveform. If the phase name is given, then the vertices for the given phase in the category given by the category identifier are returned. If the phase_name argument is NULL, then the vertices in the phases on the first phase sequence are returned for the waveform in the category given by the category identifier. In the latter case, the vertices for multiple phases are combined into one set with vertices added at each phase begin and phase end.

For "continuous" waveforms the copy returned is an exact duplicate of the set(s) of vertices for the waveform. For a "step" waveform, extra vertices are optionally added so that the new set of vertices, when treated as a continuous waveform by the function `wv` (Sec. 14.2.6), has the correct values. To do this an additional vertex is added at one phase clock tick before every vertex except the first vertex. The Y axis value of this additional vertex is equal to the Y axis value of the next earlier vertex.

Calling format:

```
struct vertex *fetch_external_vertices(char *name,
                                       char *cat_ident,
                                       char *phase_name,
                                       int *numvertexout,
             int ADD_VERTICES,
             int add_phase_start_time)
```

Function Arguments:

- `name`: A text string giving the waveform name.
- `cat_ident`: A text string giving the category identifier.
- `phase_name`: A text string giving the phase name if only vertices in that phase should be returned, or NULL if vertices over all phases used on the first phase sequence should be returned.

- `numvertexout`: A pointer to the location to store the count of vertices in the returned array.
- `ADD_VERTICES`: An integer flag specifying whether or not extra vertices should be generated for integer step target waveforms to simulate a continuous waveform. A value of 1 for `ADD_VERTICES` indicates values should be added, 0 indicates do not add vertices.
- `add_phase_start_time`: An integer flag specifying whether or not the phase starting time should be added to the X value for each vertex. A value of 0 indicates the vertices are with respect to the beginning of the (first) phase. A value of 1 indicates the vertices are with respect to the primary start time.

Return values:

- The function returns an array of structures of type `vertex`. Each structure contains the X and Y values of one vertex. This structure is defined as follows.

  ```
  struct vertex {float x; float y;}
  ```
- The number of structures in the returned array is written to the location specified by the argument `numvertexout`.

14.2.3 find_waveform_indices

Find the indices of the specified waveform in the shot phase list of waveforms and the master list of waveforms. The list of waveforms used by the shot phase contains the information about the waveforms that depends on the shot setup, such as the set of vertices. The master list of waveforms contains the information about each waveform that is fixed in the waveform descriptor provided in the algorithm code.

Calling format:

```
void find_waveform_indices(char *waveform, struct shotphase *phase,
                           char *message,
                           int *wpindex,int *windex)
```

Function Arguments:

- `waveform`: A text string specifying the waveform name.
- `phase`: The pointer to the descriptor of the shot phase.
- `message`: A text string that identifies the calling function that is printed as part of any error message that `find_waveform_indices` prints.

- `wpindex`: A pointer to the location to return the index (based at 0) of the waveform in the list of waveforms used by the algorithm assigned to the shot phase.
- `windex`: A pointer to the location to return the index (based at 0) of the waveform in the master list of waveforms defined for all algorithms.

Return values:

- Waveform indices are returned in `wpindex` and `windex` as described in the argument list.
- If the specified waveform cannot be located, this indicates a programming error. The function does not return. Instead, it prints an error message using the `msg_log` function (Sec. 14.5.2) and then aborts by calling `pcs_abort` (Sec. 14.6.38).

14.2.4 get_yvalue_at_time

Find the Y value from a list of vertices at a given time.

Calling format:

```
float get_yvalue_at_time(double time,struct vertex *vertices,
                         int num_vertices)
```

Function Arguments:

- `time`: A double value giving the time at which to return the Y value.
- `vertices`: The array of structures of type `vertex` which give the time and Y values.
- `num_vertices`: The number of vertices.

Return values:

- Calculated Y value at the given time.

14.2.5 sort_xvalues

This function sorts the input array of time values in ascending order. Any duplicate values are removed.

When two values are compared, they are first converted to an integer number of phase clock ticks so that the comparison is done using the values as they will be used on the real time processor.

Calling format:

```
void sort_xvalues(float xvalues[],  int xcount_in, int *xcount_out)
```

Function Arguments:

- `xvalues`: The array of values to be sorted.
- `xcount_in`: The count of values in the array `xvalues`.
- `xcount_out`: A pointer to the location to store the count of unique, sorted values in `xvalues` when the function returns.

Return values:

- When the function returns, the `xcount_out` values at the beginning of the `xvalues` array are the unique x values, sorted.

14.2.6 wv

Evaluate the waveform defined by the input set of vertices on a time base specified by a list of time values.

Each pair of vertices is connected by a line. Any of the time base values that fall between the pair of vertices are assigned the Y value that equals the Y axis value that the line has at that time. Any time values that fall before the time of the first vertex are assigned a value equal to the Y axis value of the first vertex. Any time values that fall after the time of the last vertex are assigned a value equal to the Y axis value of the last vertex.

Calling format:

```
float *wv(struct vertex *vertices,int count,
          float *timepoints, int numresult)
```

Function Arguments:

- `vertices`: The array of vertices. Each vertex is specified by a structure defined as follows.

  ```
  struct vertex {float x; float y;}
  ```

- `count`: The number of vertices in the input array.
- `timepoints`: The array of time values at which the vertices list should be evaluated.
- `numresult`: The number of time values in the `timepoints` array.

Return values:

- The function returns a pointer to an array of Y axis values that correspond to the time values given by the input argument array `timepoints`. The number of elements in the returned array is given by the input argument `numresult`. Note that when the calling function is finished with the values in the returned array, the memory allocated for the array must be freed by calling the function `free()`.

14.2.7 wvcontinuous

This function takes a list of vertices and generates the corresponding change list entries for a continuous target vector element.

Calling format:

```
int wvcontinuous(struct vertex *vertices,int numvertex,
 int tindex, int cpu_index, struct shotphase *phase)
```

Function Arguments:

- `vertices`: An array specifying the vertex locations.
- `numvertex`: The number of vertices provided in the `vertices` array.
- `tindex`: The index (based at 1) of the element of the target vector for which the change list entries should be generated.
- `cpu_index`: The index (based at 1) of the virtual CPU on which the target vector is located.
- `phase`: The pointer to the descriptor of the phase for which the change list is being modified.

Return values:

- The function returns 0 if there were no problems, 1 otherwise. If there was a problem, an error message is generated.

14.2.8 wvstep

This function takes a list of vertices and generates the corresponding change list entries for a step target vector element or a pointer target vector element.

Calling format:

```
void wvstep(struct vertex *vertices,
    struct ivertex *ivertices,
    int numvertex, int vertex_type,
            int tindex, int cpu_index, struct shotphase *phase,
            int target_type,int extra_integer)
```

Function Arguments:

- `vertices`: An array specifying the vertex locations. In this array the Y axis values are provided in floating point format. If the `target_type` is an integer step target vector element or a pointer target vector element, the Y axis values are cast to type `int` before adding them to the change list. This argument is ignored if `vertex_type` is 0. The structure is defined as follows.

  ```
  struct vertex {float x; float y;}
  ```

- `ivertices`: An array specifying the vertex locations. In this array the Y axis values are provided as type `int`. This argument is ignored if `vertex_type` is 1. The structure is defined as follows.

  ```
  struct ivertex {float x; int y;}
  ```

- `numvertex`: The number of vertices provided in the `vertices` array or the `ivertices` array, whichever is actually used.
- `vertex_type`: This argument must be 0 if the vertices are provided in the `ivertices` array. This argument must be 1 if the vertices are provided in the `vertices` array.
- `tindex`: The index (based at 1) of the element of the target vector for which the change list entries should be generated.
- `cpu_index`: The index (based at 1) of the virtual CPU on which the target vector is located.
- `phase`: The pointer to the descriptor of the phase for which the change list is being modified.
- `target_type`: The type of target vector element for which change list entries are to be generated.
  - `IST_CHANGE`: an integer step target vector element.
  - `FST_CHANGE`: a float step target vector element.

  - `PTRT_PARAMBLOCK_CHANGE`: an element in the pointer target vector that points to a location within a parameter data block.

- `extra_integer`: this value is ignored unless `target_type` is `PTRT_PARAMBLOCK_CHANGE`. In that case this value is recorded in the change list entry to be interpreted as the offset into the parameter data block for which the pointer should be generated.

Return values:

- This function returns no values. If there is an error in the execution of the function, an error message is generated and the function calls `pcs_abort` (Sec. 14.6.38).

14.2.9 wvmp

This function takes a waveform and scales it then adds an offset to create the time evolution of a target vector element value. This routine handles both continuous and step target vectors. If the waveform is a null pointer, then the scaling factor is ignored and a single point assigned to t=0 is generated with y value equal to the offset.

Calling format:

```
void wvmp(char *waveform, float offset, float factor,
          int tindex, int cpu_index, struct shotphase *phase,
          int target_type, int extra_integer)
```

Function Arguments:

- `waveform`: A text string specifying the waveform name.
- `offset`: The value to add to the scaled waveform.
- `factor`: The value to scale the waveform.
- `tindex`: The index (based at 1) of the element of the target vector for which the change list entries should be generated.
- `cpu_index`: The index (based at 1) of the virtual CPU on which the target vector is located.
- `phase`: The pointer to the descriptor of the phase for which the change list is being modified.
- `target_type`: The type of target vector element for which change list entries are to be generated.
  - `CT_CHANGE`: a continuous target vector element.

  - `IST_CHANGE`: an integer step target vector element.
  - `FST_CHANGE`: a float step target vector element.
  - `PTRT_PARAMBLOCK_CHANGE`: an element in the pointer target vector that points to a location within a parameter data block.

- `extra_integer`: this value is ignored unless `target_type` is `PTRT_PARAMBLOCK_CHANGE`. In that case this value is recorded in the change list entry to be interpreted as the offset into the parameter data block for which the pointer should be generated.

Return values:

- This function returns no values. If there is an error in the execution of the function, an error message is generated and the function calls `pcs_abort` (Sec. 14.6.38).

14.2.10 wv2mp

This function takes the product of two waveforms and scales and offsets it to create the time evolution of a target vector element value. This routine handles both continuous and step target vectors. By definition, the product of two continuous waveforms or the product of a step waveform and a continuous waveform makes a continuous waveform. This can be made into a continuous target vector but not a step target vector. The product of two step waveforms makes another step waveform. This can be made into either a continuous target or a step target.

Calling format:

```
void wv2mp(char *waveform1, char *waveform2,
           float offset, float factor,
           int tindex, int cpu_index, struct shotphase *phase,
           int target_type)
```

Function Arguments:

- `waveform1`: A text string specifying the first waveform name.
- `waveform2`: A text string specifying the second waveform name.
- `offset`: The value to add to the scaled product of the two waveforms.
- `factor`: The value to scale the product of the two waveforms.
- `tindex`: The index (based at 1) of the element of the target vector for which the change list entries should be generated.

- `cpu_index`: The index (based at 1) of the virtual CPU on which the target vector is located.
- `phase`: The pointer to the descriptor of the phase for which the change list is being modified.
- `target_type`: The type of target vector element for which change list entries are to be generated.
  - `CT_CHANGE`: a continuous target vector element.
  - `IST_CHANGE`: an integer step target vector element.
  - `FST_CHANGE`: a float step target vector element.

Return values:

- This function returns no values. If there is an error in the execution of the function, an error message is generated and the function calls `pcs_abort` (Sec. 14.6.38).

14.2.11 wvdp

This function computes the target vector element from the input waveform by dividing the waveform vertex y values into a constant and then adding a second value. This routine handles both continuous and step target vectors.

Calling format:

```
void wvdp(char *waveform_in, float offset, float factor,
          int tindex, int cpu_index, struct shotphase *phase,
          int target_type)
```

Function Arguments:

- `waveform_in`: A text string specifying the waveform name.
- `offset`: The value to add to the waveform after the division.
- `factor`: The value to divide the waveform vertices into.
- `tindex`: The index (based at 1) of the element of the target vector for which the change list entries should be generated.
- `cpu_index`: The index (based at 1) of the virtual CPU on which the target vector is located.

- `phase`: The pointer to the descriptor of the phase for which the change list is being modified.
- `target_type`: The type of target vector element for which change list entries are to be generated.
  - `CT_CHANGE`: a continuous target vector element.
  - `IST_CHANGE`: an integer step target vector element.
  - `FST_CHANGE`: a float step target vector element.

Return values:

- This function returns no values. If there is an error in the execution of the function, an error message is generated and the function calls `pcs_abort` (Sec. 14.6.38).

14.2.12 wvsignal

Calling format:

```
void wvsignal(int tindex, float offset, float factor,
     int cpu_index, struct shotphase *phase, char *typename,
     char *freqname, char *ampname, char *baselinename)
```

Function arguments:

- `tindex`: target vector index.
- `offset`: the value to add to the baseline waveform vertices.
- `factor`: the value to scale the baseline and amplitude waveform vertices.
- `cpu_index`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.).
- `phase`: The pointer to the descriptor of the phase for which the change list is being modified.
- `typename`: pointer to string identifying signal type waveform.
- `freqname`: pointer to string identifying signal frequency waveform.
- `ampname`: pointer to string identifying signal amplitude waveform.
- `baselinename`: pointer to string identifying baseline signal waveform.

This function computes the target vector from the input waveforms defining the type, frequency, and amplitude of a "signal-generator" signal. Add the result to the target generated by the baseline waveform. The amplitude and the baseline waveforms must be in the same units. The last vertex specifying a sine wave will time out after 3 seconds.

The value for the type waveform can be any of the following values:

- `0`: baseline value only.
- `1`: sine wave: maximum 3 seconds if last vertex.
- `2`: square wave: 6 vertices per period.
- `3`: triangle wave: 5 vertices per period.

14.2.13 make_pidtau

Calling format:

```
make_pidtau(char *waveform1, char *waveform2, char *waveform3,
            float offset1, float offset2, float offset3,
            float factor1, float factor2, float factor3,
            int tindex, int cpu_index, struct shotphase *phase)
```

Function arguments:

- `waveform1- 3`: strings giving the waveform names. The waveforms must be used by the algorithm used in `phase`. `waveform1` is the time constant for the initial low pass filter applied to the error vector value (p), `waveform2` is the derivative time constant waveform (d), `waveform3` is the integral time constant waveform (i).
- `offset1 - 3`: the value to add to the waveform vertices.
- `factor1 - 3`: the value to scale the waveform vertices.
- `tindex`: target vector index.
- `cpu_index`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.)
- `phase`: The pointer to the descriptor of the phase for which the change list is being modified.

This function is used to provide the data that will cause the necessary lookup tables for the PID calculation algorithm to be generated. Pointers to the lookup tables will be placed in the correct locations in the pidtau vector.

The waveforms provide the three necessary time constants. After the waveforms are scaled and offset the waveform units must be seconds. The waveforms must be of the step type.

14.2.14 wv_offset

This function takes a waveform and scales it then adds an offset to create the time evolution of a target vector element value. This function is intended to generate byte offset values. The waveform vertices would normally have small integral values. The vertices are converted to integers before use. The waveform vertices are multiplied by a byte offset that is computed from a data object descriptor and then another offset computed from a data object descriptor is added to the result. If the waveform is a null pointer, then the scaling factor is ignored and a single point assigned to t=0 is generated with y value equal to the offset. This routine generates values for step vector elements.

Calling format:

```
void wv_offset(char *waveform_in,
               STRUCT_DESCRIPTORS *offset_descriptor,
               STRUCT_DESCRIPTORS *factor_descriptor,
               int tindex,int cpu_index,struct shotphase *phase,
               int target_type,int extra_integer)
```

Function Arguments:

- `waveform_in`: A text string specifying the waveform name.
- `offset_descriptor`: a pointer to a data object descriptor used to generate the offset value added to the target vector. If the pointer is NULL, then the offset value is set to 0.
- `factor_descriptor`: A pointer to a data object descriptor used to generate the multiplier value. If the pointer is NULL, then the multiplier value is set to 0.
- `tindex`: The index (based at 1) of the element of the target vector for which the change list entries should be generated.
- `cpu_index`: The index (based at 1) of the virtual CPU on which the target vector is located.
- `phase`: The pointer to the descriptor of the phase for which the change list is being modified.
- `target_type`: The type of target vector element for which change list entries are to be generated.
  - `IST_CHANGE`: an integer step target vector element.
  - `FST_CHANGE`: a float step target vector element.

  - `PTRT_PARAMBLOCK_CHANGE`: an element in the pointer target vector that points to a location within a parameter data block.

- `extra_integer`: this value is usually zero unless `target_type` is `PTRT_PARAMBLOCK_CHANGE`. In that case this value is recorded in the change list entry to be interpreted as the offset into the parameter data block for which the pointer should be generated.

Return values:

- This function returns no values. If there is an error in the execution of the function, an error message is generated and the function calls `pcs_abort` (Sec. 14.6.38).

14.3 Parameter data block support routines

14.3.1 putblock

This function is used in the waveform server to create a parameter data block. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This is the most generic function available for parameter data block creation. A summary of other, similar, functions is given in Sec. 2.11.4.2. A detailed description of the parameter data block facility is given in Sec. 2.11.

Calling format:

```
int putblock(
    struct shotphase *phase,
    char *name,
    char *options,
    int block_order_index,
    STRUCT_DESCRIPTORS *descriptor,
    void *data)
```

Function arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored. The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.

- `options`: A character string giving options as keywords (See Sec. 2.11.4.4.

- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.
- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.
- `descriptor`: Pointer to the data object descriptor for the data that will be stored in the parameter data block. This argument is used to determine the number of bytes to copy into the parameter block. It is also used to determine how to copy the data to the real time processor or for archiving. Section. 2.11.4.5 has a detailed discussion of data object descriptors.
- `data`: Pointer to the block of bytes to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns the number of bytes copied into the parameter data block. If this value is 0, the parameter data block creation may have failed.

Example:
Section 2.11.3 contains a discussion of examples for various usages of parameter data blocks.

14.3.2 putblock_chars

This function is used in the waveform server to create a parameter data block that contains an array of type `char`. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This function performs the same task as the generic parameter data block creation function `putblock` (Sec. 14.3.1) except that this function doesn't require a data object descriptor as an argument. Instead, this function takes an integer argument specifying the number of elements in the array to be copied to the parameter data block. This function simply creates the necessary data object descriptor and then calls `putblock`.

A summary of other, similar, functions is given in Sec. 2.11.4.2. A detailed description of the parameter data block facility is given in Sec. 2.11.

This function is best used to create a byte array or a single character string. Though it can be used to create a concatenated set of character strings all of the same size, it is better to use `putblock_strings` to create a block with multiple strings.

Calling format:

```
int putblock_chars(
    struct shotphase *phase,
    char *name,
    char *options,
    int block_order_index,
    int count,
    char *data)
```

Function arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored. The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.
- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.
- `options`: A character string giving options as keywords (See Sec. 2.11.4.4.
- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.
- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.
- `count`: The number of bytes in the array to be copied into the parameter data block.
- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns 0.
Examples:

```
1. Create a parameter data block containing an array of 25 byte values.
{
    int ii;
    char barray[25];

    for(ii=0;ii<25;ii++) barray[ii] = 'a';

    putblock_chars(phase,               /* shot phase descriptor */
                "CHAR BLOCK EXAMPLE",   /* name */
                "USER|ARCHIVE|RTCPUS=A",  /* options */
                1,                      /* insert_order */
                25,                     /* num chars in data */
                barray);                /* pointer to data */
}
```

2. Create a parameter data block containing a single character string.

```
{
    char label = 'file_example.txt'

    putblock_chars(phase,               /* shot phase descriptor */
                "SINGLE STRING BLOCK EXAMPLE",  /* name */
                "USER|ARCHIVE|RTCPUS=B",  /* options */
                BB_ALG_DATA_BLOCK,      /* insert_order */
                strlen(label)+1,        /* num bytes in data */
                label);                 /* pointer to data */
}
```

3. Create a parameter data block containing multiple strings all of the same size.

```
{
    char labels[2][40];

    memset((char *)labels,0,2*40);

    strcpy(labels[1-1], "label number 1");
    strcpy(labels[2-1], "label number 2");

    putblock_chars(phase,               /* shot phase descriptor */
                "CHAR BLOCK EXAMPLE",   /* name */
                "USER|ARCHIVE|RTCPUS=B",  /* options */
                BB_ALG_DATA_BLOCK,      /* insert_order */
                2 * 40,                 /* num bytes in data */
                labels);                /* pointer to data */
}
```

14.3.3 putblock_shorts

This function is used in the waveform server to create a parameter data block that contains an array of type `short`. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This function performs the same task as the generic parameter data block creation function `putblock` (Sec. 14.3.1) except that this function doesn't require a data object descriptor as an argument. Instead, this function takes an integer argument specifying the number of elements in the array to be copied to the parameter data block. This function simply creates the necessary data object descriptor and then calls `putblock`.

A summary of other, similar, functions is given in Sec. 2.11.4.2. A detailed description of the parameter data block facility is given in Sec. 2.11.

Calling format:

```
int putblock_shorts(
    struct shotphase *phase,
    char *name,
    char *options,
    int block_order_index,
    int count,
    short *data)
```

Function arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored. The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.

- `options`: A character string giving options as keywords (See Sec. 2.11.4.4.

- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.

- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.

- `count`: The number of elements in the array to be copied into the parameter data block.
- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns 0.
Example:

```
{
    int ii,stat;
    short sarray[25];

    for(ii=0;ii<25;ii++) sarray[ii] = 0;

    putblock_shorts(phase,              /* shot phase descriptor */
                "SHORT BLOCK EXAMPLE",  /* name */
                "USER|ARCHIVE|RTCPUS=A",  /* options */
                1,                      /* insert_order */
                25,                     /* num chars in data */
                sarray);                /* pointer to data */
}
```

14.3.4 putblock_ints

This function is used in the waveform server to create a parameter data block that contains an array of type `int`. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This function performs the same task as the generic parameter data block creation function `putblock` (Sec. 14.3.1) except that this function doesn't require a data object descriptor as an argument. Instead, this function takes an integer argument specifying the number of elements in the array to be copied to the parameter data block. This function simply creates the necessary data object descriptor and then calls `putblock`.

A summary of other, similar, functions is given in Sec. 2.11.4.2. A detailed description of the parameter data block facility is given in Sec. 2.11.

Calling format:

```
int putblock_ints(
    struct shotphase *phase,
    char *name,
    char *options,
    int block_order_index,
    int count,
    int *data)
```

Function arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored. The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.
- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.
- `options`: A character string giving options as keywords (See Sec. 2.11.4.4.
- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.
- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.
- `count`: The number of elements in the array to be copied into the parameter data block.
- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns 0.
Examples:

1. Create a parameter data block containing a single integer. This might be, for instance, a block that is part of a static data item. This block might contain a value that indicates the size of a second block.

```
{
    int count;

    count = 0;
    putblock_ints(phase,                    /* shot phase descriptor */
                "INT BLOCK EXAMPLE",    /* name */
                "USER|ARCHIVE|RTCPUS=A",  /* options */
                1,                        /* insert_order */
                1,                        /* num ints in data */
                &count);                  /* pointer to data */
}
```

2. Store an array of 25 integers.

```
{
    int ii;
    int iarray[25];

    for(ii=0;ii<25;ii++) iarray[ii] = 0;

    putblock_ints(phase,                    /* shot phase descriptor */
                "INT BLOCK EXAMPLE",    /* name */
                "USER|ARCHIVE|RTCPUS=A",  /* options */
                1,                        /* insert_order */
                25,                       /* num ints in data */
                iarray);                  /* pointer to data */
}
```

14.3.5 putblock_longs

This function is used in the waveform server to create a parameter data block that contains an array of type `long`. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This function performs the same task as the generic parameter data block creation function `putblock` (Sec. 14.3.1) except that this function doesn't require a data object descriptor as an argument. Instead, this function takes an integer argument specifying the number of elements in the array to be copied to the parameter data block. This function simply creates the necessary data object descriptor and then calls `putblock`.

A summary of other, similar, functions is given in Sec. 2.11.4.2. A detailed description of the parameter data block facility is given in Sec. 2.11.

Calling format:

```
int putblock_longs(
    struct shotphase *phase,
    char *name,
    char *options,
    int block_order_index,
    int count,
    long *data)
```

Function arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored. The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.
- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.
- `options`: A character string giving options as keywords (See Sec. 2.11.4.4.
- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.
- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.
- `count`: The number of elements in the array to be copied into the parameter data block.
- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns 0.
Example:

```
{
    int ii;
    long larray[25];
    for(ii=0;ii<25;ii++) larray[ii] = 0;

    putblock_longs(phase),              /* shot phase descriptor */
                "LONG BLOCK EXAMPLE",   /* name */
                "USER|ARCHIVE|RTCPUS=A",  /* options */
                1,                      /* insert_order */
                25,                     /* num longs in data */
                larray);                /* pointer to data */
}
```

14.3.6 putblock_floats

This function is used in the waveform server to create a parameter data block that contains an array of type `float`. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This function performs the same task as the generic parameter data block creation function `putblock` (Sec. 14.3.1) except that this function doesn't require a data object descriptor as an argument. Instead, this function takes an integer argument specifying the number of elements in the array to be copied to the parameter data block. This function simply creates the necessary data object descriptor and then calls `putblock`.

A summary of other, similar, functions is given in Sec. 2.11.4.2. A detailed description of the parameter data block facility is given in Sec. 2.11.

Calling format:

```
int putblock_floats(
    struct shotphase *phase,
    char *name,
    char *options,
    int block_order_index,
    int count,
    float *data)
```

Function arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored. The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.

- `options`: A character string giving options as keywords (See Sec. 2.11.4.4.
- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.
- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.
- `count`: The number of elements in the array to be copied into the parameter data block.
- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns 0.
Example:

```
{
    int ii;
    float farray[25];

    for(ii=0;ii<25;ii++) farray[ii] = 0.0;

    putblock_floats(phase,              /* shot phase descriptor */
                "FLOAT BLOCK EXAMPLE",  /* name */
                "USER|ARCHIVE|RTCPUS=A",  /* options */
                1,                      /* insert_order */
                25,                     /* num floats in data */
                farray);                /* pointer to data */
}
```

14.3.7 putblock_doubles

This function is used in the waveform server to create a parameter data block that contains an array of type `double`. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This function performs the same task as the generic parameter data block creation function `putblock` (Sec. 14.3.1) except that this function doesn't require a data object descriptor as an argument. Instead, this function takes an integer argument specifying the number of elements in the array to be copied to the parameter data block. This function simply creates the necessary data object descriptor and then calls `putblock`.

A summary of other, similar, functions is given in Sec. 2.11.4.2. A detailed description of the parameter data block facility is given in Sec. 2.11.

Calling format:

```
int putblock_doubles(
    struct shotphase *phase,
    char *name,
    char *options,
    int block_order_index,
    int count,
    double *data)
```

Function arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored. The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.

- `options`: A character string giving options as keywords (See Sec. 2.11.4.4.

- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.

- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.

- `count`: The number of elements in the array to be copied into the parameter data block.

- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns 0.

Example:

```
{
    int ii;
    double farray[25];

    for(ii=0;ii<25;ii++) farray[ii] = 0.0;

    putblock_doubles(phase,             /* shot phase descriptor */
                "DOUBLE BLOCK EXAMPLE", /* name */
                "USER|ARCHIVE|RTCPUS=A",  /* options */
                1,                      /* insert_order */
                25,                     /* num doubles in data */
                farray);                /* pointer to data */
}
```

14.3.8 putblock_strings

This function is used in the waveform server to create a parameter data block that contains an array of type `char`. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This function performs the same task as the generic parameter data block creation function `putblock` (Sec. 14.3.1) except that this function doesn't require a data object descriptor as an argument. Instead, this function takes an integer argument specifying the number of elements in the array to be copied to the parameter data block. This function adds up the lengths of the strings, combines them with NULL characters as separators, creates the necessary data object descriptor, and then calls `putblock`.

This function is best used to create multiple strings in a single block. If the string array ends with a NULL string, then the count can be 0 and this function will determine how many strings there are by counting until the NULL string.

A summary of other, similar, functions is given in Sec. 2.11.4.2. A detailed description of the parameter data block facility is given in Sec. 2.11.

Calling format:

```
int putblock_strings(
    struct shotphase *phase,
    char *name,
    char *options,
    int block_order_index,
    int count,
    char **data)
```

Function arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored. The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.
- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.
- `options`: A character string giving options as keywords (See Sec. 2.11.4.4.
- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.
- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.
- `count`: The number of strings in the array to be copied into the parameter data block.
- `data`: Pointer to the string array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns 0.
Example:

```
{
    int ii;
    char strings[] = { "string1", "string2"};

    putblock_strings(phase,             /* shot phase descriptor */
                "STRING BLOCK EXAMPLE", /* name */
                "USER|ARCHIVE|RTCPUS=A",  /* options */
                1,                      /* insert_order */
```

```
               2,                      /* num strings in data */
               strings);               /* pointer to array */
}
```

14.3.9 putblock_labeled_int_array

This function is used in the waveform server to create a group of parameter data blocks that are associated with the standard static data item (Sec. 2.12) of type `labeled int array`. It should be called in an algorithm's `alg_parameters` function.

A static data item (Sec. 2.10.1.2) should also be created which uses the `generic_editor` to display the data, and the waveform descriptor should be set to `SD_LABELED_INT_ARRAY` so that the data can be restored correctly even if the size of the array changes in the future.

A summary of other, similar, functions is given in Sec. 2.11.4.2. A detailed description of the parameter data block facility is given in Sec. 2.11.

Calling format:

```
int putblock_labeled_int_array(
    struct shotphase *phase,
    char *name,
    char *options,
    int block_order_index,
    int count,
    int *data,
    char **labels,
    int *substitute_data);
```

Function arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored. The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.
- `name`: A string that specifies the name of the data item which describes this set of parameter data blocks. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.
- `options`: A character string giving options as keywords (See Sec. 2.11.4.4.
- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.

- `count`: The number of elements in the array to be copied into the parameter data block.
- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.
- `labels`: An array of labels, one per array value, which is used to match up data values when this data item is restored.
- `substitute_data`: An array of values that are used to substitute for values that are not in the archived data block.

Return value:
The function returns 0.
Example:

```
Define the static data item:

#ifdef WAVEFORMS
{/* StaticData */
 0,                          /* ignored */
 {0, 0, 0.0, 0.0},           /* ignored */
 "labeled int array",        /* label identifying param data, 20 chars max */
 "generic_editor",           /* name of idl routine to handle this data */
 " ",                        /* ignored */
 " ",                        /* ignored */
 1.0,                        /* ignored */
 0.0,                        /* ignored */
 " ",                        /* ignored */
 SD_LABELED_INT_ARRAY,       /* indicates not a waveform, 40 chars max */
 C_CODE,                     /* control category index */
 A_CODE,                     /* control algorithm index */
 {0},                        /* target vectors affected */
 0,                          /* ignored */
 0,                          /* ignored */
 {0.0},                      /* ignored */
 0,                          /* ignored */
 0,                          /* ignored */
 1,                          /* subset number */
 0                           /* ignored */
},
#endif

Create the parameter data in the alg_parameters function:
{
    int ii;
    int values[3];

    char *labels[1] = { "value 1", "value 2", "value 3" };

    for(ii=0;ii<3;ii++) values[ii] = 0;

    putblock_labeled_int_array(phase,/* pointer to the phase descriptor */
                               "labeled int array",/* name */
                               "RTCPUS=A|ARCHIVE",/* options */
                               BA_ALG_DATA_BLOCK,/* block_order_index */
                               3,/* count */
                               values,/* data */
                               labels,/* labels */
                               values);/* substitution_data */
}
```

14.3.10 putblock_labeled_float_array

This function is used in the waveform server to create a group of parameter data blocks that are associated with the standard static data item (Sec. 2.12) of type `labeled float array`. It should be called in an algorithm's `alg_parameters` function.

A static data item (Sec. 2.10.1.2) should also be created which uses the `generic_editor` to display the data, and the waveform descriptor should be set to `SD_LABELED_FLOAT_ARRAY` so that the data can be restored correctly even if the size of the array changes in the future.

A summary of other, similar, functions is given in Sec. 2.11.4.2. A detailed description of the parameter data block facility is given in Sec. 2.11.

Calling format:

```
int putblock_labeled_float_array(
    struct shotphase *phase,
    char *name,
    char *options,
    int block_order_index,
    int count,
    float *data,
    char **labels,
    float *substitute_data);
```

Function arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored. The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.
- `name`: A string that specifies the name of the data item which describes this set of parameter data blocks. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.
- `options`: A character string giving options as keywords (See Sec. 2.11.4.4.
- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.
- `count`: The number of elements in the array to be copied into the parameter data block.
- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.
- `labels`: An array of labels, one per array value, which is used to match up data values when this data item is restored.
- `substitute_data`: An array of values that are used to substitute for values that are not in the archived data block.

Return value:
The function returns 0.
Example:

```
Define the static data item:

#ifdef WAVEFORMS
{/* StaticData */
 0,                         /* ignored */
 {0, 0, 0.0, 0.0},          /* ignored */
 "labeled float array",   /* label identifying param data, 20 chars max */
 "generic_editor",        /* name of idl routine to handle this data */
 " ",                     /* ignored */
 " ",                     /* ignored */
 1.0,                     /* ignored */
 0.0,                     /* ignored */
 " ",                     /* ignored */
 SD_LABELED_FLOAT_ARRAY,  /* indicates not a waveform, 40 chars max */
 C_CODE,                  /* control category index */
 A_CODE,                  /* control algorithm index */
 {0},                     /* target vectors affected */
 0,                       /* ignored */
 0,                       /* ignored */
 {0.0},                   /* ignored */
 0,                       /* ignored */
 0,                       /* ignored */
 1,                       /* subset number */
 0                        /* ignored */
},
#endif

Create the parameter data in the alg_parameters function:
{
{
    int ii;
    float values[3];

    char *labels[1] = { "value 1", "value 2", "value 3" };

    for(ii=0;ii<3;ii++) values[ii] = 0.0;

    putblock_labeled_float_array(&(phase->parameters),/* parameter area */
                                 "labeled float array",/* name */
                                 "USER|ARCHIVE|RTCPUS=A",/* options */
                                 BA_ALG_DATA_BLOCK,/* block_order_index */
                                 3,/* count */
                                 values,/* data */
                                 labels,/* labels */
                                 values);/* substitution_data */
}
```

14.3.11 putblock_labeled_structure

This function is used in the waveform server to create a group of parameter data blocks that are associated with the standard static data item (Sec. 2.12) of type `labeled structure`. It should be called in an algorithm's `alg_parameters` function.

A static data item (Sec. 2.10.1.2) should also be created which uses the `generic_editor` to display the data, and the waveform descriptor should be set to `SD_LABELED_STRUCTURE` so that the data can be restored correctly even if the size of the structure changes in the future.

A summary of other, similar, functions is given in Sec. 2.11.4.2. A detailed description of the parameter data block facility is given in Sec. 2.11.

Calling format:

```
int putblock_labeled_structure(
    struct shotphase *phase,
    char *name,
    char *options,
    int block_order_index,
    int count,
    STRUCT_DESCRIPTORS *descriptor,
    int *data,
    char **labels,
    int *substitute_data);
```

Function arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored. The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the data item which describes this set of parameter data blocks. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.

- `options`: A character string giving options as keywords (See Sec. 2.11.4.4.

- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.

- `count`: The number of labels passed in. If the string array ends with a NULL string, then the count can be 0 and this function will determine how many strings there are by counting until the NULL string.

- `descriptor`: Pointer to the data object descriptor for the data that will be stored in the parameter data block. This argument is used to determine the number of bytes to copy into the parameter block. It is also used to determine how to copy the data to the real time processor or for archiving. Section. 2.11.4.5 has a detailed discussion of data object descriptors.
- `data`: Pointer to the structure to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.
- `labels`: An array of labels, one per structure value, which is used to match up data values when this data item is restored.
- `substitute_data`: A structure of values that are used to substitute for values that are not in the archived data block.

Return value:
The function returns 0.
Current limitations:

- Embedded arrays (except char arrays) and structures are considered multiple values so there must be a label for each element of an array or structure.
- A single char value is considered a number rather than a string. IDL uses a range of 0-255 for its BYTE value so the C equivalent of unsigned char should be used as the variable type in the structure.
- A char array of at least size 2 is considered a character string. If the structure changes, the size of the char array can be increased, but should not decreased else archived data will not get restored.

Example:

```
Define the structure:

#ifdef CONTROLDEF
#define GEN_params \
SDSTART( struct params {     ,D_params      ) \
DGEN   (   int   value1;     ,INTTYPE,     1) \
DGEN   (   float value2;     ,FLOATTYPE,   1) \
DGEN   (   float value3;     ,FLOATTYPE,   1) \
SDEND  (               };                   )
GEN_params
```

```
#endif

Define the static data item:

#ifdef WAVEFORMS
{/* StaticData */
 0,                        /* ignored */
 {0, 0, 0.0, 0.0},         /* ignored */
 "labeled structure",      /* label identifying param data, 20 chars max */
 "generic_editor",         /* name of idl routine to handle this data */
 " ",                      /* ignored */
 " ",                      /* ignored */
 1.0,                      /* ignored */
 0.0,                      /* ignored */
 " ",                      /* ignored */
 SD_LABELED_STRUCTURE,     /* indicates not a waveform, 40 chars max */
 C_CODE,                   /* control category index */
 A_CODE,                   /* control algorithm index */
 {0},                      /* target vectors affected */
 0,                        /* ignored */
 0,                        /* ignored */
 {0.0},                    /* ignored */
 0,                        /* ignored */
 0,                        /* ignored */
 1,                        /* subset number */
 0                         /* ignored */
},
#endif

Create the parameter data in the alg_parameters function:
{
    GEN_params
    struct params params;

    char *labels[1] = { "value 1", "value 2", "value 3" };

    params.value1 = 1;
    params.value2 = 1.0;
    params.value3 = 500.0;

    D_params[0].size = 1;
    putblock_labeled_structure(&(phase->parameters),/* parameter area */
                               "labeled structure",/* name */
                               "USER|ARCHIVE|RTCPUS=A",/* options */
                               BA_ALG_DATA_BLOCK,/* block_order_index */
                               3,/* count */
                               D_params,/* descriptor */
                               (void *)params,/* data */
                               labels,/* labels */
                               (void *)values);/* substitution_data */
```

14.3.12 putblock_labeled_matrices

This function is used in the waveform server to create a group of parameter data blocks that are associated with the standard static data item (Sec. 2.12) of type `labeled matrices`. It should be called in an algorithm's `alg_parameters` function.

A static data item (Sec. 2.10.1.2) should also be created which uses the `matrix_editor` to display the data, and the waveform descriptor should be set to `SD_LABELED_MATRICES` so that the data can be restored correctly even if the size of the matrix changes in the future.

A summary of other, similar, functions is given in Sec. 2.11.4.2. A detailed description of the parameter data block facility is given in Sec. 2.11.

Calling format:

```
int putblock_labeled_matrices(
    struct shotphase *phase,
    char *name,
    char *options,
    int block_order_index,
    int num_matrices,
    int num_matrix_rows,
    int num_matrix_cols,
    char *matrix_names,
    char *matrix_row_names,
    char *matrix_col_names,
    float *matrix_data,
    int *substitute_data);
```

Function arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored. The pointer to the shot phase descriptor is an argument to the `alg_parameters`

(Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the data item which describes this set of parameter data blocks. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.
- `options`: A character string giving options as keywords (See Sec. 2.11.4.4.
- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.
- `num_matrices`: The number of matrices passed in.
- `num_matrix_rows`: The number of rows in the matrix.
- `num_matrix_cols`: The number of columns in the matrix.
- `matrix_names`: The names of the matrices that get displayed in the matrix editor. This is a 2-dimensional char array:

  ```
  char matrix_names[num_matrices][MATRIX_MAX_RC_NAME_SIZE]
  ```

- `matrix_row_names`: The names of the rows that get displayed in the matrix editor. This is a 2-dimensional char array:

  ```
  char matrix_names[num_matrix_rows][MATRIX_MAX_RC_NAME_SIZE]
  ```

- `matrix_col_names`: The names of the columns that get displayed in the matrix editor. This is a 2-dimensional char array:

  ```
  char matrix_names[num_matrix_cols][MATRIX_MAX_RC_NAME_SIZE]
  ```

- `data`: Pointer to the float array. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.
- `substitute_data`: A float array the size of one matrix of values that are used to substitute for values that are not in the archived matrices.

Return value:
The function returns 0.
Example:

```
Define the static data item:

#ifdef WAVEFORMS
{/* StaticData */
 0,                         /* ignored */
 {0, 0, 0.0, 0.0},          /* ignored */
 "C matrices",              /* label identifying param data, 20 chars max */
 "matrix_editor",           /* name of idl routine to handle this data */
 "(f12.4)",                 /* format for matrix numbers */
 " ",                       /* ignored */
 1.0,                       /* ignored */
 0.0,                       /* ignored */
 " ",                       /* ignored */
 SD_LABELED_MATRICES,       /* indicates not a waveform, 40 chars max */
 C_CODE,                    /* control category index */
 A_CODE,                    /* control algorithm index */
 {0},                       /* target vectors affected */
 0,                         /* ignored */
 0,                         /* ignored */
 {0.0},                     /* ignored */
 0,                         /* ignored */
 0,                         /* ignored */
 1,                         /* subset number */
 0                          /* ignored */
},
#endif

Create the parameter data in the alg_parameters function:
/*
------------------------------------------------------------------------------
PARAMETER:  "C Matrix ..."
------------------------------------------------------------------------------
*/
{
#define NUM_MATRICES 2
#define NUM_MATRIX_ROWS 4
#define NUM_MATRIX_COLS 3
```

```
#define MATRIX_MAX_RC_NAME_SIZE 40
char matrix_names[NUM_MATRICES][MATRIX_MAX_RC_NAME_SIZE];
char row_names[NUM_MATRIX_ROWS][MATRIX_MAX_RC_NAME_SIZE];
char col_names[NUM_MATRIX_COLS][MATRIX_MAX_RC_NAME_SIZE];
float matrices[NUM_MATRICES][NUM_MATRIX_ROWS][NUM_MATRIX_COLS];
int ii,jj,kk;

/*
initializations
*/
    memset((char *)matrix_names,0,NUM_MATRICES*MATRIX_MAX_RC_NAME_SIZE);
    memset((char *)row_names,0,NUM_MATRIX_ROWS*MATRIX_MAX_RC_NAME_SIZE);
    memset((char *)col_names,0,NUM_MATRIX_COLS*MATRIX_MAX_RC_NAME_SIZE);
    for(ii=0;ii<NUM_MATRICES;ii++)
    {
        for(jj=0;jj<NUM_MATRIX_ROWS;jj++)
            for(kk=0;kk<NUM_MATRIX_COLS;kk++)
                matrices[ii][jj][kk] = 0.0;
    }
/*
initialize matrix names
*/
    for(jj = 0; jj < NUM_MATRICES; jj++)
        sprintf(matrix_names[jj],"Matrix %d",jj);
/*
initialize matrix row names
*/
    strcpy(row_names[1-1],"row 1
    strcpy(row_names[2-1],"row 2");
    strcpy(row_names[3-1],"row 3");
    strcpy(row_names[4-1],"row 4");
/*
initialize matrix column names
*/
    strcpy(col_names[1-1],"col 1");
    strcpy(col_names[2-1],"col 2");
    strcpy(col_names[3-1],"col 3");
/*
write to waveserver's memory
*/
    putblock_labeled_matrices(
        phase,  /* address of phase descriptor */
        "C matrices", /* matrix label */
        "USER|ARCHIVE|RTCPUS=A",/* options */
        BA_CMATRIX, /* block order index */
        NUM_MATRICES, /* num s matrices */
        NUM_MATRIX_ROWS, /* num s matrix rows */
        NUM_MATRIX_COLS, /* num s matrix cols */
        (char *)matrix_names, /* matrix names */
        (char *)row_names, /* matrix row names */
        (char *)col_names, /* matrix column names */
        (float *)matrices, /* matrix data */
        (float *)NULL); /* substitute data is all zeros */

#undef NUM_MATRICES
#undef NUM_MATRIX_ROWS
#undef NUM_MATRIX_COLS
#undef MATRIX_MAX_RC_NAME_SIZE
}
```

14.3.13 putblock_offsets

This function creates a “byte offset list” parameter data block. The parameter data block created contains an array of integers, each of which is the byte offset of a parameter data block from the beginning of the PCS memory heap. The list of offsets is for the parameter data blocks of the phase containing the byte offset list block that are located on a specified virtual CPU. The virtual CPU is one of the CPUs used by the category containing the phase for which the byte offset block is created.

The values in a byte offset list parameter data block are useful with the interprocessor communication functions (Sec. B). These functions take an argument that specifies the offset from the beginning of the PCS memory heap (Sec. 2.9.3) of the location to which data should be copied. It is convenient to create a scratch parameter data block to serve as a buffer to receive data from another real time CPU. A byte offset list parameter data block can be used on the CPU that is transmitting the data to determine the offset to the buffer in the scratch parameter data block. Further discussion is in Sec. B.

Note that a byte offset list parameter data block is not an ordinary block where the algorithm programmer provides the data to be stored there. Also, the algorithm programmer doesn’t provide the parameter data block name. Instead, the name “block offsets for CPUx” is used, where “x” is replaced with the appropriate virtual CPU letter. This is a special parameter data block name. The infrastructure code recognizes this block name and fills the block with the appropriate offset values. The following is the content of the block in the waveform server.

- The `io_handler` copy of the parameter data block contains a single integer equal to 0.
- The work queue copy of the parameter data block is a model of the actual block that will be on the real time processor. The number of integers in the block is the same as will be on the real time processor but the integer values are offsets from the beginning of the phase's parameter data area rather than the complete offset from the beginning of the PCS memory heap.

So, the content of the parameter data block is really only useful on the real time processor.

Calling format:

```
int putblock_offsets(
    struct shotphase *phase,
    int vcpu,
    char *options,
    int insert_order)
```

Function Arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored.
- `vcpu`: The virtual CPU where the parameter data blocks are located for which the offset block will contain byte offset values. The virtual CPU should be specified using macros of the form `CPUx`, where `x` is an uppercase letter, `A`, `B`, etc (Sec. 2.9.1).
- `options`: A character string giving options as keywords (See Sec. 2.11.4.4. For a byte offset list parameter data block, usually the options would indicate the virtual CPUs on which the parameter data block should be placed.
- `block_order_index`: The block order index (Sec. 2.11.4.7) of the parameter data block to be created.

Return values:
This function returns 0.

Here is some example code. In this example, data are to be transmitted from CPU B to CPU A. The algorithm identifier is `example` and the category identifier is `mycategory`.

1. In the `CONTROLDEFS` section of the algorithm master file, create the block order index macro for the byte offset list parameter data block which will be placed on CPU B.

   ```
   #define BB_EXAMPLE_BLOCK_OFFSETS_CPUA (3)
   ```

Create the block order index macro of the scratch parameter data block to which the data will be copied. This block will be on CPU A.

```
#define BA_EXAMPLE_SCRATCH_BUFFER (5)
```

2. In the function `example_parameters`, create the byte offset list parameter data block. Also, in the `example_parameters` function create the scratch parameter data block that will receive the data. The `alg_parameters` function (Sec. 2.11.4.1) is the appropriate place to create these blocks because the blocks only need to be created once when the shot phase is initialized. Their contents don't depend on any raw data that could be modified by the PCS user. Here, the scratch parameter block is the correct size to hold a structure of type `scratch_buffer`. The structure and its descriptor (Sec. 2.11.4.5) should be defined in the `CONTROLDEFS` section of the algorithm master file.

```
GEN_scratch_buffer

putblock_offsets(phase,
                 CPUA,
                 "RTCPUS=B",
                 BB_EXAMPLE_BLOCK_OFFSETS_CPUA);
putblock(phase,
         "scratch area CPU A",
         "RTCPUS=A|SCRATCH",
         BA_EXAMPLE_SCRATCH_BUFFER,
         D_scratch_buffer,
         NULL);
```

3. On CPU B, copy data to CPU A. Use the byte offset list parameter block to obtain the offset to the target buffer.

```
int offset,error_flag;
int *block_offsets;

block_offsets =
   (int *)rtheap->current_parameter_data[CATB_MYCATEGORY-1]\
                  [BB_EXAMPLE_BLOCK_OFFSETS_CPUA-1];
offset = block_offsets[BA_EXAMPLE_SCRATCH_BUFFER - 1];
error_flag = rtcipc_write(rtheap,
             CPUA_MYCATEGORY,offset,data_to_be_copied,
             data_length,NULL);
```

14.3.14 getblock

This routine returns all the information for the parameter data block given the block's name. The block's information is put into the PARAM_BLOCK_INFO structure. The return value is a pointer to the block's data.

Calling format:

```
PARAM_BLOCK_INFO block_info;
    block_ptr = (block_type *)getblock(phase,name,&block_info);
```

Function Arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored.
- `char *block_name`: The name of the parameter data block.
- `PARAM_BLOCK_INFO *block_info`: Pointer to the structure that contains all the information about the block. The fields in this structure include
  - `blockstart`: the byte offset into the memory where the block starts.
  - `int blocksize`: the number of bytes that the block uses including header information. If the block does not exist, this value will be zero.
  - `int flags`: the block flags bitmask.
  - `int rtcpus`: a bitmask indicating which real time virtual cpus the block is to be copied to.
  - `int hostcpus`: a bitmask indicating which virtual host_cpu processes the block is to be copied to.
  - `int memory_region`: the memory region the block is copied to on the real time processor.
  - `int insert_order`: the block storage order (See Sec. 2.11.4.7).
  - `int mem_align`: the alignment of the block (See Sec. 2.11.4.6).
  - `int numdescs`: the number of descriptors used to describe the block.
  - `int data_size`: the number of bytes of data in the block.
  - `STRUCT_DESCRIPTORS *descriptors`: the pointer to the location in the block where the array of block descriptor structures start (See Sec. 2.11.4.5).

Return value:

- The return value of the routine is a pointer to the data in the parameter data block. If the parameter data block couldn't be located, or the block is a scratch block, a null pointer is returned. The programmer needs to cast the return value to the desired variable type.

Note that the parameter data block pointer returned can become invalid if any changes are made to the parameter data in the phase in which the parameter data block is located. (See Sec. 2.11.4.9 for more discussion of this point.)

14.3.15 getblock_data_ptr

This routine returns the pointer to the data in the parameter data block given the block's name. It is the equivalent of calling `getblock` with the last argument as NULL.

Calling format:

```
block_ptr = (block_type *)get_param_block_data_ptr(phase,block_name);
if(block_ptr != NULL)
{
    ...
}
```

Function Arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored.
- `char *block_name`: The name of the parameter data block.

Return value:

- The return value of the routine is a pointer to the data in the parameter data block. If the parameter data block couldn't be located, a null pointer is returned. The programmer needs to cast the return value to the desired variable type.

Note that the parameter data block pointer returned can become invalid if any changes are made to the parameter data in the phase in which the parameter data block is located. (See Sec. 2.11.4.9 for more discussion of this point.)

14.3.16 getblock_data_copy

This routine returns a pointer to a `COPY` of the data in the parameter data block given the block's name. The pointer to the memory must be freed when finished.

Calling format:

```
block_ptr = (block_type *)get_param_block_data_copy(phase,block_name);
if(block_ptr != NULL)
{
   ...
   free(block_ptr);
}
```

Function Arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored.
- `char *block_name`: The name of the parameter data block.

Return value:

- The return value of the routine is a pointer to memory containing a copy of the data in the parameter data block. If the parameter data block couldn't be located, a null pointer is returned. The programmer needs to cast the return value to the desired variable type. When done, the pointer to the memory must be freed.

This routine is useful when adding or modifying the parameter data in a phase because it returns a copy of a parameter data block rather than a pointer to the data in the block. Adding, deleting, or replacing blocks will not affect the validity of this pointer.

14.3.17 get_external_param_block_ptr

This routine returns a pointer to the data in a particular parameter data block in a specified phase.

Note that the phase used is the one with the `PROCESSED` data. This routine should only be called from the `alg_vectors` routine. If a parameter data block is needed from a phase's `RAW` data, then `get_external_param_block` should be used. This routine has an additional argument which specifies which queue to use to get the data.

The phase is specified by providing the identifier of the category in which the phase is located, the identifier of the algorithm used in the phase and/or the phase name. Wildcard characters can be used. If an identifier or the phase name is specified simply as an asterisk (multicharacter wildcard) then that string does not contribute to the specification of the phase (i.e. the wildcard character will match any identifier or phase name).

Calling format:

```
void * get_external_param_block_ptr(char *block_name,
                                    char *cat_ident,
                                    char *alg_ident,
                                    char *phase_name,
                                    int instance,
                                    char **block_name_out,
                                    void **phase_out)
```

Function Arguments:

- `char *block_name`: The name of the parameter data block. The string can contain one or more asterisks as a multicharacter wildcard and one or more question marks as a single character wildcard.
- `char *cat_ident`: The identifier of the category in which the parameter data block is located. The string can contain one or more asterisks as a multicharacter wildcard and one or more question marks as a single character wildcard. Note that this is the category identifier, not the category name which can be a different string.
- `char *alg_ident`: The identifier of the algorithm used by the phase in which the parameter data block is located. The string can contain one or more asterisks as a multicharacter wildcard and one or more question marks as a single character wildcard. Note that this is the algorithm identifier, not the algorithm name which can be a different string.
- `char *phase_name`: The name of the phase in which the parameter data block is located. The string can contain one or more asterisks as a multicharacter wildcard and one or more question marks as a single character wildcard.
- `int instance`: Normally this argument is 0. However, if a wildcard is used anywhere when specifying the location of the parameter data block, it is possible that there could be more than one match. In this case, the first `instance` matches are skipped and the pointer returned is to the parameter data block that is the next match.
- `block_name_out`: This argument specifies a location to store the pointer to the name of the parameter data block that was located. If the value of this argument is a null pointer, it is ignored.

- `phase_out`: This argument specifies a location to store the pointer to the descriptor for the phase containing the parameter data block that was located. If the value of this argument is a null pointer, it is ignored.

Return values:

- The return value of the routine is a pointer to the data in the parameter data block. If the parameter data block couldn't be located, a null pointer is returned.
- `block_name_out`: If a pointer to a location to store a character string pointer is provided as input by this argument (rather than a null pointer), then a pointer to the name of the parameter data block located is returned. This feature is useful if wildcard characters are used to specify the parameter data block location. The returned value can be used to determine exactly which parameter data block was located in case the wildcard characters cause a match to multiple parameter data blocks.
- `phase_out`: If a pointer to a location to store a pointer to a shot phase descriptor is provided as input by this argument (rather than a null pointer), then a pointer to the descriptor of the shot phase containing the parameter data block that was located is returned. This feature is useful if wildcard characters are used to specify the parameter data block location. The returned value can be used to determine exactly which parameter data block was located in case the wildcard characters cause a match to multiple parameter data blocks.

Note that the parameter data block pointer returned can become invalid if any changes are made to the parameter data in the phase in which the parameter data block is located. This is also true of the pointer returned in `block_name_out`. (See Sec. 2.11.4.9 for more discussion of this point.) So, code should make use of these pointers by accessing or copying the data before any calls to routines that would modify the parameter data in that phase. In practice, this is unlikely to be a problem because `get_external_param_block_ptr` will generally be used by the `alg_vectors` routine to access data in a phase other than the phase for which the `alg_vectors` is calculating processed data.

The `instance` argument might be used in a loop as in the following example.

```
int instance;
void *pointer;

instance = 0;
/*
   Process all parameter data blocks that match the wildcarded
   name.
*/
while(1){
   pointer = get_external_param_block_ptr("my_param_block",
                                          "my_category",
                                          "*",
                                          "name*",
                                          instance,
                                          NULL,NULL);
   if(pointer != NULL){
      /* Process the parameter data block.*/
      instance = instance + 1;
   }
   else
      break;
}
```

14.3.18 get_external_param_block

This routine returns a pointer to the data in a particular parameter data block in a specified phase. The author specifies which queue the data should be retrieved from: `DOIO` for raw data or `DOQUEUE` for processed data. This routine needs to be used in the `alg_fromarchive` or `alg_parameters` routines since these routines are called for both copies of the data.

The phase is specified by providing the identifier of the category in which the phase is located, the identifier of the algorithm used in the phase and/or the phase name. Wildcard characters can be used. If an identifier or the phase name is specified simply as an asterisk (multicharacter wildcard) then that string does not contribute to the specification of the phase (i.e. the wildcard character will match any identifier or phase name).

Calling format:

```
void * get_external_param_block(char *block_name,
                                char *cat_ident,
                                char *alg_ident,
                                char *phase_name,
                                int instance,
                                int which_queue,
                                char **block_name_out,
                                void **phase_out)
```

Function Arguments:

- `char *block_name`: The name of the parameter data block. The string can contain one or more asterisks as a multicharacter wildcard and one or more question marks as a single character wildcard.

- `char *cat_ident`: The identifier of the category in which the parameter data block is located. The string can contain one or more asterisks as a multicharacter wildcard and one or more question marks as a single character wildcard. Note that this is the category identifier, not the category name which can be a different string.

- `char *alg_ident`: The identifier of the algorithm used by the phase in which the parameter data block is located. The string can contain one or more asterisks as a multicharacter wildcard and one or more question marks as a single character wildcard. Note that this is the algorithm identifier, not the algorithm name which can be a different string.

- `char *phase_name`: The name of the phase in which the parameter data block is located. The string can contain one or more asterisks as a multicharacter wildcard and one or more question marks as a single character wildcard.

- `int instance`: Normally this argument is 0. However, if a wildcard is used anywhere when specifying the location of the parameter data block, it is possible that there could be more than one match. In this case, the first `instance` matches are skipped and the pointer returned is to the parameter data block that is the next match.

- `int which_queue`: The queue for which the data should be returned: `DOIO` or `DOQUEUE`.

- `block_name_out`: This argument specifies a location to store the pointer to the name of the parameter data block that was located. If the value of this argument is a null pointer, it is ignored.

- `phase_out`: This argument specifies a location to store the pointer to the descriptor for the phase containing the parameter data block that was located. If the value of this argument is a null pointer, it is ignored.

Return values:

- The return value of the routine is a pointer to the data in the parameter data block. If the parameter data block couldn't be located, a null pointer is returned.

- `block_name_out`: If a pointer to a location to store a character string pointer is provided as input by this argument (rather than a null pointer), then a pointer to the name of the parameter data block located is returned. This feature is useful if wildcard characters are used to specify the parameter data block location. The returned value can be used to determine exactly which parameter data block was located in case the wildcard characters cause a match to multiple parameter data blocks.

- `phase_out`: If a pointer to a location to store a pointer to a shot phase descriptor is provided as input by this argument (rather than a null pointer), then a pointer to the descriptor of the shot phase containing the parameter data block that was located is returned. This feature is useful if wildcard characters are used to specify the parameter data block location. The returned value can be used to determine exactly which parameter data block was located in case the wildcard characters cause a match to multiple parameter data blocks.

Note that the parameter data block pointer returned can become invalid if any changes are made to the parameter data in the phase in which the parameter data block is located. This is also true of the pointer returned in `block_name_out`. (See Sec. 2.11.4.9 for more discussion of this point.) So, code should make use of these pointers by accessing or copying the data before any calls to routines that would modify the parameter data in that phase. In practice, this is unlikely to be a problem because `get_external_param_block` will generally be used by the `alg_fromarchive` or `alg_parameters` routines to access data in a phase other than the phase for which the routine is called.

14.3.19 replaceblock

This routine replaces a given parameter data block. The block `MUST` exist. Most of the attributes of the block, the block order index, the memory alignment, and the flags, remain unchanged. The array of descriptors, however, can change and an argument is provided for such a case; if no changes are needed to the descriptors, this argument is set to NULL.

Calling format:

```
replace_param_block(phase,block_name,descriptors,data_ptr);
```

Function Arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored.
- `char *block_name`: The name of the parameter data block.
- `STRUCT_DESCRIPTORS *descriptors`: A pointer to an array of block descriptor structures describing the block (See Sec. 2.11.4.5), can be NULL. If NULL, then the existing descriptors are unchanged, in which case, the data length remains the same.
- `char *data_ptr`: The pointer to the char data.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`. (See Sec. 2.11.4.9 for more discussion of this point.)

14.3.20 replaceblock_chars

This routine replaces a given parameter data block of type `char`. The block `MUST` exist.

Calling format:

```
replace_param_block_chars(phase,block_name,num_chars,carray);
```

Function Arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored.
- `char *block_name`: The name of the parameter data block.
- `int num_chars`: The number of chars in the data.
- `char *carray`: The pointer to the char data array.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

14.3.21 replace_param_block_shorts

This routine replaces a given parameter data block of type `short`. The block `MUST` exist.

Calling format:

```
replace_param_block_shorts(phase,block_name,num_shorts,sarray);
```

Function Arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored.
- `char *block_name`: The name of the parameter data block.
- `int num_shorts`: The number of shorts in the data.
- `short *sarray`: The pointer to the short data array.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

14.3.22 replace_param_block_ints

This routine replaces a given parameter data block of type `int`. The block `MUST` exist.

Calling format:

```
replace_param_block_ints(phase,block_name,num_ints,iarray);
```

Function Arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored.
- `char *block_name`: The name of the parameter data block.
- `int num_ints`: The number of ints in the data array.
- `int *iarray`: The pointer to the int data array.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

14.3.23 replace_param_block_longs

This routine replaces a given parameter data block of type `long`. The block `MUST` exist.

Calling format:

```
replace_param_block_longs(phase,block_name,num_longs,larray);
```

Function Arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored.
- `char *block_name`: The name of the parameter data block.
- `int num_longs`: The number of longs in the data array.
- `long *larray`: The pointer to the long data array.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

14.3.24 replace_param_block_floats

This routine replaces a given parameter data block of type `float`. The block `MUST` exist.

Calling format:

```
replace_param_block_floats(phase,block_name,num_floats,farray);
```

Function Arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored.
- `char *block_name`: The name of the parameter data block.
- `int num_floats`: The number of floats in the data array.
- `char *farray`: The pointer to the float data array.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

14.3.25 replace_param_block_doubles

This routine replaces a given parameter data block of type `double`. The block `MUST` exist.

Calling format:

```
replace_param_block_doubles(phase,block_name,num_doubles,darray);
```

Function Arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored.
- `char *block_name`: The name of the parameter data block.
- `int num_doubles`: The number of doubles in the data array.
- `char *darray`: The pointer to the double data array.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

14.3.26 replace_param_block_strings

This routine replaces a given parameter data block which is made up of an array of strings. The block `MUST` exist.

Calling format:

```
replace_param_block_strings(phase,block_name,number,char *strings[]);
```

Function Arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored.
- `char *block_name`: The name of the parameter data block.
- `int number`: The number of strings in the data array.
- `char *strings[]`: The pointer to the strings array.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

14.3.27 copyblock

This routine copies an existing parameter data block to a new parameter data block. The new block may or may not exist. It is the equivalent of getting the block given by the first name and all its attributes and creating the block with the second name using those attributes.

Calling format:

```
copyblock(phase,name,new_name);
```

Function Arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored.
- `char *name`: The name of the parameter data block to copy.
- `int new_name`: The name of the block to create.

Return value:
The function returns 1 if the block does not exist, otherwise it returns 0.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

14.3.28 deleteblock

This routine deletes a parameter data block if it exists.

Calling format:

```
deleteblock(phase,name);
```

Function Arguments:

- `phase`: The pointer to the descriptor of the phase for which the parameter data is stored.
- `int name`: The name of the block to delete.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

14.3.29 pcs_forhost

This routine is the standard routine which gathers the parameter data blocks needed by a given virtual cpu for the algorithm. The header information associated with each block determines whether the block should go on the host cpu and/or the real-time cpu. An algorithm can have its own specific forhost routine though no such routine has yet been needed. Note that the `return_data_type` argument indicates which process is requesting the data, the host_cpu process or the realtime_cpu process.

Calling format:

```
void pcs_forhost(
     struct shotphase *phase,
     int vcpu,
     char **data,
     int *length,
     int return_data_type,
     struct forhost_struct *forhost_info)
```

Function Arguments:

- `struct shotphase *phase`: pointer to the descriptor of the shot phase.
- `int vcpu`: the virtual cpu number whose data is requested.
- `char **data`: the address to receive the pointer to the memory with the parameter data.
- `int *length`: the address of an int to receive the total length.
- `int return_data_type`: input value of `FORHOSTPARAM` to specify the data is for the `host_cpu` process or `FORHOSTRTCPU` to specify data for the `realtime_cpu` process.
- `struct forhost_struct *forhost_info`: the pointer to the special structure used by the `forhost` routine which has in it the following fields:
  - `int highest_block_index` set to the highest value of the block order index for the phase. This is used for the `FORHOSTRTCPU` case in order to set aside space at the beginning of the parameter data for that many pointers, each of which points to its corresponding block.
  - `int client_mask` set to the client processor type input to the `return_parameter_data` routine.

14.3.30 pcs_foruser

This routine is the standard routine which gathers the parameter data blocks needed by the user interface. The message from the user interface may contain the name(s) of the block(s) to return. If no names are specified, then all blocks in the phase are returned. Names are not case sensitive but no wildcarding is allowed. An algorithm can have its own specific foruser routine though no such routine has yet been needed.

Calling format:

```
void pcs_foruser(
     struct shotphase *phase,
     char **data,
     int *length,
     struct foruser_struct *foruser_info)
```

Function Arguments:

- `struct shotphase *phase`: pointer to the descriptor of the shot phase.
- `char **data`: the address to receive the pointer to the memory with the parameter data.
- `int *length`: the address of an int to receive the total length.
- `struct foruser_struct *foruser_info`: the pointer to the special structure used by the `foruser` routine which has in it the following fields:
  - `int client_mask` set to the client processor type input to the `return_parameter_data` routine.
  - `int forwho` a value of `FORUSER` to get parameter data blocks that have the `FORUSER_MASK` mask bit set in the block's flags or `FORALL` to get all blocks in the phase (the FORUSER_MASK bit is ignored).
  - `int string_count` the number of strings.
  - `char **strings` a list of null-terminated strings giving the names of the blocks to return. If the string_count is 0, then all blocks in the phase matching the `forwho` request are returned.

14.3.31 pcs_fromuser

This routine is the standard routine which takes the parameter data blocks sent from the user interface and inserts them into the parameter data for the phase. An algorithm can have its own specific fromuser routine though no such routine has yet been needed.

Calling format:

```
void pcs_fromuser(
     struct shotphase *phase,
     char *data,
     int length,
     struct fromuser_struct *fromuser_info)
```

Function Arguments:

- `struct shotphase *phase`: pointer to the descriptor of the shot phase.
- `char *data`: the pointer to the memory with the parameter data to insert into the phase. This memory has a different format for each block than the format used in the wavefrom server.

- `int length`: the length of the input data.
- `struct fromuser_struct *fromuser_info`: the pointer to the special structure used by the `fromuser` routine which has in it the following fields:
  - `int client_mask` indicates the client processor type.
  - `int which_queue` a value specifying which copy of the data to change: `raw` or `processed`.

14.3.32 pcs_forarchive

This routine is the standard routine which takes the parameter data blocks from a phase and prepares them to be archived. The archive requires that each header item and each type of data be separated into the different types of variables: char, short, int, float, and double. For example, the header items for a block are all ints except the name which is char so all header values go into the int array and all the names go into the ascii array. The block's data goes into its respective variable type array. An algorithm can have its own specific forarchive routine though no such routine has yet been needed.

Calling format:

```
void pcs_forarchive(
     struct shotphase *phase,
     char **asciidata,
     int *asciicount,
     short **shortdata,
     int *shortcount,
     int **intdata,
     int *intcount,
     float **floatdata,
     int *floatcount,
     double **doubledata,
     int *doublecount,
     struct forarchive_struct *forarchive_info
```

Function Arguments:

- `struct shotphase *phase`: pointer to the descriptor of the shot phase.
- `char **asciidata`: the address to receive the pointer to the memory with the ascii parts of all the parameter data.

- `int *asciicount`: the address of an int to receive the number of ascii values.
- `char **shortdata`: the address to receive the pointer to the memory with the short parts of all the parameter data.
- `int *shortcount`: the address of an int to receive the number of short values.
- `char **intdata`: the address to receive the pointer to the memory with the int parts of all the parameter data.
- `int *intcount`: the address of an int to receive the number of int values.
- `char **floatdata`: the address to receive the pointer to the memory with the float parts of all the parameter data.
- `int *floatcount`: the address of an int to receive the number of float values.
- `char **doubledata`: the address to receive the pointer to the memory with the double parts of all the parameter data.
- `int *doublecount`: the address of an int to receive the number of double values.
- `struct forarchive_struct *forarchive_info`: the pointer to the special structure used by the `forarchive` routine which has in it the following fields:
  - `int client_mask` indicates the client processor type.

14.3.33 pcs_fromarchive

This routine is the standard routine which takes the parameter data from the archive and creates the parameter data blocks in a phase. The parameter data from the archive is separated into various pieces based on their data types. For example, the header values of a block, the flags, size, alignment, etc., are all of type int while the block name is of type char. This routine takes values from the different arrays and combines them to create the parameter data blocks.

An algorithm can have its own specific `fromarchive` routine. Such a routine would be needed in cases where a block changes size or fields in a structure get moved around. In practice, if the various `putblock_labeled_*` routines are used to create one or more blocks, then an algorithm specific fromarchive routine is not necessary.

Calling format:

```
void pcs_fromarchive(
     struct shotphase *phase,
     char *asciidata,
     int asciicount,
     short *shortdata,
     int shortcount,
     int *intdata,
     int intcount,
     float *floatdata,
     int floatcount,
     double *doubledata,
     int doublecount,
     struct fromarchive_struct *fromarchive_info
```

Function Arguments:

- `struct shotphase *phase`: pointer to the descriptor of the shot phase.
- `char *asciidata`: the pointer to the memory with the ascii parts of all the archived parameter data.
- `int asciicount`: the number of ascii values.
- `char *shortdata`: the pointer to the memory with the short parts of all the archived parameter data.
- `int shortcount`: the number of short values.
- `char *intdata`: the pointer to the memory with the int parts of all the archived parameter data.
- `int intcount`: the number of ascii values.
- `char *floatdata`: the pointer to the memory with the float parts of all the archived parameter data.
- `int floatcount`: the number of ascii values.
- `char *doubledata`: the pointer to the memory with the double parts of all the archived parameter data.
- `int doublecount`: the number of ascii values.
- `struct fromarchive_struct *fromarchive_info`: the pointer to the special structure used by the `fromarchive` routine which has in it the following fields:
  - `int client_mask` indicates the client processor type.

- `int which_queue` a value specifying which copy of the data to change: `raw` or `processed`.
- `int group_name` a string which is either "RESTORE ALL BLOCKS" to restore all blocks in the phase or the name of a data item that is being restored separately. Any archived block whose name matches the group name up to the length of the data item name will be restored. For example, if the item name is "M matrix" then any block that starts with "M matrix" will be restored.
- `int new_name` if not NULL, then any blocks whose name matches `group_name` will have those characters replaced with those in `new_name`. For example, if group_name is "M matrix" and the new_name is "M matrices", then all blocks that start with "M matrix" will be renamed with the new prefix of "M matrices".
- `int legacy_names_count` the count of legacy names.
- `char **legacy_names_old` a string array of old names whose blocks need to be renamed.
- `char **legacy_names_new` a string array of the new names. Group names are considered prefixes, but legacy names must be given in full, though they are compared without regard to case.
- `int legacy_groups_count` the count of legacy groups.
- `char **legacy_groups` a string array of legacy group names which allow the restore of blocks whose names match these groups to be restored with "_a" appended to the names so the current blocks are unaffected. Then the `fromarchive` routine can transfer data as needed to the current blocks. This was an old way of allowing blocks to change size.

Example of a `fromarchive` routine. This routine is required because the structure of the data was appended to it twice. Since there is no parameter data block with a label for each item in the structure, the matching up of old values to the new structure can not be done automatically like it is for blocks created with the `putblock_labeled_*` routines.

```
/*
*****************************************************************************
SUBROUTINE: sysmain_fromarchive

This special fromarchive routine exists because:
1. icpatch was increased from 4 c-supplies to 5, requiring
   restores of old shots to have the 5th c-supply added in
*****************************************************************************
*/
void sysmain_fromarchive(
    struct shotphase *phase,
    char *asciidata,    int asciicount,
    short *shortdata,   int shortcount,
    int *intdata,       int intcount,
    float *floatdata,   int floatcount,
    double *doubledata, int doublecount,
    struct fromarchive_struct *fromarchive_info)
{
char *data1, *data2;
int i,j,n,datasize1,datasize2,AAs_used;
static GEN_ic_line
struct icpatch_data *icpatch;
GEN_icpatch_data

   pcs_fromarchive(
                   phase,
                   asciidata, asciicount,
                   shortdata, shortcount,
                   intdata,  intcount,
                   floatdata,  floatcount,
                   doubledata, doublecount,
                   fromarchive_info);
/*
For the data item "icpatch data", the data structure was expanded
from 8 ic_lines to 10.  So if the size of the block is smaller
than the size of the default block, then transfer the data
from the smaller one to the bigger one.
*/
   if((strcasecmp(fromarchive_info->group_name,"RESTORE ALL BLOCKS") == 0) ||
      (strcasecmp(fromarchive_info->group_name,"icpatch data") == 0))
   {
/*
Check the size to see how big the restored data is.
If it is smaller, we must transfer the restored data to
the default data and store it.
*/
       data1 = (char *)get_param_block(phase->parameters,
                       "icpatch data default",
                       0,0,&datasize1,0,0,0,0,0);

       data2 = (char *)get_param_block(phase->parameters,
                      "icpatch data",
                      0,0,&datasize2,0,0,0,0,0);
/*
Oldest shots have C1-C4:     416 bytes
Then C5 was added:           512 bytes
Then 12 AA names were added: 608 bytes
*/
       n = 10;
       if(datasize2 == 416) n = 8;
       for(j=0;j<n;j++)
       {
           for(i=0;i<sizeof(icpatch->csupply_data[j].csupply_polarity);i++)
           {
               if(icpatch->csupply_data[j].csupply_polarity[i] == 0)
                  icpatch->csupply_data[j].csupply_polarity[i] = 32;
           }
           for(i=0;i<24;i++)
           {
               if(icpatch->csupply_data[j].coil_names[i] == 32)
                  icpatch->csupply_data[j].coil_names[i] = 0;
           }
       }
/*
Since the structure was appended to,
we can simply copy the older sized structure into the newer sized
structure.  Use a copy of the default data so the added bytes
will be the same as the default values.
*/
       if(datasize2 < datasize1)
       {
          data1 = get_param_block_data_copy(phase->parameters,
                  "icpatch data default");
          memcpy(data1,data2,datasize2);
          replace_param_block(&(phase->parameters),
              "icpatch data", D_icpatch_data, data1);
          free(data1);
       }
/*
Check if the AA's are being used.
If the first name is not blank, then AA's are used.
*/
```

```
      icpatch = (struct icpatch_data *)get_param_block_data_ptr(
          phase->parameters,"icpatch data");
      if(strncmp(icpatch->AA_coil_name[0],"   ",3) == 0) AAs_used = 0;
      else AAs_used = 1;
      replace_param_block_ints(&(phase->parameters),"audioamps used",1,
          &AAs_used);
   }
   return;
}
```

14.3.34 pcs_paramsize

This routine is the standard routine which calculates the size of the parameter data that go to the `realtime_cpu` process. An algorithm can have its own specific paramsize routine though no such routine has yet been needed.

Calling format:

```
int pcs_paramsize(struct shotphase *phase, int vcpu)
```

Function Arguments:

- `struct shotphase *phase`: pointer to the descriptor of the shot phase.
- `int vcpu`: the virtual cpu number whose data is requested.

Return values:

- The return value of the routine is the total size of the parameter data used in the phase on the real-time cpu given by vcpu.

14.3.35 get_param_numblocks

Routine that returns the number of parameter blocks given a pointer to a parameter data region.

Calling format:

```
int get_param_numblocks(char *mem_area)
```

Function Arguments:

- `char *mem_area`: a pointer to a parameter data region.

Return values:

- The return value of the routine is the number of parameter data blocks found in the parameter data region.

14.3.36 get_cpu_from_name

Routine that returns the cpu number from the last character in the given character string. For example, if the input string is "CPUC", then this function returns 3. Note that "Z" returns 26 and "a" returns 27.

Calling format:

```
int get_cpu_from_name(char *name)
```

Function Arguments:

- `char *name`: the input string which has a cpu letter at the end.

Return values:

- The return value is the integer corresponding to the last letter in the input string where "A" is 1, "B" is 2, etc.

14.3.37 get_rt_flag_from_cpu

Routine that returns the realtime flag mask value from the cpu number.

Calling format:

```
int get_rt_flag_from_cpu(int vcpu)
```

Function Arguments:

- `int vcpu`: the virtual cpu number whose data is requested.

Return values:

- The return value of the routine is the parameter data flag mask for the given virtual cpu number.

14.3.38 get_host_flag_from_cpu

Routine that returns the host realtime flag mask value from the cpu number.
Calling format:

```
int get_host_flag_from_cpu(int vcpu)
```

Function Arguments:

- `int vcpu`: the virtual cpu number whose data is requested.

Return values:

- The return value of the routine is the parameter data flag mask for the given host virtual cpu number.

14.3.39 pcs_delete_archive_blocks

Deletes parameter data blocks whose names end in "_a". If a name is given, then it is a "group" name so this routine will match blocks beginning with the name. If name is NULL, then delete all archive parameter data blocks.
Calling format:

```
void pcs_delete_archive_blocks(struct shotphase *phase, char *name)
```

Function Arguments:

- `struct shotphase *phase`: pointer to the descriptor of the shot phase.
- `char *name`: the name of a "group" whose blocks should be deleted. If name is NULL, then all archive parameter data blocks are deleted.

Return values:

- The return value of the routine is the parameter data flag mask for the given host virtual cpu number.

14.4 Registration of messages used in real time

14.4.1 rtmessage_register

Register a real time message. This function is called in the waveform server code to store information about the message that will be sent from one real time processor to another. For additional background information on the the real time message facility see Sec. 2.9.4.

Calling format:

```
int rtmessage_register(
                  struct shotphase *phase,
                  char *name,
                  char *options,
                  STRUCT_DESCRIPTORS *descriptors
                 )
```

Function Arguments:

- `phase`: Pointer to the phase structure.
- `name`: The name to use for the message.
- `options`: A character string describing the options allowed using keywords. Keywords may or may not have a value. Separate keywords with a vertical bar. The first group of keywords below have no value and specify the bits in a bitmask that gets stored with the other message information. The second group of keywords below specify quantities like the source CPU number and the destination CPU number.
  - `NOSENDBUFFER`: do not allocate space for a send buffer. By default, space is set aside on the source CPU in case it is needed to put the message together. Use this option to save on real time memory if the buffer is already allocated elsewhere. Note that this disables the built-in synchronization feature since counters cannot be added to the buffer that is used.
  - `RETURNLATEST`: always return the latest message even if that message was returned earlier. Note that if this option is NOT given, then `rtmessage_read` returns a NULL pointer if no new message has arrived.
  - `SWINGBUFFER`: allocate two destination buffers. The source CPU will alternate between writing to the two buffers. On the destination processor, if no message has arrived, then `rtmessage_read` returns a NULL pointer. Otherwise, `rtmessage_read` returns a pointer to the LAST message that has arrived, even if that message was returned earlier. This option is equivalent to `RETURNLATEST|BUFFERS=2`.

- `NOCOUNTERS`: do not allocate the counters used to synchronize sending and receiving. By default, an 8-byte space is allocated both at the beginning and at the end of the destination buffer so that a counter and source cpu number can be added at the beginning and the same counter added at the end of the message. These counters are used to verify the message has been completely written and that the message is not currently being written. This option can be used to save memory in the case where messages are sent between processors in a multi-processor computer using shared memory where no synchronization may be needed.
- `SHAREBUFFERS`: overlay the send and receive buffers. Use this option for cases where shared memory on a multi-processor computer is to be used. If no synchronization is needed, then also use the `NOCOUNTERS` keyword. In this case, just fill the send buffer and "the message" has arrived. There is no need to use the `rtmessage_write` or `rtmessage_read` functions. However, synchronization requires the use of these two functions and probably multiple receive buffers to be sure that the receiver does not use data while the data are changing.

- `SOURCE=cpu`: The source cpu. This can be a number of formats.
  * A single virtual CPU letter, e.g., "SOURCE=A", specifies the virtual CPU for the algorithm's category.
  * A single CPU number, e.g., "SOURCE=1", specifies physical CPU 1.
  * A single CPU number with the "CPU" prefix, e.g., "SOURCE=CPU1" specifies physical CPU 1, "SOURCE=CPUA" specifies virtual CPU A.
  * A string specifying a physical CPU tied to a category, e.g., "SOURCE=PCPU_SHAPEA", specifies the physical CPU that equates to virtual CPU A of the SHAPE category.

  The best formats are the ones that do not hardcode a number.
- `DESTINATION=cpu`: The destination CPU or CPUs. More than one CPU may be given to "broadcast" the message. Specify multiple values separated by commas. This can be a number of formats.
  * A single virtual CPU letter, e.g., "DESTINATION=A", specifies the virtual CPU for the algorithm's category.
  * A single CPU number, e.g., "DESTINATION=1", specifies physical CPU 1.
  * A single CPU number with the "CPU" prefix, e.g., "DESTINATION=CPU1" specifies physical CPU 1, "DESTINATION=CPUA" specifies virtual CPU A.
  * A string specifying a physical CPU tied to a category, e.g., "DESTINATION=PCPU_SHAPEA", specifies the physical CPU that equates to virtual CPU A of the SHAPE category.
  * The keyword "ALL" specifies all other CPUs except the source CPU.

The best formats are the ones that do not hardcode a number.

- `BUFFERS=n`: allocate multiple destination buffers. The source CPU will always write to the next buffer on the receiver, wrapping around to the first buffer. On the destination processor, other options determine what `rtmessage_read` returns. If there are no other options, then `rtmessage_read` returns a NULL pointer if no message has arrived, a pointer to the next message if not already returned, and a NULL pointer if all messages have been returned earlier.
- `TIMEOUT=n`: this specifies the number of microseconds that the destination CPU waits for a new message to arrive. The counters are used for this purpose. When the counter at the end of the message changes, `rtmessage_read` returns with the pointer to the buffer just written. If no message arrives within the timeout period, then `rtheap->return_status` is set to `TIMEOUT_RETURN`.
- `REGION=n`: Number of the memory region to use for the send and receive buffers. The memory region needs to be created beforehand. See Sec. 13.2 for a discussion of this feature.
- `SENDREGION=n`: Number of the memory region to use for the send buffer. The memory region needs to be created beforehand. See Sec. 13.2 for a discussion of this feature.
- `RECEIVEREGION=n`: Number of the memory region to use for the receive buffer. The memory region needs to be created beforehand. See Sec. 13.2 for a discussion of this feature.
- `ALIGNMENT=n`: specify an alignment for the message buffers. The default alignment is 8 bytes. If 16 or 32 byte alignment is required, then use this option.

• `descriptors`: A data object descriptor (Sec. A) describing the data that will be sent in real time. This descriptor determines the size of the message. If the message length is a known size (like a bunch of ints which are always 4 bytes each), then the size can be given as a keyword and this argument can be NULL. Creating a data object descriptor for the data is the recommended method since it will always be portable if hardware changes in the future.

Return value:
This function returns 0 if there are no errors, -1 if there is an error.

The default behavior if no options in the first group are given is that `rtmessage_read` returns a NULL pointer if no new message has arrived (or no messages at all), and a pointer to the next message if that message has not been read as yet. In the case of multiple buffers, all messages will be returned one at a time if a backlog occurs. If a delay on the destination processor is longer than the number of buffers times the cycle time, then messages could be lost.

14.4.2 Examples of registering messages

Here are a few examples of registering messages.

Example 1.

```
    rtmessage_register(phase,"alg case 1",
        "SOURCE=B|DESTINATION=A|SIZE=4",NULL);
```

Register a message of size 4 that will be sent from virtual CPUB to virtual CPUA. When CPUA calls `rtmessage_read`, a pointer to the message buffer will be returned if a new message has arrived, else a NULL pointer will be returned. There is no waiting for a new message.

Example 2.

```
{
GEN_int

    D_int[0].size = 2;
    rtmessage_register(phase,"alg case 2",
        "SOURCE=B|DESTINATION=A|TIMEOUT=1250",D_int);
}
```

Register a message that consists of two ints that will be sent from virtual CPUB to virtual CPUA. In this case, `rtmessage_read` on CPUA will wait up to 1250 usec during the shot to get a new message. If this time elapses without a new message, then `rtheap->return_status` will be set to `TIMEOUT_RETURN`. If `rtmessage_read` is used before the shot or after the shot, then the function will wait up to 15 seconds for the message to arrive. Note that in `runsa` there is no waiting during the shot.

Example 3.

```
{
GEN_int

    D_int[0].size = 2;
    rtmessage_register(phase,"alg case 3",
        "SOURCE=B|DESTINATION=A|SWINGBUFFER",D_int);
}
```

Register a message that consists of two ints that will be sent from virtual CPUB to virtual CPUA. Two buffers are allocated on CPUA. If a message has arrived, CPUA will always get a pointer to the last message even if that message was returned earlier. A NULL pointer is returned only if no message has yet arrived. There is no waiting for a new message.

Example 4.

```
{int cpus[] = {CPU1,CPU2,0};
    set_rt_memory_region(2, cpus, MEMORY_SHAREABLE, MEMORY_CONTIGUOUS);
}
{
GEN_int

    D_int[0].size = 2;
    rtmessage_register(phase,"alg case 4",
        "SOURCE=B|DESTINATION=A|BUFFERS=200|REGION=2",D_int);
}
```

Register a message that consists of two ints that will be sent from virtual CPUB to virtual CPUA. In this case, 200 destination buffers are allocated in a shared memory region (region 2) located on physical cpus 1 and 2 which are in the same computer. CPUA might be a real time scope which cycles much slower than CPUB so it needs a lot of buffers so that no data is lost. CPUA will get the pointer to the next buffer each time `rtmessage_read` is called. If there is no new message and all messages have been returned, then `rtmessage_read` returns a NULL pointer.

Example 5.

```
{
GEN_int

    D_int[0].size = 2;
    rtmessage_register(phase,"alg case 5",
        "SOURCE=B|DESTINATION=A|BUFFERS=4|RETURNLATEST",D_int);
}
```

Register a message that consists of two ints that will be sent from virtual CPUB to virtual CPUA. In this case, 4 destination buffers are allocated. `rtmessage_read` will return a NULL pointer only when no messages have arrived. Otherwise, it will always return the latest message even if more than one has arrived since the last read. This case could be used if the destination processor needs to use the content of the buffer for a long time. The multiple buffers would help to ensure that the buffer is not overwritten while it is being used.

Example 6.

```
{
GEN_int

    rtmessage_register(phase,"alg case 6",
        "SOURCE=A|DESTINATION=ALL",D_int);
}
```

Register a message that consists of one int that will be broadcast from virtual CPUA to all other CPUs in the PCS.

14.5 Routines for communicating with the user

14.5.1 raw_data_problem

This routine maintains the raw data problem list.

This routine is called by functions invoked during processing by the queue handler (normally the `alg_vectors` function) to indicate problems encountered when converting raw data into processed data. Primarily these would be problems with the values specified by the user for some of the raw data. The raw data problem list alerts the user to changes that must be made in the raw data.

For instance, if 2 pieces of raw data conflict, then when this is discovered a problem entry is added to the list. When the raw data are changed so that there is no longer a problem, then the problem entry is removed from the list.

For additional background on the raw data problem list see Sec. 2.14.1.

Calling format:

```
raw_data_problem(int action, char *type, struct shotphase *phase,
                 variable list of arguments)
```

Function Arguments:
(note that not all arguments are required):

- `int action`: either `ADD_RAW_DATA_PROBLEM`, `APPEND_RAW_DATA_PROBLEM` or `REMOVE_RAW_DATA_PROBLEM`, required argument.

- `char *type`: a short string used as the identifier of the type of raw data problem. Required argument.

- `struct shotphase *phase`: pointer to the shot phase descriptor of the phase in which the problem is located. Used with `type` to uniquely identify the type and location of the problem. `phase->category` and `phase->number` may be 0 to indicate that the problem is not associated with any specific phase but can be considered of system-wide concern. Required argument.

- `int severity`: either `RAW_PROBLEM_WARNING` or `RAW_PROBLEM_FATAL`. Indicates whether the problem will cause the shot to fail. Optional argument, required if action is `ADD_RAW_DATA_PROBLEM` or `APPEND_RAW_DATA_PROBLEM`.

- `char *description`: a string that is used to provide the basic description of the problem to the user. This string is used together with the instance string to provide the complete description and location of the problem. If the string is zero length or a null pointer then it is ignored. Optional argument, required if action is `ADD_RAW_DATA_PROBLEM` or `APPEND_RAW_DATA_PROBLEM`.

- `char *format`: first portion of the instance description. A format string for the `sprintf` function that will be used to generate the text string describing the particular instance of the problem for the user. The complete instance string cannot contain more than `RAW_PROBLEM_TEXT_LENGTH_LIMIT` (defined in `serverdefs.h`, normally 1024) characters. If the string is zero length or a null pointer then it is ignored. Optional argument, required if the action is `ADD_RAW_DATA_PROBLEM` or `APPEND_RAW_DATA_PROBLEM`.

- Other arguments: a variable length list providing other arguments as required by the format string. Optional arguments only necessary if required by the string description argument `format`.

Return values:

- If the action is `ADD_RAW_DATA_PROBLEM` or `APPEND_RAW_DATA_PROBLEM`, the number of characters in the instance text string including the terminating null character is returned. This value can be compared to the maximum allowable length of the string (`RAW_PROBLEM_TEXT_LENGTH_LIMIT`) to detect problems when the string is too long. (If the string is too long then an error message is sent to the log file. It is also possible that the waveform server will crash.)

- If the action is `ADD_RAW_DATA_PROBLEM` and the problem is already on the list, -1 is returned to indicate nothing was done. This is a normal return.

- If the action is `REMOVE_RAW_DATA_PROBLEM` and the problem is not on the list, -1 is returned to indicate nothing was done. This is a normal return.

- If the action is `REMOVE_RAW_DATA_PROBLEM` and the problem is on the list, 0 is returned.

- If the action is `APPEND_RAW_DATA_PROBLEM`, the number of characters in the instance is always returned as described above.

Actions:

- `REMOVE_RAW_DATA_PROBLEM`: the problem identified by `type` and `phase` is removed from the list if it is there.

- `ADD_RAW_DATA_PROBLEM`: the problem identified by `type` is added to the list if it is not already there. If the problem is already on the list, nothing is done. The text string used to describe the problem is the description string followed by the instance string.

- `APPEND_RAW_DATA_PROBLEM`: the problem identified by `type` is added to the list if it is not already there. If the problem is already on the list, the description string is ignored and the instance string is appended to the string that is already on the problem list. This can be used to collect information about multiple instances of the same problem.

14.5.2 msg_log

This routine can be used to send a message to the message server and log it to the process' log file. It uses `fprintf`.

Calling format:

```
 int msg_log(int msg_code, char *strm_fmt, ...)
```

Function Arguments:

- `int msg_code`: macro value indicating the source of the message. In most cases, this value should be `PCSMSG` (other cases are reserved for infrastructure routines).
- `char *strm_fmt`: the format passed on to fprintf.
- `...`: zero or more variables which match the format.

Example:

```
msg_log(PCSMSG,"error reading simserver data file %s\n",filename);
```

14.5.3 logit

This routine writes a time stamped message to stderr. The time stamp is written out before the message. This function uses vfprintf. This is a function that should only be used in the waveform server code or the host real time code.

Calling format:

```
 int logit(char *strm_fmt, ...)
```

Function Arguments:

- `char *strm_fmt`: the format passed on to vfprintf.
- `...`: zero or more variables which match the format.

Example:

```
logit("error reading data file %s\n",filename);
```

14.6 Other utility routines

14.6.1 set_rt_memory_region

To create a new memory region, the waveform server needs to be told that a new memory region is desired. This is usually done in the `alg_parameters` function or in the `alg_vectors` function of an algorithm. A region number must be specified. So it is best to set macros in the installation file `config.h` that would specify the region numbers (in case more than one is desired).

To specify that a memory region is to be created on a particular real time processor, call the following function:

Calling format:

```
void set_rt_memory_region(int region_number,
                          int *cpus_list,
                          int type,
                          int flag);
```

Function Arguments:

- region_number: int value starting at 2 (1 is the real time heap)
- cpus: int array of cpu numbers ending with a zero or a -1 to indicate all cpus followed by a zero; e.g., {CPU1,CPU2,0} or {-1,0}
- type: int indicating the type of memory region:
  - MEMORY_GENERIC uses malloc to create a memory region
  - MEMORY_SHAREABLE creates a shareable memory region
  - MEMORY_INSTALL_SPECIFIC creates a memory region that is installation specific
- flag: bit mask indicating options:
  - MEMORY_CONTIGUOUS all memory sections for real time processors will follow each other in the region, i.e., processor2 follows processor1

– MEMORY_NO_SFILE do not include content of memory in S file (just in case it is not accessible by the host real time)
– MEMORY_NO_DELETE do not delete the memory region at the end of the shot (the default is to delete it)

This function has no return value.

Example 1:

```
{
int cpus[] = {-1,0};
   set_rt_memory_region(2, cpus, MEMORY_SHAREABLE, MEMORY_CONTIGUOUS);
}
```

One shareable memory region is created on the real time computer. The region is continguous which means that the same region is used for all real time processors. Any parameter data blocks or other data redirected to this region will be allocated so that one CPU's data follows another CPU's data. This example could be used for cases where one multi-processor computer is used for all real time processes. Or it could be used when multiple computers use reflective memory to communicate.

Example 2:

```
{
int cpus[] = {PCPU_EQUILA,PCPU_EQUILB,0};
    set_rt_memory_region(2, cpus, MEMORY_SHAREABLE, MEMORY_CONTIGUOUS);
}
```

Here the sharable memory region is created for the two rtefit virtual CPUs which will be located on processors in the same computer. The region is continguous which means that the same region is used for both real time processors. Any parameter data blocks or other data redirected to this region will be allocated so that one CPU's data follows the other CPU's data.

14.6.2 set_rt_memory_preference

As discussed in Sec. 13.2, new memory regions can be created. Different parts of the `rtheap` memory can then be redirected to one of these regions. The function `set_rt_memory_preference` is used to move a section to a new memory region. It can be put into the parameters function of the system category's algorithm.

Calling format:

```
void set_rt_memory_preference(int section, int region, int *cpus, int flags)
```

Function arguments:

- `section`: one of the `_PREFERENCE` macros found in the infrastructure file `serverdefs.h`. One example is `INSTALL_BUFFER_PREFERENCE`.
- `region`: the memory region number (this must be greater than 1).
- cpus: int array of physical cpu numbers ending with a zero, or a -1 to indicate all cpus followed by a zero; e.g., {CPU1,CPU2,0} or {-1,0}
- `flags`: a bitmask specifying different options.
  - `MEMORY_OVERLAY`: overlay the section on multiple CPUs, i.e., share the section. For example, the dmabuffer could be overlayed so it is shared by processors on the same computer.

This function has no return value.

Example:
Overlay the dmabuffer so all real time CPUs point to the same address.

```
{
int cpus[] = {-1,0};
    set_rt_memory_region(2, cpus, MEMORY_SHAREABLE, MEMORY_CONTIGUOUS);
    set_rt_memory_preference(DMABUFFER_PREFERENCE, 2, cpus, MEMORY_OVERLAY);
}
```

14.6.3 set_offload_data_flag

If an installation needs to archive data during a long discharge but there is not enough memory, then the installation can off-load the data during the shot. This function needs to be called for any cpu that will off-load the data. The real time vectors are then allocated such that one follows the next so that one buffer is created for each time slice. This buffer can then be sent to another computer outside of the PCS so the data can be archived. All vectors are included except the target and pointer target.

Calling format:

```
void set_offload_data_flag(int *cpus,int value)
```

Function arguments:

- cpus: int array of physical cpu numbers ending with a zero, or a -1 to indicate all cpus followed by a zero; e.g., {CPU1,CPU2,0} or {-1,0}
- `value`: a non-zero value turns this feature on, a value of zero turns this feature off.

This function has no return value.

14.6.4 set_cache_preference

As discussed in Sec. 2.9.2, the different parts of the `rtheap` memory are put in one of two sections: `cacheable` or `noncacheable`. The waveform server puts all sections into `cacheable` memory except the following:

- `adcombuffer`: the section of the communications buffer which is written to by other real time processors.
- `dmabuffer`: the section of memory that holds raw data which could be directly written by digitizers.
- `timetostop_flag`: the address of the flag that indicates that the discharge is over. This address could be written to by another real time processor.

The function `set_cache_preference` can be used to move other sections to `noncacheable` memory. It can be put into the parameters function of the system category's algorithm.
Calling format:

```
void set_cache_preference(int section, int preference)
```

Function arguments:

- `section`: one of the `_PREFERENCE` macros found in the infrastructure file `serverdefs.h`. One example is `INSTALL_BUFFER_PREFERENCE`.
- `preference`: one of the two preferences: `CACHEABLE` or `NONCACHEABLE`.

This function has no return value.

Example:

```
  set_cache_preference(INSTALL_BUFFER_PREFERENCE, NONCACHEABLE);
```

14.6.5 set_vertices

This function changes the vertices for a specified waveform. The arguments specify the waveform and the new set of vertices. This new set of vertices completely replaces the previous set of vertices for the waveform.

This function does not change the vertices of the waveform if the access control mask for the waveform has the `AC_NORESTORE` bit set and the current stage of the waveform initialization procedure requires that this bit be checked (as done by the function `check_access_control`, Sec. 14.6.39, with the check type `AC_INITIALIZE`).

Calling format:

```
set_vertices(char *waveform, struct shotphase *phase, struct vertex newv[],
int numvertex)
```

The arguments are:

- `waveform`: string giving the name of the waveform. The waveform must be used by the algorithm assigned to the shot phase specified.
- `phase`: pointer to the descriptor for the shot phase for which the waveform vertices are being changed.
- `newv`: an array of structures of type `vertex` that has `numvertex` elements. Each of the structures specifies one of the new vertices.
- `numvertex`: the count of the new vertices.

14.6.6 set_phsseq_vertices

This function changes the vertices for a specified phase sequence waveform. The arguments specify the phase sequence waveform and the new set of vertices. This new set of vertices completely replaces the previous set of vertices for the waveform.

Calling format:

```
set_phsseq_vertices(struct wvphsseq *phsseq, struct vertex newv[], int numvertex)
```

The arguments are:

- `phsseq`: pointer to the descriptor for the phase sequence for which the vertices are being changed.
- `newv`: an array of structures of type `vertex` that has `numvertex` elements. Each of the structures specifies one of the new vertices.
- `numvertex`: the count of the new vertices.

14.6.7 find_phase_sequence

Return the phase sequence number (starting at one) specifying the name of the category and the name of the phase sequence. This function can be used to set a Y vertex value that points to a given phase sequence.

Calling format:

```
int find_phase_sequence(char *cat_name, char *seqname)
```

The arguments are:

- `cat_name`: string giving the name of the category. The category name is case sensitive.
- `seqname`: string giving the name of the sequence. The sequence name is case sensitive.

Return value:

- The return value of the routine is the seqence number (starting at 1) if the sequence is found. If not found, this routine returns -1.

14.6.8 locate_phase_sequence

Given the name of a category and a phase sequence name, return a pointer to the phase sequence. If not found, then return a NULL pointer.

Calling format:

```
struct wvphsseq *locate_phase_sequence(char *cat_name, char *seqname)
```

The arguments are:

- `cat_name`: string giving the name of the category. The category name is case sensitive.
- `seqname`: string giving the name of the sequence. The sequence name is case sensitive.

Return value:

- The return value of the routine is a pointer to the phase sequence structure. If not found, this routine returns NULL.

14.6.9 is_phase_used

For the specified shot phase, return a flag indicating which phase sequence the phase is actually specified in (return index of phase sequence if used, 0 if not used). If the shot phase is not actually used, there is no point in loading data for the phase into the real-time cpu.

Calling format:

```
int is_phase_used(struct shotphase *phase)
```

Function Arguments:

- `phase`: The pointer to the descriptor of the shot phase.

Return values:

- The index of the phase sequence is returned if the phase is used, 0 is returned if the phase is not used on any phase sequences.

14.6.10 get_used_phase_list

Given the identifier of a category, return an array of pointers to the descriptors for the phases that are actually specified on any of the phase sequence waveforms for that category. Only phases that actually exist are returned. Vertices on phase sequence waveforms referencing phases that do not actually exist are ignored.

The information is taken from the queue handler's copy so this function can only be called from the `alg_vectors` function or a check must be made on `phase->whichqueue` before the function is called.

The last valid pointer in the returned array is followed by a null pointer to mark the end of the list. The calling routine must free the memory at the location of the return variable when it is no longer needed.

Calling format:

```
struct shotphase **get_used_phase_list(char *catident)
```

The arguments are:

- `catident`: string giving the name of the category identifier.

Return value:

- The return value of the routine is an array of pointers that point to each of the used phases. This array pointer must be freed.

14.6.11 locate_string_in_labels

Locate the given string in the list of labels and return the corresponding value in the list of values. The labels and values are both concatenated strings ending with a null.

Calling format:

```
char *locate_string_in_labels(char *labels, char *values, char *string)
```

The arguments are:

- `labels`: concatenated string of one or more labels ending with a 0 length entry (the last two bytes in the string must be 0).
- `values`: concatenated string of one or more labels ending with a 0 length entry (the last two bytes in the string must be 0).
- `string`: the string to search for in the labels.

Return value:

- The return value is a pointer to the corresponding value if the string is found, or NULL if the string is not found.

14.6.12 fill_dma_region_constants

Fill in the constants related to the dma_region. If the installation has a different dma_region than the default, then this function must be called for each cpu.

Calling format:

```
int fill_dma_region_constants(STRUCT_DESCRIPTORS *descriptor_in, int rtcpu)
```

The arguments are:

- `descriptor_in`: an array of descriptors describing the dma_region. This is usually generated with a `GEN_` macro which creates the structure.
- `rtcpu`: integer giving the real-time cpu number from 0 to rtcpu_count-1.

Return value:

- The return value of the routine is always 0.

14.6.13 get_nocomment_line

Return a line from a file without a comment. Keep reading until there is something to return. The comment character is an exclamation point. Any comments are removed before returning.

Calling format:

```
char *get_nocomment_line(char *line, int nline, FILE *fptr)
```

The arguments are:

- `line`: pointer to a char to return the next line from the file.
- `nline`: the size of `line` in bytes.
- `fptr`: the file descriptor of the file opened by `fopen`.

Return value:

- The return value is a pointer to `line` that contains any characters before a comment character (exclamation point). If there are no more lines to return, `NULL` is returned.

14.6.14 get_line_with_nocomments

Return a line from a file without a comment. Keep reading until there is something to return. The comment character is an exclamation point. Any comments are removed before returning.

Calling format:

```
char *get_line_with_nocomments(char *line, int nline, FILE *fptr)
```

The arguments are:

- `line`: pointer to a char to return the next line from the file.
- `nline`: the size of `line` in bytes.
- `fptr`: the file descriptor of the file opened by `fopen`.

Return value:

- The return value is a pointer to `line` that contains any characters before a comment character (exclamation point).

14.6.15 get_algorithm_identifier_by_phase

Given the pointer to a shot phase descriptor, return the pointer to the algorithm identifier.

Calling format:

`int get_algorithm_identifier_by_phase(struct shotphase *phase)`

The arguments are:

- `phase`: a pointer to the descriptor of the shot phase.

Return value:

- A pointer to the algorithm identifier.

14.6.16 locate_phase_byname

Return the phase number of the specified shot phase descriptor for the specified category. If the phase cannot be located and an errorname string is provided, then issue an error message. By setting errorname to NULL, a calling routine can use this function to test if a phase with a particular name exists.

Calling format:

`int locate_phase_byname(int category, char *phasename, char *errorname, int which_queue)`

The arguments are:

- `category`: the category number starting at 1.
- `phasename`: a pointer to the phase name to locate.
- `errorname`: a pointer to the error string to use if the phase cannot be located. Set this to NULL to test that the phase exists.
- `int which_queue`: The queue for which the data should be returned: `DOIO` or `DOQUEUE`.

Return value:

- The phase number of the phase if found, else zero.

14.6.17 get_virtual_cpu_num

Search for the virtual cpu number for the given category and physical cpu number and return it. If no virtual cpu is found for the input category and physical cpu, return -1. Both arguments start at 1.

Calling format:

```
int get_virtual_cpu_num(int category,int pcpu_num)
```

The arguments are:

- `category`: the category number starting at 1.
- `pcpu_num`: the physical cpu number starting at 1.

Return value:

- The virtual cpu number if found, else -1.

14.6.18 get_physical_cpu_num

Search for the physical cpu number for the given category and virtual cpu number and return it. If no physical cpu is found for the input category and virtual cpu, return -1. Both arguments start at 1.
Calling format:

```
int get_physical_cpu_num(int category,int vcpu_num)
```

The arguments are:

- `category`: the category number starting at 1.
- `vcpu_num`: the virtual cpu number starting at 1.

Return value:

- The physical cpu number if found, else -1.

14.6.19 mult_4

Round up input number to nearest int value which is multiple of 4. Returns rounded value.
Calling format:

```
int mult_4(int number)
```

14.6.20 mult_8

Round up input number to nearest int value which is multiple of 8. Returns rounded value.
Calling format:

```
int mult_8(int number)
```

14.6.21 mynint

Compute the nearest integer given an input float value.

Calling format:

```
int mynint(float input)
```

14.6.22 mynuint

Compute the nearest unsigned integer given an input float value.

Calling format:

```
int mynuint(float input)
```

14.6.23 swap_i2

Swap bytes for an array of short integers to convert endianness. This function takes a char array so that the short integers can be unaligned. Note that this function also takes a count in bytes rather than the number of shorts.

Calling format:

```
int swap_i2(char *buffer,int numbytes)
```

The arguments are:

- `buffer`: a pointer to the input buffer.
- `numbytes`: the total number of bytes in the array.

Return value:

- This function always returns a value of 0.

14.6.24 swap_i4

Swap bytes for an array of integers to convert endianness. This function takes a char array so that the integers can be unaligned. Note that this function also takes a count in bytes rather than the number of ints.

Calling format:

```
int swap_i4(char *buffer,int numbytes)
```

The arguments are:

- `buffer`: a pointer to the input buffer.
- `numbytes`: the total number of bytes in the array.

Return value:

- This function always returns a value of 0.

14.6.25 swap_i8

Swap bytes for an array of 64-bit integers to convert endianness. This function takes a char array so that the 64-bit integers can be unaligned. Note that this function also takes a count in bytes rather than the number of 64-bit integers.

Calling format:

```
int swap_i8(char *buffer,int numbytes)
```

The arguments are:

- `buffer`: a pointer to the input buffer.
- `numbytes`: the total number of bytes in the array.

Return value:

- This function always returns a value of 0.

14.6.26 str2lower

Create an all lowercase version of input string in output string. The two string pointers can point to the same address.

Calling format:

```
int str2lower(char *input_string,char *output_string)
```

The arguments are:

- `input_string`: a pointer to the input character string.
- `output_string`: a pointer to the output character string.

Return value:

- This function always returns a value of 0.

14.6.27 str2upper

Create an all uppercase version of input string in output string. The two string pointers can point to the same address.

Calling format:

```
int str2upper(char *input_string,char *output_string)
```

The arguments are:

- `input_string`: a pointer to the input character string.
- `output_string`: a pointer to the output character string.

Return value:

- This function always returns a value of 0.

14.6.28 str_trim

Trim blanks and tabs from the beginning and end of the input string. Note: the input string is modified!

Calling format:

```
void str_trim(char *string)
```

The arguments are:

- `string`: a pointer to the input character string.

Return value:

- This function does not return a value. The input string is modified!

14.6.29 str_cmp

Compare a string to a pattern that can have wildcards. The two wildcards used are "*" (match any number of characters) and "?" (match one character). Note that case is NOT ignored.

Calling format:

```
int str_cmp(char *string1, char *string2)
```

The arguments are:

- `string1`: a pointer to the pattern string which may contain wildcards.
- `string2`: a pointer to the string which is being matched against the pattern.

Return value:

- This function returns one of three values:
  - +1 if string1 ¿ string2
  - 0 if string1 = string2
  - -1 if string1 ¡ string2

14.6.30 str_cmpi

Compare a string to a pattern that can have wildcards. The two wildcards used are "*" (match any number of characters) and "?" (match one character). Note that case IS ignored.
Calling format:

```
int str_cmpi(char *string1, char *string2)
```

The arguments are:

- `string1`: a pointer to the pattern string which may contain wildcards.
- `string2`: a pointer to the string which is being matched against the pattern.

Return value:

- This function returns one of three values:
  - +1 if string1 ¿ string2
  - 0 if string1 = string2
  - -1 if string1 ¡ string2

14.6.31 str_find

Find the index in an array of strings which matches another string. Return the index of the string in the array (0-nstrings-1), or -1 if not found. Case is NOT ignored.
Calling format:

```
int str_find(char *string, char **strings, int nstrings)
```

The arguments are:

- `string`: a pointer to the input character string to match.
- `strings`: an array of character strings to look in.
- `nstrings`: the number of strings in the array.

Return value:

- This function returns the index into the array of the matching string or -1 if no match is found.

14.6.32 str_find_ignore

Find the index in an array of strings which matches another string. Return the index of the string in the array (0-nstrings-1), or -1 if not found. Case IS ignored.
Calling format:

```
int str_find_ignore(char *string, char **strings, int nstrings)
```

The arguments are:

- `string`: a pointer to the input character string to match.
- `strings`: an array of character strings to look in.
- `nstrings`: the number of strings in the array.

Return value:

- This function returns the index into the array of the matching string or -1 if no match is found.

14.6.33 str_nitems

Find the number of items in a string. Items are separated by whitespace (either space or tabs or control characters).
Calling format:

```
int str_nitems(char *string)
```

The arguments are:

- `string`: a pointer to the input character string.

Return value:

- This function returns the count of items in the string or 0 if the string pointer is NULL or the first byte is a 0.

14.6.34 strlowcase

Create an all lowercase version of input string in output string. The two string pointers can point to the same address.

Calling format:

```
void strlowcase(char *output_string, char *input_string)
```

The arguments are:

- `output_string`: a pointer to the output character string.
- `input_string`: a pointer to the input character string.

Return value:

- This function does not return a value.

14.6.35 strupcase

Create an all uppercase version of input string in output string. The two string pointers can point to the same address.

Calling format:

```
void strupcase(char *output_string, char *input_string)
```

The arguments are:

- `output_string`: a pointer to the output character string.
- `input_string`: a pointer to the input character string.

Return value:

- This function does not return a value.

14.6.36 strpos

Find a substring and return the index into the string. Case is NOT ignored.

Calling format:

```
int strpos(char *string, char *substring)
```

The arguments are:

- `string`: a pointer to the character string to search.
- `substring`: a pointer to the character string being searched for.

Return value:

- This function returns 0 to length-1 if found, -1 if not.

14.6.37 strreplace

Replace the first occurance of a substring with a replacement string. Always return the result in a malloc'ed string. If the substring does not exist, then a copy of the input string is returned. Case is NOT ignored.

Calling format:

```
char *strreplace(char *string, char *substring, char *replacement)
```

The arguments are:

- `string`: a pointer to the character string to search.
- `substring`: a pointer to the character string being searched for.
- `replacement`: a pointer to the character string which is substituted for the substring if the substring is found.

Return value:

- This function returns a malloc'ed string with the replacement string in place of the first occurance of the substring if it is found, else a duplicate of the input string is returned. The return value must be freed.

14.6.38 pcs_abort

This function is called in order to cause a core dump of the executing process. It is used when an error is encountered from which it is not possible to recover. Typically, this type of error is a programming error which must be corrected by the algorithm author.

Normally, an error caused by the settings chosen by the PCS operator should not be handled by calling this function. Instead, the cause of the error should be communicated to the operator (Sec. 2.14) and, perhaps, an entry added to the raw data problem list (Sec. 2.14.1).

Calling format:

```
pcs_abort()
```

14.6.39 check_access_control

This function is called in order to determine the status of a specified access control feature for a specified data item. For instance, this function is used to determine within the `alg_parameters` function whether the data for a static data item should be set to default values.

Calling format:

```
int check_access_control(char *data_item_name,
                         struct shotphase *phase,
                         int check_type)
```

Function arguments:

- `data_item_name`: the name of the data item for which the access control check should be made.
- `phase`: a pointer to the descriptor of the shot phase in which the data item is defined.
- `check_type`: the type of check that should be made. Possible values are:
  - `AC_INITIALIZE`: check whether the data item should be initialized. The decision is made based on whether the `AC_NORESTORE` bit (Sec. 2.17.3) is set in the access control mask (Sec. 2.17.2) and the current stage of the PCS setup restore procedure (Sec. 2.17.3.1). The function returns 1 if the initialization should be done, 0 if the initialization should not be done.

Return value:

- The function returns 1 or 0. The meaning depends on the value of the input argument `check_type`.

14.6.40 create_start_time_list

For the specified category and shot phase, search all of the phase sequence waveforms for the category for each time that the waveform indicates that the specified shot phase should start. Return a list of these time values. The information comes from either the io handler's or the queue handler's copy.

If the shot phase is zero, then return a list of start times appropriate for the phase sequence waveforms.

Note that the first phase sequence for a given category is always relative to the discharge t=0 (i.e. absolute time). Others are always relative to t=0. So, vertices found on the first phase sequence have their X values adjusted by primary_start_time while vertices on other phase sequences are not altered.

Calling format:

```
struct starttime *create_start_time_list(
     int category,
     int phase,
     int whichone)
```

Function arguments:

- `category`: the category code in the cat_alg structure (1-based).
- `phase`: the phase number of the phase to check.
- `whichone`: whether queue or io handler is source of information.

Return value:

- The function returns a pointer to a linked list of `starttime` structures. This structure has the following fields.

```
struct starttime
{
    float time1;  /* the time the phase starts */
    float time2;  /* when the switch to another phase is made */
    int absolute; /* are the time values relative or absolute? */
    char *name;   /* name of the phase sequence where the phase change
                     times are defined (do not free) */
    struct starttime *next; /* next structure in the list */
};
```

Example:

```
    struct starttime *data,*nextstart;
    /*
    Get the linked list of start times for this phase.
    This list will be used here and then deleted.
    */
    data = create_start_time_list(phase->category,
           phase->number, phase->whichqueue);
    ...

    /*
    Delete the list.
    */
    nextstart = data;
    while(nextstart->name != NULL)
    {
       data = nextstart;
       nextstart = nextstart->next;
       free(data);
    }
    free(nextstart);
```

14.6.41 get_first_start_time

For the specified shot phase, search all of the phase sequence waveforms for the category for the first time that the phase is referenced. Return this time value. If the phase is not referenced at all, 0.0 is returned. The time value returned can be used as an offset to time values that are relative to the start of the phase in error messages to provide more informative time values for the user. However, this is only a quick solution since it will not always be the case the user is most interested in the first time that the phase is referenced.

The phase sequence waveforms are searched in order starting with the primary phase sequence.

Calling format:

```
void get_first_start_time(
     struct shotphase *phase,
     float *return_time)
```

Function arguments:

- `phase`: a pointer to the descriptor of the shot phase.

- `return_time`: the first time at which the phase is referenced on a phase sequence, 0.0 if not referenced at all.

There is no return value.
Example:

```
 float start_time;
 get_first_start_time(phase,&start_time);
```

14.6.42 search_target_maps

Search for a target pointname in the target vector maps. Return the physical cpu number and the index into the target vector on that physical cpu. Target pointnames either have a "T" or an "I" as the third letter.
Calling format:

```
int search_target_maps(char *input_name, int *on_cpu,int *point_index)
```

Function arguments:

- `input_name`: a pointer to the pointname.
- `on_cpu`: the return value of the physical cpu.
- `point_index`: the return value of the target vector element.

Return value:

- The function returns a 0 if the name is found, -1 if not.

14.6.43 search_shape_maps

Search for a shape pointname in the shape vector maps. Return the physical cpu number and the index into the shape vector on that physical cpu. Shape pointnames have a "S" as the third letter.
Calling format:

```
int search_shape_maps(char *input_name, int *on_cpu,int *point_index)
```

Function arguments:

- `input_name`: a pointer to the pointname.

- `on_cpu`: the return value of the physical cpu.
- `point_index`: the return value of the shape vector element.

Return value:

- The function returns a 0 if the name is found, -1 if not.

14.6.44 search_error_maps

Search for an error pointname in the error vector maps. Return the physical cpu number and the index into the error vector on that physical cpu. Error pointnames have an "E" as the third letter.

Calling format:

```
int search_error_maps(char *input_name, int *on_cpu,int *point_index)
```

Function arguments:

- `input_name`: a pointer to the pointname.
- `on_cpu`: the return value of the physical cpu.
- `point_index`: the return value of the error vector element.

Return value:

- The function returns a 0 if the name is found, -1 if not.

14.6.45 search_p_maps

Search for a pvector pointname in the error vector maps. Return the physical cpu number and the index into the error vector on that physical cpu. The pvector is archived using the error vector maps replacing the third letter of the pointname with a "P".

Calling format:

```
int search_p_maps(char *input_name, int *on_cpu,int *point_index)
```

Function arguments:

- `input_name`: a pointer to the pointname.
- `on_cpu`: the return value of the physical cpu.
- `point_index`: the return value of the pvector element.

Return value:

- The function returns a 0 if the name is found, -1 if not.

14.6.46 search_command_maps

Search for a command pointname in the command vector maps. Return the physical cpu number and the index into the command vector on that physical cpu. Command pointnames have a ”C” as the third letter.

Calling format:

```
int search_command_maps(char *input_name, int *on_cpu,int *point_index)
```

Function arguments:

- `input_name`: a pointer to the pointname.
- `on_cpu`: the return value of the physical cpu.
- `point_index`: the return value of the command vector element.

Return value:

- The function returns a 0 if the name is found, -1 if not.

14.6.47 search_fpcom_maps

Search for a fpcom pointname in the command vector maps. Return the physical cpu number and the index into the fpcom vector on that physical cpu. The fpcom vector is archived using the command vector maps replacing the third letter of the pointname with an ”F”.

Calling format:

```
int search_fpcom_maps(char *input_name, int *on_cpu,int *point_index)
```

Function arguments:

- `input_name`: a pointer to the pointname.
- `on_cpu`: the return value of the physical cpu.
- `point_index`: the return value of fpcom vector element.

Return value:

- The function returns a 0 if the name is found, -1 if not.

14.6.48 badstatus

This function simply logs an error message and returns a 0 status. It can be used to simplify error handling. Note that this function returns 0 which is considered an error because it was written under VMS where 1 is a good status and 0 is a bad status.

Calling format:

```
int badstatus(int status,char *name)
```

Function arguments:

- `status`: the integer error value.
- `name`: the name of the function.

Return value:

- The function always returns the value of `FAILURE` which is equal to 0.

Example:

```
ier = 10;
if(ier != 0) return(badstatus(ier,"myfunc"));
```

```
sends the log message:

HR:MN:SC-M-(wv)  bad status 10 from myfunc
```

14.6.49 ckstatus

This function checks the value of the input status. If the status is the value 1 (which is equal to `SUCCESS`), it returns 1. Else, it calls the `badstatus` function 14.6.48 which logs a message and returns 0 (which is equal to `FAILURE`). It can be used to simplify error handling. Note that this function returns the opposite of what linux uses for SUCCESS/FAILURE because it was written under VMS where 1 is a good status and 0 is a bad status.

Calling format:

```
int ckstatus(char *name,int status)
```

Function arguments:

- `name`: the name of the function.

- `status`: the integer error value.

Return value:

- The function returns the value 1 if the input status is 1, else it returns the value 0.

Example:

```
ckstatus("myfunc",myfunc(...));
```

Chapter 15

Routines used in the real time code

This chapter describes all the routines needed in the real time code along with the many utility routines that can be used. Included are sections on functions that communicate to the user Sec. 15.1, functions used to communicate between real time processors Sec. 15.2, installation required functions Sec. 15.3, and optional installation functions Sec. 15.4.

15.1 Routines for communicating to the user

15.1.1 rtmsg_log

This routine is used in the real-time code to send a message to the message server and log it to the process' log file. It should be used to notify users of significant errors during setup of the shot or during the shot. The message gets buffered in a region of memory whose size is given by the installation specific macro `RTMSG_LOG_SIZE` which is 10000 by default. Messages are sent after all setup is complete at first lockout and again after the shot is finished. Messages of type `PCSERR` will be counted by the view log, the count displayed, and will cause the status light to turn yellow (Sec. 7.1). The current time during the shot will automatically be inserted at the beginning of the message if the messsage is buffered during the shot.

Calling format:

```
int rtmsg_log(struct rt_heap_misc *rtheap,
              int msg_code, char *strm_fmt, ...)
```

Function Arguments:

- `rtheap`: Pointer to the structure that defines the layout of the PCS memory heap Sec. 13.3.

- `int msg_code`: macro value indicating the source of the message. In most cases, this value should be `PCSMSG` or `PCSERR`.
- `char *strm_fmt`: the format passed on to sprintf.
- `...`: zero or more variables which match the format.

Example:

```
rtmsg_log(rtheap,PCSERR,"ABORT dring shot\n")
```

15.1.2 rtprintf

This function is used to write messages in the real time code. The messages get buffered during the shot and then they are written to the real time log after the shot using the function `rtprintf_end` (Sec. 15.1.2.2). See the function `rtprintf_init` (Sec. 15.1.2.1) to set up the message buffer.

Calling format:

```
    int rtprintf(char *strm_fmt, ...)
```

Function Arguments:

- `char *strm_fmt`: the format passed on to sprintf.
- `...`: zero or more variables which match the format.

Return value:
The same return value from system function `vsnprintf`. If positive, this is the number of bytes added to the message buffer. Otherwise, the return value is -1 indicating an error.

Example:

```
rtprintf("%u: input=%f\n",rtheap->currenttime,
         rtheap->physicsvector[D_PS1-1]);
```

15.1.2.1 rtprintf_init

This function is usually called from the system algorithm's init function. It will store away the pointer to the message buffer region so that the pointer nor the memory region's size need to be passed into `rtprintf`.

Calling format:

```
int rtprintf_init(char *buffer, int size)
```

Function Arguments:

- buffer: A pointer to the memory region. If NULL, then a buffer will be allocated using the `malloc` function. It is probably better to use a scratch parameter data block. The size could be adjusted from the user interface by changing another parameter data block containing an array of sizes, one for each real time cpu.
- size: The size of the region. If this value is 0 and the buffer is NULL, then 1000000 bytes will be allocated.

Return value:
This function always returns a value of 0.
Example:

```
#ifdef CONTROLDEF
#define BX_SYSTEM_RTMESSAGE_BUFFER_SIZE 1
#endif

#ifdef WAVEFORMS
/* StaticData */
{
 0,                              /* ignored */
 {0.0, 0.0, 0.0, 0.0},           /* ignored */
 "RTmessage buffer size",        /* label identifying param data */
 "generic_editor",               /* name of idl routine to handle the data */
 " ",                          /* idl specific, e.g., format for matrix editor */
 " ",                            /* ignored */
 1.0,                            /* ignored */
 0.0,                            /* ignored */
 " ",                            /* ignored */
 SD_LABELED_INT_ARRAY,           /* indicates not a waveform, 40 chars max */
 C_CODE,                         /* control category index */
 A_CODE,                         /* control algorithm index */
 {PA_SYSTEM(BX_SYSTEM_RTMESSAGE_BUFFER_SIZE), 0}, /* target vectors affected */
 0,                              /* ignored */
 0,                              /* ignored */
 {0.0},                          /* ignored */
 0.0,                            /* ignored */
 0.0,                            /* ignored */
 1,                             /* subset number */
 0,                             /* access control */
},
#endif
/*
In the sysmain_parameters function
*/
{
int *sizes;
char *label;
char **labels;
int j;

/*
Create a block that has the size of the realtime message buffer
for each cpu.
*/
     labels = (char **)malloc(rtcpu_count * sizeof(char *));
     sizes = (int *)malloc(rtcpu_count * sizeof(int));

     for(j=0; j<rtcpu_count; j++)
     {
         label = malloc(64 * sizeof(char));
         sizes[j] = 0;
         sprintf(label,"Size of RTmessage buffer for cpu %d",j+1);
         labels[j] = label;
     }
     putblock_labeled_int_array(phase,
                                "RTmessage buffer size",
                                "USER|ARCHIVE|RTCPUS=ALL",
                                BX_SYSTEM_RTMESSAGE_BUFFER_SIZE,
                                rtcpu_count,
                                sizes,
                                labels,
                                sizes);
    for(j=0; j<rtcpu_count; j++)
    {
       free(labels[j]);
    }
    free(labels);
    free(sizes);
```

```
}

/*
In the sysmain_init function
*/
{
void **parameter_data;
int *sizes;

    parameter_data = (void **)phase->parameter_data;
    sizes = (int *)parameter_data[BX_SYSTEM_RTMESSAGE_BUFFER_SIZE-1];
/*
Pass NULL in first argument to use malloc.
Else, could use scratch block.
*/
    rtprintf_init(NULL,sizes[CPU_NUM-1]);
}
```

15.1.2.2 rtprintf_end

This function is usually called from the system algorithm's cleanup function. It will print the messages found in the message buffer.

Calling format:

```
    int rtprintf_end(FILE *stream_in)
```

Function Arguments:

- stream_in: A pointer to the FILE stream to write to. If the value is NULL, then `stderr` is used.

Return value:
This function always returns a value of 0.

Example:

```
/*
In the sysmain_cleanup function
*/
{
    rtprintf_end(stderr);
}
```

15.2 Interprocessor communication routines

This section describes utility functions used for communication between the real time processors. All are used in the real time code with the exception of `rtmessage_register` (Sec. 14.4.1) which is used in the waveform server.

15.2.1 rtmessage_register

Register a real time message. This function is called in the waveform server code to store information about the message that will be sent from one real time processor to another. See Sec. 14.4.1 for details of how to setup the message in the waveform server.

15.2.2 rtmessage_get_buffer

Get the pointer to the send buffer on the message source processor. For additional background information on the use of this function see Sec. 2.9.4.

Calling format:

```
int  rtmessage_get_buffer(
                struct rt_heap_misc *rtheap,
                char *name)
```

Function Arguments:

- `rtheap`: Pointer to the structure that defines the layout of the PCS memory heap (Sec. 13.3).
- `name`: The name to use for the message.

Return value:
This function returns a pointer to the next send buffer. In the case of multiple buffers, a pointer to the next buffer is returned wrapping around to the first buffer when necessary. If the message was registered to not have a send buffer, then this function returns a NULL pointer, which indicates a programming error.

15.2.3 rtmessage_write

Send the message to the destination processor. The entire message is sent from beginning to end. For additional background information on the use of this function see Sec. 2.9.4.

Calling format:

```
int  rtmessage_write(
                 struct rt_heap_misc *rtheap,
                 char *name,
                 void *buffer)
```

Function Arguments:

- `rtheap`: Pointer to the structure that defines the layout of the PCS memory heap (Sec. 13.3).
- `name`: The name to use for the message.
- `buffer`: The buffer with the data to send.

Return value:
This function returns the status from the call to the `rtmessage_write_rtmessage` function (Sec. 15.2.14).

15.2.4 rtmessage_partial_write

This is a special function that allows the author to completely control what happens if the message needs to be sent in multiple pieces. This is not recommended, but this function is available in case it is needed.

This function has arguments that specify the offset in the message to start at, the number of bytes to send, and a bitmask that indicates what steps to perform with each call. For additional background information on the use of this function see Sec. 2.9.4.

Calling format:

```
int  rtmessage_partial_write(
                            struct rt_heap_misc *rtheap,
                            char *name,
                            void *buffer,
                            int offset,
                            int size,
                            int bitmask)
```

Function Arguments:

- `rtheap`: Pointer to the structure that defines the layout of the PCS memory heap (Sec. 13.3).

- `name`: The name to use for the message.
- `buffer`: The buffer with the data to send.
- `offset`: The offset of the first byte in the buffer to send.
- `size`: The size of the buffer in bytes to send.
- `bitmask`: A bitmask that indicates what actions to perform in this call. The following macros define each bit.
  - `ADD_BEGINNING_COUNTER`: add in the beginning counter. The input buffer must be the beginning of the buffer and the input offset must be zero.
  - `ADD_ENDING_COUNTER`: add in the ending counter. The input buffer, offset, and size must all be given such that the end of the message will be sent along with the ending counter.
  - `INCREMENT_BUFFER`: increment the buffer to the next one in a multiple receive buffer. This bit needs to be set on the LAST call that finishes sending the message.
  - `SEND_MESSAGE`: actually send the message. This bit is probably going to be set in all calls to this function. If not set, the function could do one or more of the other actions depending on which other bits in the bitmask are set.

Return value:
This function returns the status from the call to the `rtmessage_write_rtmessage` function (Sec. 15.2.14).

Example:
Send a message in three parts. The message is 80 bytes long, uses counters at the beginning and end of the buffer, and there are multiple receive buffers.

```
rtmessage_partial_write(rtheap,"partial example",
    &buffer[0],0,20,ADD_BEGINNING_COUNTER|SEND_MESSAGE);
rtmessage_partial_write(rtheap,"partial example",
    &buffer[20],20,40,SEND_MESSAGE);
rtmessage_partial_write(rtheap,"partial example",
    &buffer[60],60,20,ADD_ENDING_COUNTER|INCREMENT_BUFFER|SEND_MESSAGE);
```

The receiver would not return the message until the third piece arrives.

15.2.5 rtmessage_read

Get the pointer to the message. For additional background information on the use of this function see Sec. 2.9.4.

Calling format:

```
int  rtmessage_read(
                  struct rt_heap_misc *rtheap,
                  char *name
                 )
```

Function Arguments:

- `rtheap`: Pointer to the structure that defines the layout of the PCS memory heap (Sec. 13.3).
- `name`: The name to use for the message.

Return value:
This function returns a pointer to the message buffer. In some cases, it will return a NULL pointer if a new message is requested and one has not yet arrived. This function may also wait for a message to arrive if the the kwyword "TIMEOUT=" was specified when the message was registered. For more information see Sec. 14.4.1.

Example 1:
This is an example of a real time scope application where multiple receive buffers are used. Every message that arrives will be processed even if the processor gets an interrupt since there should be enough buffers to handle the messages that arrive during the interrupt. See Example 4 in Sec. 14.4.2 for the registration of this message.

```
{
int *buffer;

    /* loop until there are no new messages */
    while(1)
    {
        buffer = rtmessage_read(rtheap,"alg case 4");
        if(buffer == NULL) break;

        /* process the message, the first int is the current time */

        if((unsigned int)(buffer[0]) >= rtheap->currenttime) break;
    }
}

Example 2:\\
Read a simple message from another real time processor.
Always check if the buffer address returned is NULL.
This code can be used for most other cases.
```

```
\begin{verbatim}
{
int *buffer;

    buffer = rtmessage_read(rtheap,"alg case 2");
    if(buffer != NULL)
    {
    }
}
```

15.2.6 rtcipc_write

Copy data to a buffer on any of the PCS processors at an arbitrary location in the PCS memory heap. For additional background information on the use of this function see Sec. 2.9.2.

Calling format:

```
int  rtcipc_write(
                  struct rt_heap_misc *rtheap,
                  int destination_physical_cpu_number,
                  int destination_offset,
                  void *input_data,
                  int input_length,
                  struct rtcipc_struct *rtcipc_info
                 )
```

Function Arguments:

- `rtheap`: Pointer to the structure that defines the layout of the PCS memory heap (Sec. 13.3).
- `destination_physical_cpu_number`: The number (based at 1) of the physical CPU to which the data should be copied. This value should always be specified by using one of the macros discussed in Sec. 2.16.3. The names of these macros are based on virtual CPU number which is how algorithm code should always refer to a particular CPU (as discussed in Sec. B).
- `destination_offset`: The offset (in bytes) from the beginning of the PCS memory heap on the destination CPU of the buffer to which the data should be copied. See Sec. B for background information on how to determine the value to provide here.

- `input_data`: Pointer to the data to be copied.
- `input_length`: The number of bytes of data to be copied.
- `rtcipc_info`: Pointer to an installation-specific structure that provides extra parameters, if required. This structure might provide information, for instance, that depends on the type of interprocessor communication hardware. Or, this structure might specify some additional behavior for the communication functions. If this parameter isn't required, specify `NULL`.

Return value:
This function returns 0 if there were no errors. In the case of an error, the nonzero value returned is installation-specific.

15.2.7 rtcipc_write_commbuff

Copy an array of values to a specified location in the communication vector on any of the PCS processors. For additional background information on the use of this function see Sec. 2.9.2. In particular, see the note in Sec. 2.9.3 about the case in which the data are to be copied to the same CPU as the one on which the data originate.

Calling format:

```
int rtcipc_write_commbuff(
                          struct rt_heap_misc *rtheap,
                          int destination_physical_cpu_number,
                          int element,
                          void *input_data,
                          int input_length,
                          struct rtcipc_struct *rtcipc_info
                         )
```

Function Arguments:

- `rtheap`: Pointer to the structure that defines the layout of the PCS memory heap (Sec. 13.3).
- `destination_physical_cpu_number`: The number (based at 1) of the physical CPU to which the data should be copied. This value should always be specified by using one of the macros discussed in Sec. 2.16.3. The names of these macros are based on virtual CPU number which is how algorithm code should always refer to a particular CPU (as discussed in Sec. B).

- `element`: The number (based at 1) of the element of the communication vector on the destination CPU to which the data should be copied. The communication vector is nominally an array of type `float`, so this argument (minus 1) is the index in an array of that type.
- `input_data`: Pointer to the data to be copied.
- `input_length`: The number of values with length the same as a value of type `float` that should be copied. In other words, this function behaves like it is copying an array of type `float`.
- `rtcipc_info`: Pointer to an installation-specific structure that provides extra parameters, if required. This structure might provide information, for instance, that depends on the type of interprocessor communication hardware. Or, this structure might specify some additional behavior for the communication functions. If this parameter isn't required, specify `NULL`.

Return value:
This function returns 0 if there were no errors. In the case of an error, the nonzero value returned is installation-specific.

15.2.8 rtcipc_write_commbuff_float

Copy a single value of type `float` to a specified location in the communication vector on any of the PCS processors. For additional background information on the use of this function see Sec. 2.9.2. In particular, see the note in Sec. 2.9.3 about the case in which the data are to be copied to the same CPU as the one on which the data originate.

Calling format:

```
int  rtcipc_write_commbuff_float(
                                 struct rt_heap_misc *rtheap,
                                 int destination_physical_cpu_number,
                                 int element,
                                 float *input_data
                                )
```

Function Arguments:

- `rtheap`: Pointer to the structure that defines the layout of the PCS memory heap (Sec. 13.3).

- `destination_physical_cpu_number`: The number (based at 1) of the physical CPU to which the data should be copied. This value should always be specified by using one of the macros discussed in Sec. 2.16.3. The names of these macros are based on virtual CPU number which is how algorithm code should always refer to a particular CPU (as discussed in Sec. B).
- `element`: The number (based at 1) of the element of the communication vector on the destination CPU to which the data should be copied. The communication vector is nominally an array of type `float`, so this argument (minus 1) is the index in an array of that type.
- `input_data`: Pointer to the data to be copied. This function copies a single `float`.

Return value:
This function returns 0 if there were no errors. In the case of an error, the nonzero value returned is installation-specific.

15.2.9 rtcipc_write_commbuff_int

Copy a single value of type `int` to a specified location in the communication vector on any of the PCS processors. For additional background information on the use of this function see Sec. 2.9.2. In particular, see the note in Sec. 2.9.3 about the case in which the data are to be copied to the same CPU as the one on which the data originate.

Calling format:

```
int rtcipc_write_commbuff_int(
                              struct rt_heap_misc *rtheap,
                              int destination_physical_cpu_number,
                              int element,
                              int *input_data
                             )
```

Function Arguments:

- `rtheap`: Pointer to the structure that defines the layout of the PCS memory heap (Sec. 13.3).
- `destination_physical_cpu_number`: The number (based at 1) of the physical CPU to which the data should be copied. This value should always be specified by using one of the macros discussed in Sec. 2.16.3. The names of these macros are based on virtual CPU number which is how algorithm code should always refer to a particular CPU (as discussed in Sec. B).

- `element`: The number (based at 1) of the element of the communication vector on the destination CPU to which the data should be copied. The communication vector is nominally an array of type `float`, so this argument (minus 1) is the index in an array of that type. This function assumes that the size of a `float` is the same as the size of a `int`.

- `input_data`: Pointer to the data to be copied. This function copies a single `float`.

Return value:
This function returns 0 if there were no errors. In the case of an error, the nonzero value returned is installation-specific.

15.2.10 rtcipc_write_install_buffer

Copy any amount of data to a general location on any of the PCS processors. For additional background information on the use of this function see Sec. 2.9.2. In particular, see the note in Sec. 2.9.3 about the case in which the data are to be copied to the same CPU as the one on which the data originate.

Calling format:

```
int rtcipc_write_install_buffer(
                              struct rt_heap_misc *rtheap,
                              int destination_physical_cpu_number,
                              int input_offset,
                              void *input_data,
                              int input_length
                             )
```

Function Arguments:

- `rtheap`: Pointer to the structure that defines the layout of the PCS memory heap (Sec. 13.3).

- `destination_physical_cpu_number`: The number (based at 1) of the physical CPU to which the data should be copied. This value should always be specified by using one of the macros discussed in Sec. 2.16.3. The names of these macros are based on virtual CPU number which is how algorithm code should always refer to a particular CPU (as discussed in Sec. B).

- `input_offset`: The offset to the location in the installation buffer where the data should be put.

- `input_data`: Pointer to the data to be copied.
- `input_length`: Number of bytes to copy.

Return value:
This function returns 0 if there were no errors. In the case of an error, the nonzero value returned is installation-specific.

The installation buffer is usually set up as a structure which is the same size on each of the PCS processors. Fields may include a cpu status array, room for raw data, an abort array, etc. These are usually data which get written once (like an abort flag) or written over (like raw data passed from an acquisition cpu). The idea is that the destination cpu can use whatever is sitting there at any time before, during, or after a shot.

This structure must be set up in the waveform server in the `alg_parameters` routine of one of the categories. The pointer to the installation buffer is then stored in `rtheap->install_buffer` on each processor. An example of setting up and using the install buffer follows.

Example:

```
#ifdef CONTROLDEF

/*
Raw data is 2 bytes as is status register and diodata.
Raw data is followed by the time which is 4 bytes.
When the time changes, the data will have arrived.
The D/A outputs are also two bytes followed by the time.
Again, when the time changes, the D/A values will have
been updated.
*/
#define MAX_CPUS 2
#define SIZE_PASSED_DATA        (NUMCHANNELS + MAX_CPUS * 2)
#define CPU_STATUS_OFFSET       (SIZE_PASSED_DATA * 2)
#define CPU_RETURN_STATUS_OFFSET (CPU_STATUS_OFFSET + MAX_CPUS * 4)

#define STATUS_REGISTER_OFFSET  (CPU_RETURN_STATUS_OFFSET + MAX_CPUS * 4)
#define DIODATA_OFFSET          (STATUS_REGISTER_OFFSET + 2)
#define PRESHOT_START_OFFSET    (DIODATA_OFFSET + 2)
#define PRESHOT_ABORT_OFFSET    (PRESHOT_START_OFFSET + 4)

#define DAOUT_GROUP1_OFFSET     (PRESHOT_ABORT_OFFSET + 4)
#define NUM_DAOUT_GROUP1        ((16+1)/2*2)
#define TIME_GROUP1_OFFSET      (DAOUT_GROUP1_OFFSET + 2*NUM_DAOUT_GROUP1)
```

```
#define SIZE_GROUP1             (2*NUM_DAOUT_GROUP1 + 4)

#define DAOUT_GROUP2_OFFSET     (TIME_GROUP1_OFFSET + 4)
#define NUM_DAOUT_GROUP2        ((0+1)/2*2)
#define TIME_GROUP2_OFFSET      (DAOUT_GROUP2_OFFSET + 2*NUM_DAOUT_GROUP2)
#define SIZE_GROUP2             (2*NUM_DAOUT_GROUP2 + 4)

/*
Structure of installation buffer.
*/
#define GEN_install_buffer \
SDSTART(struct install_buffer {     ,D_install_buffer             )\
DGEN   (  short passed_data[SIZE_PASSED_DATA]; ,SHORTTYPE,SIZE_PASSED_DATA)\
DGEN   (  int cpu_status[MAX_CPUS];            ,INTTYPE,MAX_CPUS  )\
                /* used to synchronize cpus */\
DGEN   (  int cpu_return_status[MAX_CPUS];     ,INTTYPE,MAX_CPUS  )\
                /* used to send fault status to each cpu */\
\
DGEN   (  unsigned short status_register;      ,SHORTTYPE,1       )\
                /* status register value from vmea */\
DGEN   (  unsigned short diodata;              ,SHORTTYPE,1       )\
                /* digital input value from vmea */\
DGEN   (  unsigned int preshot_start;          ,INTTYPE,1       )\
                /* preshot signals from digitizers */\
DGEN   (  unsigned int preshot_abort;          ,INTTYPE,1       )\
                /* abort signal */\
\
DGEN   (  short daout_group1[NUM_DAOUT_GROUP1];,SHORTTYPE,NUM_DAOUT_GROUP1)\
                /* D/A output channels group 1 */\
DGEN   (  unsigned int time_group1;            ,INTTYPE,1       )\
                /* time for group 1 */\
\
DGEN   (  unsigned int time_group2;            ,INTTYPE,1       )\
                /* time for group 2 */\
SDEND  (                  };                                    )
GEN_install_buffer

#endif /* end of CONTROLDEF */

#if defined(MASTERVECTORS)
```

```
extern STRUCT_DESCRIPTORS *install_buffer_descriptor;
static GEN_install_buffer

void alg_parameters(struct shotphase *phase)
{
/*
Set the infrastructure descriptor for the install_buffer.
*/
    install_buffer_descriptor = D_install_buffer;
}

#endif /* end of MASTERVECTORS */

#if defined(REALTIME)

void sysmain(struct rt_heap_misc *rtheap)
{
int wcpu;
struct install_buffer *install_buffer;

...

/*
Timeout getting data.  Send the return_status to other cpus.
*/
  rtheap->return_status = TIMEOUT_RETURN;
  install_buffer = (struct install_buffer *)rtheap->install_buffer;
  rtcipc_check_buffer((void *)(&install_buffer->cpu_return_status[CPU_NUM-1]));
  install_buffer->cpu_return_status[CPU_NUM-1] = rtheap->return_status;
  for(wcpu=0;wcpu<rtheap->other_cpu_count;wcpu++)
  {
      rtcipc_write_install_buffer(rtheap,wcpu+1,
              CPU_RETURN_STATUS_OFFSET+wcpu*sizeof(int),
              &install_buffer->cpu_return_status[CPU_NUM-1],sizeof(int));
  }
  return;
}

#endif /* end of REALTIME */
```

15.2.11 rtcipc_write_timetostopflag

Copy a value to the `timetostop_flag` on another CPU. For additional background information on the use of this function see Sec. 2.9.2. In particular, see the note in Sec. 2.9.3 about the case in which the data are to be copied to the same CPU as the one on which the data originate.

Calling format:

```
int rtcipc_write_timetostopflag(
                              struct rt_heap_misc *rtheap,
                              int destination_physical_cpu_number,
                              int *value
                             )
```

Function Arguments:

- `rtheap`: Pointer to the structure that defines the layout of the PCS memory heap (Sec. 13.3).
- `destination_physical_cpu_number`: The number (based at 1) of the physical CPU to which the data should be copied. This value should always be specified by using one of the macros discussed in Sec. 2.16.3. The names of these macros are based on virtual CPU number which is how algorithm code should always refer to a particular CPU (as discussed in Sec. B).
- `value`: Pointer to the int value to write.

Return value:
This function returns 0 if there were no errors. In the case of an error, the nonzero value returned is installation-specific.

15.2.12 rtcipc_read_timetostopflag

Get the value of the `timetostop_flag` from a CPU. For additional background information on the use of this function see Sec. 2.9.2. In particular, see the note in Sec. 2.9.3 about the case in which the data are to be copied to the same CPU as the one on which the data originate.

Calling format:

```
int rtcipc_read_timetostop_flag(struct rt_heap_misc *rtheap,
    int *value,int destination_physical_cpu_number)
```

Function Arguments:

- `rtheap`: Pointer to the structure that defines the layout of the PCS memory heap (Sec. 13.3).
- `value`: Pointer to the int value to read.
- `destination_physical_cpu_number`: The number (based at 1) of the physical CPU from which the data should be read. This value should always be specified by using one of the macros discussed in Sec. 2.16.3. The names of these macros are based on virtual CPU number which is how algorithm code should always refer to a particular CPU (as discussed in Sec. B).

Return value:
This function returns 0 if there were no errors. In the case of an error, the nonzero value returned is installation-specific.

15.2.13 rtcipc_write_paramdata

Copy data to a parameter data block on another CPU. For additional background information on the use of this function see Sec. 2.9.2. In particular, see the note in Sec. 2.9.3 about the case in which the data are to be copied to the same CPU as the one on which the data originate.

Calling format:

```
int rtcipc_write_paramdata(
                              struct rt_heap_misc *rtheap,
                              int destination_physical_cpu_number,
                              int destination_offset,
                              void *input_data,
                              int input_length,
                              struct rtcipc_struct *rtcipc_info
                             )
```

Function Arguments:

- `rtheap`: Pointer to the structure that defines the layout of the PCS memory heap (Sec. 13.3).
- `destination_physical_cpu_number`: The number (based at 1) of the physical CPU to which the data should be copied. This value should always be specified by using

one of the macros discussed in Sec. 2.16.3. The names of these macros are based on virtual CPU number which is how algorithm code should always refer to a particular CPU (as discussed in Sec. B).

- `destination_offset`: The offset (in bytes) from the beginning of the PCS memory heap on the destination CPU of the buffer to which the data should be copied. See Sec. B for background information on how to determine the value to provide here.
- `input_data`: Pointer to the data to be copied.
- `input_length`: The number of bytes of data to be copied.
- `rtcipc_info`: Pointer to an installation-specific structure that provides extra parameters, if required. This structure might provide information, for instance, that depends on the type of interprocessor communication hardware. Or, this structure might specify some additional behavior for the communication functions. If this parameter isn't required, specify `NULL`.

Return value:
This function returns 0 if there were no errors. In the case of an error, the nonzero value returned is installation-specific.

15.2.14 rtcipc_write_rtmessage

This is the most general function to pass data from one processor to another. It needs to be able to handle the use of different memory regions other than the memory heap. See Sec. 2.9.4 for more information about the real time message facility. For additional background information on the use of this function see Sec. 2.9.2. In particular, see the note in Sec. 2.9.3 about the case in which the data are to be copied to the same CPU as the one on which the data originate. This function is called by `rtmessage_write` and should not be called directly.

Calling format:

```
int rtcipc_write_rtmessage(
                              struct rt_heap_misc *rtheap,
                              int destination_physical_cpu_number,
                              int memory_region,
                              int destination_offset,
                              void *input_data,
                              int input_length
                             )
```

Function Arguments:

- `rtheap`: Pointer to the structure that defines the layout of the PCS memory heap (Sec. 13.3).
- `destination_physical_cpu_number`: The number (based at 1) of the physical CPU to which the data should be copied. This value should always be specified by using one of the macros discussed in Sec. 2.16.3. The names of these macros are based on virtual CPU number which is how algorithm code should always refer to a particular CPU (as discussed in Sec. B).
- `memory_region`: The memory region to write the data to. The PCS memory heap is memory region 1.
- `destination_offset`: The offset (in bytes) from the beginning of the memory region on the destination CPU of the buffer to which the data should be copied. This value is known by the PCS and is used in the function `rtmessage_write`.
- `input_data`: Pointer to the data to be copied.
- `input_length`: The number of bytes of data to be copied.

Return value:
This function returns 0 if there were no errors. In the case of an error, the nonzero value returned is installation-specific.

15.3 Required installation real time functions

This section describes functions which are required by the installation. Some have default versions in the file `realmain.c`. If the installation requires a version of a function different from the default, then the new version of the function needs to be added to the file `realinstall.h` and a macro defined that will cause the default function not to be compiled. For example, to not compile the default version of of the function `time_critical`, add the following to `realinstall.h`:

```
#define REPLACE_TIMECRIT
```

15.3.1 real_install_init

This function allows the installation to look at the command line arguments and save away any information. For example, the fourth and arguments are the lockserver host and port number which may be needed if the real time process needs to communicate with the lock-server process.

Calling format:

```
int real_install_init(int argc, char **argv)
```

Function Arguments:

- argc: the count of arguments on the command line. Note that the name of the program is the first argument.
- argv: an array of pointers to the command line arguments.

15.3.2 real_install_main

This function allows the installation to completely replace the main function of the real time process.

Calling format:

```
int real_install_main(int argc, char **argv)
```

Function Arguments:

- argc: the count of arguments on the command line. Note that the name of the program is the first argument.
- argv: an array of pointers to the command line arguments.

15.3.3 set_time_zero

This function handles the synchronization of all the cpus. All installations need to provide this function since synchronization usually depends on the communication hardware involved. In general, one processor, the coordinator, would do some kind of handshaking with each of the other processors.

Calling format:

```
void set_time_zero(struct rt_heap_misc *rtheap,int cpu_num)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.
- cpu_num: the number of the cpu.

15.3.4 wait_for_start_signal

This function is executed after all algorithms have been initialized. Its duty is to wait for a start signal, usually from a digitizer or other hardware, or a message from another processor, and then return so that the time critical code can be executed for the shot. This function is the one which puts the processor into real time mode, if necessary, by turning off system interrupts. All installations need to provide this function since it is hardware specific.

Calling format:

```
int wait_for_start_signal(struct rt_heap_misc *rtheap)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.

15.3.5 time_critical

This function executes all the real time functions. The default version is found in the file `realmain.c`. If a different version is required, then it would be placed in the file `realinstall.h`. Please see the default version of this function for detailed documentation.

Calling format:

```
int time_critical(struct rt_heap_misc *rtheap)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.

15.3.6 update_the_state

This function is executed whenever `rtheap->currenttime` is greater than `rtheap->update_time`. This can happen, in theory, in any of the real time functions. The duty of this function is to test whether it is time to do phase tick processing by executing `new_shot_phase_tick` and/or time to do target vector calculations by executing `new_continuous_target_c`. Both of these functions check and set `rtheap->update_time`.

The default version is found in the file `realmain.c`. If a different version is required, then it would be placed in the file `realinstall.h`. Please see the default version of this function for detailed documentation.

Calling format:

```
int update_the_state(struct rt_heap_misc *rtheap)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.

15.3.7 new_shot_phase_tick

This function computes the next shot phase tick time. The shot phase tick time may be larger than the cycle time so that this function is not necessarily executed every cycle.

The default version is found in the file `phasetick_c.c`. To replace this function with an installation specific version specify a different file in the build of the real time process. Please see the detailed documentation found in the file `phasetick_c.c`..

Calling format:

```
int new_shot_phase_tick(struct rt_heap_misc *rtheap)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.

15.4 Optional installation real time functions

This section describes functions which are optional for the installation. All have default versions in the file `realmain.c`. If the installation requires a version of a function different from the default, then the new version of the function needs to be added to the file `realinstall.h` and a macro defined that will cause the default function not to be compiled. For example, to not compile the default version of of the function `readticks`, add the following to `realinstall.h`:

```
#define REPLACE_READTICKS
```

15.4.1 realtime_cleanup

This function can be added to clean up after a shot, e.g., turn digital outputs off or set D/A outputs to 0. It is executed as soon as the `time_critical` function has finished. The default version in file `realmain.c` does not do anything.

Calling format:

```
int realtime_cleanup(struct rt_heap_misc *rtheap)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.

15.4.2 install_create_memory

This function can be added to create a memory region other than the "real time heap", e.g., one that uses reflective memory. Ordinary shared memory regions are taken care of in the real time code. The default version in file `realmain.c` does not do anything.

Calling format:

```
int install_create_memory(struct rt_memory_region *memory_region)
```

Function Arguments:

- memory_region: a structure that describes the memory region. This function must set the field `addr` which is a void array of two pointers, one for the realtime address of the memory and one for the host real time.

Example:
If reflective memory has the realtime address `rfm_address`, then code would look something like this.

```
#if defined(PCS_RT_CPU)
if (memory_region->type == MEMORY_INSTALL_SPECIFIC)
{
     memory_region->addr[ADDR_RT] = (void *)rfm_address;
     memory_region->addr[ADDR_HOST] = (void *)rfm_address;
     memory_region->shmid = 0;
     return(0);

}
#endif
```

Note that for the now standard combined host real time and real time process, the macro PCS_RT_CPU is used to access the hardware. The host address is needed in order to dump the memory to the S file so it can be analyzed. In runsa mode there wouldn't be any reflective memory.

15.4.3 install_delete_memory

This function can be added to delete a memory region other than the “real time heap”, e.g., one that uses reflective memory. Ordinary shared memory regions are taken care of in the real time code. The default version in file `realmain.c` does not do anything.

Calling format:

```
int install_delete_memory(struct rt_memory_region *memory_region)
```

Function Arguments:

- memory_region: a structure that describes the memory region.

15.4.4 get_num_processors

This function returns the number of processors in the computer. The default version is for linux and simply reads the system file `/proc/cpuinfo` and greps for ”processor” and extracts the number associated with this line in the file.

Calling format:

```
int get_num_processors()
```

Return value:
The function returns the number of processors.

15.4.5 set_affinity

This function tells the system to set the current process to use the specified processor. The input value is a number starting at 0. The default version is found in file `realmain.c` and should handle all cases.

Calling format:

```
int set_affinity(int processor)
```

Function Arguments:

- processor: the processor to put the process on (starting at 0).

Return value:
The function will abort if there is an error, so the return is always a value of 0 to indicate success.

15.4.6 offload_archive_data

This function is needed for streaming off archive data for long discharges. The default empty version is found in file `realmain.c`. See (Sec. 14.6.3) for more information about setting up the archive data area so that it can be streamed off.

Calling format:

```
int offload_archive_data(struct rt_heap_misc *rtheap,char *sbuffer)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.
- sbuffer: pointer to the buffer of offload.

Return value:
The function will return 0 if no error.

15.4.7 get_previous_time

This function returns a pointer to a previously archived time value. The argument given by `count` specifies the number of archived times to go back. It is important to note that not all cycles may be archived. Thus, the returned time may not be equal to the time from `count` cycles ago. This is a useful function when getting a previous pointer to one of the other archived vectors.

Calling format:

```
float *get_previous_time(struct rt_heap_misc *rtheap,int count)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.
- count: an integer giving the number of archived times to go back.

Return value:
The function will return an unsigned int value of the archived time in fine scale clock ticks.

15.4.8 get_previous_shapevector

This function returns a pointer to the previously archived shapevector that is a specified distance from the current pointer. The argument given by `count` specifies the distance as a positive number. It is important to note that not all cycles may be archived. Thus, the time associated with the returned pointer may be earlier than the time associated with `count` cycles ago. Use the function `get_previous_time` (Sec. 15.4.7) to get the time associated with the returned shapevector.

Calling format:

```
float *get_previous_shapevector(struct rt_heap_misc *rtheap,int count)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.
- count: an integer giving the distance from the current vector pointer. Thus, a value of 1 would return a pointer to the previous shapevector.

Return value:
The function will return a float pointer to the beginning of the archived shapevector that is `count` steps away from the current shapevector.

15.4.9 get_previous_errorvector

This function returns a pointer to the previously archived errorvector that is a specified distance from the current pointer. The argument given by `count` specifies the distance as a positive number. It is important to note that not all cycles may be archived. Thus, the time associated with the returned pointer may be earlier than the time associated with `count` cycles ago. Use the function `get_previous_time` (Sec. 15.4.7) to get the time associated with the returned errorvector.

Calling format:

```
float *get_previous_errorvector(struct rt_heap_misc *rtheap,int count)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.
- count: an integer giving the distance from the current vector pointer. Thus, a value of 1 would return a pointer to the previous errorvector.

Return value:
The function will return a float pointer to the beginning of the archived errorvector that is `count` steps away from the current errorvector.

15.4.10 get_previous_pvector

This function returns a pointer to the previously archived pvector that is a specified distance from the current pointer. The argument given by `count` specifies the distance as a positive number. It is important to note that not all cycles may be archived. Thus, the time associated with the returned pointer may be earlier than the time associated with `count` cycles ago. Use the function `get_previous_time` (Sec. 15.4.7) to get the time associated with the returned pvector.

Calling format:

```
float *get_previous_pvector(struct rt_heap_misc *rtheap,int count)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.
- count: an integer giving the distance from the current vector pointer. Thus, a value of 1 would return a pointer to the previous pvector.

Return value:
The function will return a float pointer to the beginning of the archived pvector that is `count` steps away from the current pvector.

15.4.11 get_previous_fpcommand

This function returns a pointer to the previously archived fpcommand that is a specified distance from the current pointer. The argument given by `count` specifies the distance as a positive number. It is important to note that not all cycles may be archived. Thus, the time associated with the returned pointer may be earlier than the time associated with `count` cycles ago. Use the function `get_previous_time` (Sec. 15.4.7) to get the time associated with the returned fpcommand.

Calling format:

```
float *get_previous_fpcommand(struct rt_heap_misc *rtheap,int count)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.
- count: an integer giving the distance from the current vector pointer. Thus, a value of 1 would return a pointer to the previous fpcommand.

Return value:
The function will return a float pointer to the beginning of the archived fpcommand that is `count` steps away from the current fpcommand.

15.4.12 get_previous_intcommand

This function returns a pointer to the previously archived intcommand that is a specified distance from the current pointer. The argument given by `count` specifies the distance as a positive number. It is important to note that not all cycles may be archived. Thus, the time associated with the returned pointer may be earlier than the time associated with `count` cycles ago. Use the function `get_previous_time` (Sec. 15.4.7) to get the time associated with the returned intcommand.
Calling format:

```
int *get_previous_intcommand(struct rt_heap_misc *rtheap,int count)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.
- count: an integer giving the distance from the current vector pointer. Thus, a value of 1 would return a pointer to the previous intcommand.

Return value:
The function will return an int pointer to the beginning of the archived intcommand that is `count` steps away from the current intcommand.

15.4.13 get_previous_dmabuffer

This function returns a pointer to the previously archived dmabuffer that is a specified distance from the current pointer. The argument given by `count` specifies the distance as a positive number. It is important to note that not all cycles may be archived. Thus, the time associated with the returned pointer may be earlier than the time associated with `count` cycles ago. Use the function `get_previous_time` (Sec. 15.4.7) to get the time associated with the returned dmabuffer.
Calling format:

```
float *get_previous_dmabuffer(struct rt_heap_misc *rtheap,int count)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.
- count: an integer giving the distance from the current vector pointer. Thus, a value of 1 would return a pointer to the previous dmabuffer.

Return value:
The function will return a float pointer to the beginning of the archived dmabuffer that is `count` steps away from the current dmabuffer.

15.4.14 default_save_samples_setup

This function should be executed in the last function in the function vector (usually `save_samples`). It is the first of two functions to be called (the other being `default_save_samples` (Sec. 15.4.15)) to save a set of samples.

This function will decide whether to save away the current set of data samples from all the common vectors (e.g. shape, error, p, intcommand, fpcommand) and the raw data DMA buffer. If the decision is to save data, then `rtheap->new_dmabuffer` will be set to the address of the DMA buffer to be used next and `rtheap->nextsample` will be updated to the next time that data should be saved. If the decision is NOT to save samples, then `rtheap->new_dmabuffer` will be set to NULL. This value can be used by the installation code to save other data not in the common vectors.

There is a limited amount of storage space allocated to save data. The storage space allocated holds the number of samples to save plus two. The first extra space is the buffer used for the remainder of the shot after the data storage space has been filled. The second extra space was used by the older VME system and is no longer needed.

`rtheap->nextsample` holds the time (in units of fine scale clock ticks) to save the next sample of data. It is compared with `rtheap->currenttime` when deciding if it is time to save the samples. When the data storage space is full, the `rtheap->nextsample` location is loaded with the largest possible time value 0xffffffff.

The value in `rtheap->datasample_flag` is used to force the saving of data even if it is not time. Set this flag to the value 0xfffffffe to force the data to be saved. When there is no more space left, the time for the next sample is set to 0xffffffff, which will prevent `rtheap->datasample_flag` from triggering the saving of data.

Calling format:

```
void default_save_samples_setup(struct rt_heap_misc *rtheap)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.

15.4.15 default_save_samples

This function should be executed in the last function in the function vector (usually `save_samples`). It is the second of two functions to be called (the other being `default_save_samples_setup` (Sec. 15.4.14)) to save a set of samples.

This function first updates all the common "previous" vector pointers in `rtheap`, e.g., `rtheap->previous_errorvector`, to the values of the current vector pointers. Then it updates the current vector pointers if `rtheap->new_dmabuffer` is not NULL. It then saves away a set of fast data if it is time to do so by incrementing `rtheap->alldata`. Lastly, it resets `rtheap->datasample_flag` to 0.

Calling format:

```
void default_save_samples(struct rt_heap_misc *rtheap)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.

15.4.16 default_copy_combuffer

This function copies the communication buffer contents from the write-only vector `rtheap->adcombuffer` to the read-only vector `rtheap->adcombufloop`. This function should be called after all raw data has been acquired. The purpose of making this copy is to acquire the absolute latest content for use in the current cycle. If any time is spent waiting for a new set of data, the communication vector content might have changed. Call this function before any functions that need to use the communications buffer. Note that the phase sequence numbers, one per category, are at the beginning of the communications vector. These alone get copied in the function `default_save_samples` (Sec. 15.4.15). This function copies all elements.

Calling format:

```
void default_copy_combuffer(struct rt_heap_misc *rtheap)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.

15.4.17 default_set_pid_index

This function can be executed after `rtheap->currenttime` and `rtheap->previoustime` have been updated for a cycle. The function determines the offset into the pid lookup table based on the delta of the current and previous cycle times; note that if the delta is greater than the piddtmax, then set the delta to piddtmax because that is the last entry in the pidtables.

Calling format:

```
void default_set_pid_index(struct rt_heap_misc *rtheap)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.

15.4.18 read_timetostop_flag

This function returns the value of `rtheap->timetostop_flag`. This function is used in function `set_time_zero` (Sec. 15.3.3) for synchronization between all the real time processes. The default version is found in file `realmain.c` and should handle all cases.

Calling format:

```
void read_timetostop_flag(struct rt_heap_misc *rtheap,
     unsigned int *return_value)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.
- return_value: pointer to the unsigned int value to be returned.

15.4.19 read_ocpu_timetostop_flag

This function returns an int value from the given address. It is used in the function `rtcipc_read_timetostop_flag` (Sec. 15.2.12) to return the value of the timetostop_flag from another cpu. The default version is found in file `realmain.c` and should handle all cases.

Calling format:

```
void read_ocpu_timetostop_flag(unsigned int *return_value,int *address)
```

Function Arguments:

- return_value: pointer to the unsigned int value to be returned.
- address: pointer to the address to get the value from.

15.4.20 readticks

This function reads the number of clock ticks from the computer's processor. For linux there is a version of `readticks` in the file `common/read_clock_intel.c`. This function returns an unsigned int value which is the lower 32 bits of the 64-bit clock value.

Calling format:

```
int readticks(unsigned long long *ticks)
```

Function Arguments:

- ticks: returned number of clock ticks.

15.4.21 readticks64

This function reads the number of clock ticks from the computer's processor. For linux there is a version of `readticks64` in the file `common/read_clock_intel.c`. This function returns an unsigned long long value (64-bits).

Calling format:

```
int readticks64(unsigned long long *ticks)
```

Function Arguments:

- ticks: returned number of clock ticks.

15.4.22 get_cpu_speed

This function returns the clock speed in clock ticks per microsecond. The default function in file `realmain.c` reads the file `/proc/cpuinfo` and looks for the line that gives the processor speed.

Calling format:

```
unsigned int get_cpu_speed()
```

Return value:
This function returns the cpu speed in clock ticks per microsecond.

15.4.23 delay_awhile

This function delays the given amount of time. The input value is specified in microseconds. The default function in file `realmain.c` uses functions `readticks` and `get_cpu_speed`.
Calling format:

```
void delay_awhile(int delay)
```

Function Arguments:

- delay: the number of microseconds to delay.

15.4.24 lock_memory

This function locks the program and all its resources into memory so that system interrupts can be turned off. The default version is found in file `realmain.c` and should handle all cases.
Calling format:

```
int lock_memory()
```

Return value:
The function returns the status from the `mlockall` system function.

15.4.25 unlock_memory

This function unlocks the program and all its resources. The default version is found in file `realmain.c` and should handle all cases.
Calling format:

```
int lock_memory()
```

Return value:
The function returns the status from the `munlockall` system function.

15.4.26 get_rtoffset

This function takes a pointer address as input and returns the offset from the beginning of the memory region where that address is located. The default version is found in file `realmain.c` and should handle all cases.

Calling format:

```
int get_rtoffset(struct rt_heap_misc *rtheap, void *address)
```

Function Arguments:

- rtheap: pointer to the real time heap structure.
- address: pointer to the address whose memory region is desired.

Return value:
The function returns the offset from the beginning of the memory region to the address specified.

Chapter 16

Modifications

This section lists the dates of modifications to this document and references to the pages where changes were made. The reader can consult this section to determine where this document has been changed since it was last referenced.

1. 9/25/98:
   - Calling format descriptions for the host process initialization (4.6.2) and cleanup functions (4.6.2).
   - Some more details on the real time CPU algorithm description structure (4.6.3).
2. 5/4/99:
   - Sections 2.5.3 and 2.11 have been completely rewritten to describe the new block structured parameter data support and to label the unstructured parameter data method as obsolete. This documentation of the block structured parameter data support is not complete. There are still empty sections of the document. There may also still be some references outside of sections 2.5.3 and 2.11 to the old method of doing parameter data.
   - Section 2.14 is new. It describes the methods that the algorithm author can use to communicate error and status information to the PCS operator.
   - A section on obsolete features has been added (Sec. D).
3. 5/24/99:
   - Continuing to fill in the details about parameter data blocks by starting the description of how to write a user interface parameter data editing routine. So far, the description of `extract_paramdata` (Sec. 7.6.2) is all that has been finished.
   - Sections from the beginning through "PCS timing" were reviewed for gross inaccuracies and a few updates were made.

4. 7/29/2000:

   - The pre-shot initialization and post-shot cleanup functions that run on the real time processor were updated to have `struct rt_heap_misc *rtheap` as a second function argument. This makes the call format of these functions more consistent with the other initialization and cleanup functions. This also provides access to the complete memory layout on the real time processor which will make these functions more useful. See 4.6.3.

5. 8/15/2000:

   - The method to indicate the new phase sequence when making an asynchronous change in phase sequence has been changed. Instead of specifying the pointer to the descriptor of the new phase sequence, the "number" of the new sequence is specified. This is much easier because the phase sequence number is easily obtained using a predefined macro. See Sec. 2.15.1.
   - The description in Sec. 2.11.7 of how to access parameter data in real time has recently had a description of code that hadn't yet been implemented. The code is now up to date and the description in Sec. 2.11.7 is now accurate.

6. 11/24/2000

   - An easier method to define a descriptor for a data object has been implemented. (A descriptor is required when creating a parameter data block in order to define the format of the data within the block.) The format of a data object descriptor has also been changed so it is essential that the new method be used. The old method is now obsolete and not documented. The new method is described in Sec. 2.11.4.5, a section which existed previously but which has been completely rewritten.
   - A new data structure on the real time processor has been added: the pointer target vector. This vector should be used to hold values with C language data type `pointer`. This new vector replaces the previous practice of placing pointers in elements of the target vector. The pointer target vector was added because the target vector contains only values with data type requiring 4 bytes of storage and a pointer could require 8 bytes, depending on the processor architecture.

     The pointer vector and its usage are described in these locations: Sec. 2.5, Sec. 2.5.1, Sec. 2.7, Sec. 2.11.7, Sec. D.4.1, Sec. 14.1.5, Sec. 2.10.3, Sec. 4.5, Sec. 4.6.1.
   - Two sections (Sec. 2.5.5 and Sec. 2.10.10) were added discussing the pros and cons of global and static variables within the PCS code. In general these types of variables should not be used when writing application-specific code.

- Some general cleanup of the text in the first portion of the manual was done to try to make the descriptions more general. Some specific references to DIII-D were removed.

7. 11/28/2000:

   - Some hints on cleaning up resource usage after using `runsa` were inserted in Sec. 10.6.

8. 2/1/2001:

   - A data object descriptor entry of type `NESTEDTYPE` can only reference a data object which is a C language structure or `typedef` (4 and A).

9. 3/10/2001:

   - Section 4.7 was rewritten to include information on descriptors for both waveforms and static data items.
   - The string in the descriptor for a waveform that provides the Y axis label for a waveform plot can now include keywords that specify some special functions. This is discussed in Sec. 4.7 under the description of the `yunits` component of a data item descriptor.

     The first function implemented is the ability to specify a parameter data block that contains the labels to use for the grid levels of a gridded waveform. This allows a waveform that is used to select one of an array of data objects to have its labels updated dynamically to indicate the current set of available data objects. For instance, this would be useful with a waveform that selects one of an array of matrices. The standard matrix editor allows each matrix to be named. The matrix names would be used as labels on the waveform plot. Also, the allowable range of Y axis values can be constrained to match the number of available data objects.

10. 4/11/2001:

    - The alignment of a parameter data block is the larger of the normally required alignment for the data object in the block and the alignment specified by the argument to the function that creates the parameter data block (Sec. 2.11.4.6).

11. 7/3/2001:

    - The information in Sec. 13.4 about getting set up to read the S file content was updated to be consistent with the latest PCS release. The documentation about reading the data in the S file is still, however, far away from being complete.

12. 9/22/2001:

    - The concept of "global parameter data" has been removed from the PCS. Formerly, global parameter data was used to hold information about the PCS or tokamak setup that could be common to all algorithms. However, as implemented the global parameter data had a fixed structure that was inflexible and rather specific to the DIII-D tokamak.

      Global parameter data has been reimplemented using the more general purpose parameter data tools (Sec. 2.11). The data formerly held as global parameter data is now held in static data items (Sec. 2.10.1.2) in appropriate control algorithms (e.g. algorithms in the data acquisition and system categories).

      References to global parameter data were removed from sections 2.10.1, 2.10.5, 2.11.6.
    - The "external data item" has been added. Previously, it was possible to specify in an algorithm that when the global parameter data changed, some processed data in the algorithm needed to be recomputed. Since the former global parameter data is now ordinary data in a shot phase, it is necessary to allow an algorithm to specify that some of its processed data must be recomputed when raw data in a different shot phase is changed. This raw data in the different phase is an external data item. Each algorithm can have a list of these data items. See sections 2.10.2, 2.10.5, 2.10.6, 4.8, and 14.3.17.
    - An introduction was added to Sec. 2.10.
    - The single section on vector indices and data entry numbers was broken into 2 separate sections, Secs. 2.10.3 and 2.10.4.
    - Section 2.10.7 was added to describe how processed data is stored in the waveform server. The purpose of this section is primarily to provide background useful for understanding Sec. 2.10.8.
    - Section 2.10.8 was added to describe the general procedure for creating data for a target vector element from the vertices on one or more waveforms. This information is needed if none of the standard macros provided for the `alg_vectors` function (the macros are described in Sec. 14.1) are adequate and a customized function must be written.
    - The utility functions for creating the change list entries for a target vector element were documented in Sec. 14.2.
    - The `pcs_abort` function is documented in Sec. 14.6.38.
    - A new section was added that discusses facilities for "regulating" the PCS setup (Sec. 2.17). This section discusses how the algorithm author can ensure that the PCS operator chooses setup options that will correctly control a discharge.

Discussed are the access control mask (Sec. 2.17.2), the "no restore" capability (Sec. 2.17.3) and the procedure followed during a PCS restore was outlined (Sec. 2.17.3.1). Associated changes were also made in Secs. 2.11.4.1 (the parameter data initialization function), 14.6.5 (the `set_vertices` function) and 4.7 (the `access_control` field in the data item descriptor). Associated Sec. 14.6.39 (the `check_access_control` function) was added.

- Some improvements were made in the documentation for parameter data. Section 2.11.3 provides some step by step procedures for using parameter data blocks. Sections 2.10.1, 2.5.3 and 2.11.5 were rewritten. Some text was rewritten and reorganized into Sec. 2.11.1. Section 2.11.2 is new. All of the background information on static data items was consolidated into a single section (Sec. 2.10.1.2) which was relocated.
- More detail was added to Sec. 4.6.1.

13. 12/10/2001:

    - Section 2.17.4 was added to discuss the "parameter data block external file" facility. Section 2.17 was modified to provide a better introduction including the external file facility. Section D.8.27 was modified to include the mask for the external file bit.
    - Sec. 8 was added to discuss the procedure to build the PCS code (this section isn't complete).
    - Sec. 2.18 was added to describe the directories used by the PCS.
    - Sec. 9 describes the scripts used to start, stop and control the PCS. This section is also an introduction to the various ways to run the PCS for testing and normal tokamak operations.
    - A pdf version of this document is now available (see Sec.). Hopefully I will remember to keep it up to date since it isn't generated automatically. If the pdf file falls behind the html and ps versions please send email to ferron@fusion.gat.com.

14. 1/14/2002

    - Extended the discussion in Sec. 2.9 on using multiple real time processors in the PCS and removed DIII-D specific discussion. Added the detailed discussion of the interprocessor communication application programmer's interface (API) to Sec. 2.9.2 and made this part of Sec. 2.9.
    - The function `put_param_block_offsets` is described in Sec. 14.3.13. This function is a useful companion to the functions for interprocessor communication.
    - The communication vector was added to the lists in Sec.2.10.3.

- Section 15.2 was added. This section contains the calling formats of the interprocessor communication functions.

15. 2/24/2002

    - Expanded the discussion on building the PCS code. Corrected some errors in Sec. 8.1 on the first build. Added a section on the `make` command line options (Sec. 8.3).
    - Added a chapter on the simulation server (Sec. 12). This is a first attempt to describe the simulation server that will probably need some improvement at a later date.
    - Made small updates to the parameter data block creation description in Sec. 2.11.4.2 and the scratch parameter data block mask description in Sec. D.8.27.
    - Removed references to the function `put_param_block_scratch`. This function was obsolete. The description of how to create a scratch parameter data block was updated in Sec. 2.11.10.
    - The descriptions of the `put_param_*` functions were added to Sec. 14.3.

16. 2/25/2002

    - Fix the web addresses in Sec. .

17. 3/6/2002

    - Added a brief (and cryptic) section on how the simulation server determines its time step (Sec. 12.3.4). This section will certainly need improvement in the future.
    - In Sec. 4.6.1, the description of the structure element `parampro` incorrectly said that it should be `NULL`. This was corrected.

18. 9/25/2003

    - An appendix (Sec. C) was added that describes the facilities provided to allow restoring waveform vertices from a legacy archive format that is different from the standard PCS setup archive format. There were also a few small changes in Sec. 4.7 referencing this new appendix.
    - Section 4.12 was added to describe the algorithm master file section where the function prototypes for algorithm code that is compiled into the `host_cpu` process are placed.
    - Section 4.16 was added to provide background material on how the various sections of the algorithm master files are combined when the PCS code is built. This section should be helpful in understanding the role of each algorithm master file section and the context in which each master file section is compiled. This section isn't quite finished.

- In Sec. 4.13 and Sec. 4.15 a description was added of the usage of the `category_CPUx` macro to specify the code that should be compiled for a given `host_cpu` process or real time CPU executable when an algorithm executes code on multiple real time CPUs.
- Two sections of the algorithm master file don't yet have descriptions. Sections 4.2 and 4.4 were added with the text to be filled in later.
- In Sec. 2.11.4.2 added sections for `put_param_block_strings` D.8.8, and sections for `put_param_labeled_*` functions 14.3.9 to 14.3.12. Changed discussion of `put_param_block_chars` D.8.2.
- In Sec. 2.12 added mention of standard static data types with references to the `put_param_labeled_*` functions.
- Added new field `all_info` to `category_usage` 4.6.1 and mentioned the new field in the descriptions of the different vector element arrays: target, shape, etc.
- Redid `WAVEGLOBALS` discussion 4.5 in order to describe the `all_info` field. Moved old discussion to a section under `Obsolete features` D.6.
- Added new fields to the `HOST_CONTROL` structure described in Sec. 4.6.2.
- Added `NAME_CHANGES` Sec. 4.9.

19. 11/24/2003
    - Some small editing changes in sections 4.9, 4.6.2.
    - Description of the algorithm-specific post-shot calculation and archiving functions was added to Sec. 2.6.
20. 12/04/2003
    - Modified or added sections 8.2 to 8.5.
    - Modified section 13.3.
21. 6/24/2004
    - Added information about additional category index macros in Sec. 2.16.1 and about macros for algorithm indices in Sec. 2.16.4.
    - Added information on obtaining pointers to parameter data blocks for an arbitrary phase of any category in Sec. 2.11.7.1.
22. 10/3/2005 (JRF)
    - Some wording modifications in Sec. 4.5 and related parts of Sec. 4.6.1.
23. 6/30/2006 (RDJ)

- Filled in sections for parameter data routines Sec. 14.3.14 through Sec. 14.3.34.

24. 7/13/2006 (RDJ)
    - Added section for function `is_phase_used` Sec. 14.6.9.
25. 9/29/2006 (RDJ)
    - Updated examples 1-3 in use of parameter data GEN_ macros to emphasize that the GEN_ macro should only be used in the CONTROLDEF section for `typedef` and structure definitions. Sec. 2.11.4.5.
26. 4/02/2007 (RDJ)
    - Added section for the realmain.c infrastructure file. Sec. 4.16.7.
27. 4/02/2007 (RDJ)
    - Added sections for other rtcipc functions. Sec. 15.2.11. Sec. 15.2.12. Sec. 15.2.13. Sec. 15.2.14.
28. 4/04/2007 (RDJ)
    - Corrected paragraphs concerning the alg_beginphase and alg_enterphase functions. Old text mentioned that the continuous portion of the target vector had not been calculated when these functions execute. This is no longer the case. The continuous portion of the target vector is now correct when these functions execute. Sec. 4.6.3 and 4.6.3.
29. 4/04/2007 (RDJ)
    - Added sections for the generic_editor and matrix_editor. Sec. 2.11.8.13 Sec. 2.11.8.14.
30. 4/25/2007 (RDJ)
    - Updated section about the data structures used in real time Sec. 2.5.
    - Modified section about allocating space in the real time data structures Sec. 2.5.6.
    - Added section about memory regions Sec. 13.2.
    - Added section on function set_rt_memory_region Sec. 14.6.1.
    - Added section on function fetch_external_vertices Sec. 14.2.2.
    - Moved section D.7 to Obsolete features.
31. 5/18/2007 (RDJ)
    - Updated section on support for PID calculations Sec. 5.

- Added sections on pidv4 Sec. 5.0.1, pidv3 Sec. 5.0.2, and the older versions of the pid function Sec. 5.0.3.
- Added sections on periodic actions and raw data actions Sec. 6.
- Added section describing the view log window Sec. 7.1.
- Modified section on log messages (Sec. 2.14.2).
- Modified section on msg_log (Sec. 14.5.2) and added section describing rtmsg_log (Sec. 15.1.1).
- Added sections describing rtprintf_init Sec. 15.1.2.1, rtprintf Sec. 15.1.2, and rtprintf_end Sec. 15.1.2.2.

32. 6/15/2007 (RDJ)
    - Added section for function wvmp Sec. 14.2.9.
    - Added section for function wv2mp Sec. 14.2.10.
    - Added section for function wv2mp Sec. 14.2.11.
    - Added section for function wv_offset Sec. 14.2.14.
33. 08/21/2007 (RDJ)
    - Changed references of host_realtime to host_cpu and host_realtime_cpu to host_cpu.
    - Modified slightly the discussion of access control in Sec. 2.17.
    - Modified slightly the discussion of external data files Sec. 2.17.4.
    - Added discussion of DATA_DIR in Sec. 2.18.
34. 08/31/2007 (RDJ)
    - Added section describing how to use parameter data blocks to specify Y grid labels, Y axis labels, Y minimum and maximum values, etc. Sec. 2.13.
35. 09/05/2007 (RDJ)
    - Added sections describing the real time message facility Sec. 2.9.4.
36. 10/10/2007 (RDJ)
    - Added sections describing the newer putblock* functions and getblock* functions Sec. 2.11.4.2.
    - Changed many sections about parameter data to handle the newer putblock* functions and getblock* functions Sec. 2.11.
    - Added discussion about shared parameter data blocks Sec. 2.11.4.8.

- Updated section on backward compatibility for parameter data blocks Sec. 2.11.5.3.
- Moved sections describing put_param_block* functions and get_param_block* functions to the obsolete section Sec. D.8.14.

37. 10/16/2007 (RDJ)

    - Added sections describing functions badstatus Sec. 14.6.48, ckstatus Sec. 14.6.49, create_start_time_list Sec. 14.6.40, and get_first_start_time Sec. 14.6.41.

38. 10/25/2007 (RDJ)

    - Added sections describing functions set_rt_memory_preference Sec. 14.6.2 and set_cache_preference Sec. 14.6.4.
    - Reworked sections describing the communications functions Sec. 2.9.2 and the real time message facility Sec. 2.9.4.

39. 09/03/2008 (RDJ)

    - Added sections describing macros case_wvdp Sec. 14.1.6 and case_fst_wvdp Sec. 14.1.7.
    - Moved section case_phsseq_ptr Sec. D.2 to obsolete features.

40. 09/04/2008 (RDJ)

    - Added sections describing functions that handle strings: swap_i2 Sec. 14.6.23, swap_i4 Sec. 14.6.24, swap_i8 Sec. 14.6.25, str2lower Sec. 14.6.26, str2upper Sec. 14.6.27, str_trim Sec. 14.6.28, str_cmp Sec. 14.6.29, str_cmpi Sec. 14.6.30, str_find Sec. 14.6.31, str_find_ignore Sec. 14.6.32, str_nitems Sec. 14.6.33, strlowcase Sec. 14.6.34, strupcase Sec. 14.6.35, strpos Sec. 14.6.36, and strreplace Sec. 14.6.37.
    - Added section describing function set_phsseq_vertices Sec. 14.6.6.
    - Added sections describing functions that return the index into a vector and the physical cpu where the vector is located given a pointname: search_target_maps Sec. 14.6.42, search_shape_maps Sec. 14.6.43, search_error_maps Sec. 14.6.44, search_p_maps Sec. 14.6.45, search_command_maps Sec. 14.6.46, and search_fpcom_maps Sec. 14.6.47.
    - Added section describing function make_pidtau Sec. 14.2.13.
    - Added sections describing functions pcs_delete_archive_blocks Sec. 14.3.39, locate_phase_byname Sec. 14.6.16, mult_4 Sec. 14.6.19, mult_8 Sec. 14.6.20, mynint Sec. 14.6.21, and mynuint Sec. 14.6.22.
    - Added sections describing functions get_nocomment_line Sec. 14.6.13, get_line_with_nocomments Sec. 14.6.14, get_param_numblocks Sec. 14.3.35, get_rt_flag_from_cpu Sec. 14.3.37 and get_host_flag_from_cpu Sec. 14.3.38.

- Added sections describing function get_used_phase_list Sec. 14.6.10, find_phase_sequence Sec. 14.6.7, locate_phase_sequence Sec. 14.6.8, locate_string_in_labels Sec. 14.6.11, fill_dma_region_constants Sec. 14.6.12, and get_algorithm_identifier_by_phase Sec. 14.6.15.

41. 09/05/2008 (RDJ)
    - Added section describing function logit Sec. 14.5.3.
    - Added sections describing functions get_virtual_cpu_num Sec. 14.6.17 and get_physical_cpu_num Sec. 14.6.18.
    - Added section describing function get_cpu_from_name Sec. 14.3.36.
    - Added section describing function wvsignal Sec. 14.2.12.
42. 09/12/2008 (RDJ)
    - Added sections describing required real time functions. Sec. 15.3 through Sec. 15.3.7.
    - Added sections describing optional real time functions. Sec. 15.4 through Sec. 15.4.26.
    - Redid the first few paragraphs of Sec. 2.7 to bring it up to date.
    - Changed many instances of ”computer” to ”processor” to reflect the multiple processors now found in computers. it up to date.
43. 09/17/2008 (RDJ)
    - Added sections describing the waveform editor Sec. 7.5, the menu options Sec. 7.5.1, the restoring of data, Sec. 7.5.3, the checkboxes on the waveform editor Sec. 7.5.4 and Sec. 7.5.4, the buttons on the waveform editor Sec. 7.5.5, the plot region Sec. 7.5.7, and using the mouse in the waveform editor Sec. 7.5.8.
44. 09/29/2008 (RDJ)
    - Added sections describing the restore waveform editor Sec. 7.5.3.1 through Sec. 7.5.3.3.
    - Added section describing the commands menu Sec. 7.5.1.9.
    - Added sections describing the information menu choices Sec. 7.5.1.10.
45. 10/02/2008 (RDJ)
    - Added section describing waveform groups Sec. 7.5.2.
    - Added section describing the vertex region of the waveform editor Sec. 7.5.6.
46. 09/09/2009 (RDJ)
    - Added sections describing realtime functions that return a previously archived time or a pointer to an archived vector. Sec. 15.4.7 to Sec. 15.4.13.

- Added sections describing the functions that allow for off-loading of data during a long discharge Sec. 14.6.3 and Sec. 15.4.6.

Appendix A

Data object descriptors

This section contains various details about the use of data object descriptors in the PCS that were not discussed in Sec. 2.11.4.5.

A data object descriptor is an array of type `STRUCT_DESCRIPTORS`. This data type is defined as follows.

```
struct data_descriptor {
    int type;
    int size;
    struct *data_descriptor next;
};
typedef struct data_descriptor STRUCT_DESCRIPTORS;
```

Each element of the array, referred to here as a "descriptor entry," describes a component of the data object. Each component of the data object is an array of values of a specified data type. The `type` indicates the type of variable in the data object component and `size` gives the number of elements in the array of this type.

The elements of `STRUCT_DESCRIPTORS` are as follows.

- `type`: Specifies the variable type. It is one of the following values: `CHARTYPE`, `SHORTTYPE`, `INTTYPE`, `FLOATTYPE`, `LONGTYPE`, `DOUBLETYPE`, `POINTERTYPE`, `STRUCTURETYPE`, `UNIONTYPE`, `NESTEDTYPE`. The variable type can also have the following values, which do not indicate a particular type of variable but instead are used for managing the descriptor: `STRUCTUREEND`, `UNIONEND`, `ENDOFDESCRIPTOR`, `UNKNOWNTYPE`.
- `size`: the count of values with variable type `type`. Thus, a descriptor entry can represent an array of values of a single type.
- `next`: This value is meaningful only if `type` is `NESTEDTYPE`. In this case, `next` is a pointer to another data object descriptor. The meaning is that a data object is contained within another data object and that a separate description is available for the inner data object. If `type` is not `NESTEDTYPE`, the value of `next` should be `NULL`.

Every complete descriptor (i.e. an array of type `STRUCT_DESCRIPTORS`), has in the last entry the type `ENDOFDESCRIPTOR` to indicate the end of the descriptor. Every descriptor entry with type `STRUCTURETYPE` has a matching entry with type `STRUCTUREEND` to indicate the end of the structure. Every descriptor entry with type `UNIONTYPE` has a matching entry with type `UNIONEND` to indicate the end of the union.

C language unions cannot actually be used in a parameter data block. This is because a union can be defined to contain one of several possible types of data, and there isn't any way to know which data type is actually present in the parameter data block. So, `UNIONTYPE` and `UNIONEND` will not be used for parameter data block descriptors. These two macros are used by descriptors that are part of the PCS infrastructure code.

In an entry with type `STRUCTURETYPE`, the `size` value indicates the number of elements in an array of structures. In every descriptor entry that falls between a `STRUCTURETYPE`, `STRUCTUREEND` pair, the `size` value indicates size of an array that is contained within the structure.

A structure can be explicitly defined within another structure without using a nested descriptor. Here is an example.

```
#define GEN_phase_changelist \
SDSTART(      struct phase_changelist {             ,D_phase_changelist)\
DGEN   (        unsigned int time;                  ,INTTYPE,1         )\
DGEN   (        int data_structure;                 ,INTTYPE,1         )\
DGEN   (        int element;                        ,INTTYPE,1         )\
DGEN   (        char *element_name;                 ,POINTERTYPE,1     )\
DGEN   (        struct {                            ,STRUCTURETYPE,1   )\
DGEN   (        int intdata[MAX_CHANGE_DATA];       ,INTTYPE,MAX_CHANGE_DATA)\
DGEN   (        float floatdata[MAX_CHANGE_DATA];  ,FLOATTYPE,MAX_CHANGE_DATA)\
DGEN   (               } data;                      ,STRUCTUREEND,0    )\
DGEN   (        struct phase_changelist *next_change;,POINTERTYPE,1     )\
SDEND  (                              };                               )
GEN_phase_changelist
```

There is no limit to the number of levels of nesting of structures within structures.

Sometimes errors in the coding of the definition for a data object descriptor prevent the code from compiling and the errors, although often simple such as leaving spaces after the backslash at the end of the line, are not easy to find. For this reason, the routine `size_list_install.c` was written. It is easy to add code to this routine that will print a summary of a data object. Because `size_list_install.c` is a relatively simple routine, it compiles quickly and it is easy to use the compiler options to view the output from the C language preprocessor (e.g. `cc -P`). By viewing the preprocessor output the mistakes coding the descriptor definition macro are easy to find.

The PCS infrastructure code uses data object descriptors primarily in conjunction with the routines in the infrastructure file `convert_functions.c`. The routine `convert_size` will calculate the size of a data object on any processor. The routine `convert_data` converts the data format from that of one processor to another. Consult the comments in `convert_functions.c` for information on how to use these functions.

A `GEN_` macro to generate a descriptor can behave in one of 3 ways, depending on the current set of definitions for the macros `DSTART`, `DGEN`, `DEND`, `SDSTART`, `SDGEN`, and `NDGEN`. The definitions for these macros are changed by including lines that look like the following.

```
#define DESCRIPTORS_PART_*
#include "descriptors.h"
#undef DESCRIPTORS_PART_*
```

It shouldn't be necessary to include this code in an algorithm master file because the files into which the master file code is included already contain the correct instances of this code.

Here, the * character indicates one of the following: "1", "2", or "3".

1. `DESCRIPTORS_PART_1`: the `GEN_` macro will generate the code to define the data object to the compiler (e.g. the code will be a structure definition if the data object is a structure).

2. `DESCRIPTORS_PART_2`: the `GEN_` macro will generate the code to initialize a variable with the `D_` prefix that contains the actual data object descriptor.

3. `DESCRIPTORS_PART_3`:the `GEN_` macro will print a summary of the data object descriptor. This option is used in `size_list_install.c`.

Some additional infrastructure functions for dealing with data object descriptors are in the file `descriptors.c`. These functions are primarily used by the parameter data handling routines. The function `descriptor_copy` returns a copy of a descriptor with any entries of type `NESTEDTYPE` replaced by the descriptor referenced by the pointer in the `NESTEDTYPE` entry. The function `descriptor_unrolled_copy` returns a descriptor with any entries which describe an array of structures unrolled to include only the actual components of the structure without any indication of beginning or end of the structure. The descriptor array returned by this function has a unique format that includes the offset into the data object of each piece of data indicated by a descriptor entry.

A descriptor entry of type `NESTEDTYPE` has a `size` value indicating how many objects of that type are present (i.e. the descriptor entry can indicate an array of values of the type described by the descriptor referenced by `next`). When the nested descriptor is substituted for by `descriptor_copy`, the `size` value must be used as a multiplier for the `size` values in the descriptor copy. In addition, the required alignment for the nested object must be handled properly, particularly if padding is required at the end of the data object in order to maintain proper alignment in an array of data objects of the type described by the nested descriptor. These requirements are not handled in `descriptor_copy` and `descriptor_unrolled_copy` for any arbitrary data object. This leads to the requirement mentioned in Sec. 2.11.4.5 that a nested descriptor can only be for a C language structure or `typedef` because these are the only types of data objects handled properly in this context. There is really no loss here because referencing these 2 types of data object is the primary purpose of the nested object feature.

Appendix B

Specifying the target buffer location for communication

This section discusses the way to specify the location to which data should be copied using the interprocessor communication API. The basic requirements are to specify the CPU to which data should be copied and the memory location on that CPU.

The target CPU is identified by using a macro that identifies the physical CPU that corresponds to a particular virtual CPU of a particular category. These macros are described in Sec. 2.16.3. For instance, the algorithm code might want to copy data to virtual CPU C of the category with the identifier `example`. In this case the target CPU would be identified by specifying the macro `CPUC_EXAMPLE`. This macro is automatically defined during the PCS build process to be equal to the equivalent physical CPU.

The easiest way to transfer data to another CPU is to write into its communication vector. It is easy to provide space in the communication vector to receive the data as part of the PCS configuration and the interprocessor communication API provides several functions that are used for writing into the communication vector on another CPU. These functions require as an input argument the number of the element of the communication vector that is at the start of the portion of the vector to be written. This value is easy to determine using the vector element number macros (Sec. 2.10.3) that are defined as part of the `CONTROLDEF` section (Sec. 4.3) of the algorithm code.

The most generic function in the interprocessor communication API writes to an arbitrary memory location. This function, `rtcipc_write`, requires an argument that specifies the location on the target CPU where the data are to be written. This is specified as a byte offset from the beginning of the PCS memory heap (see the background discussion in Sec. 2.9.2).

There are two ways to determine the the appropriate byte offset.

1. On each real time CPU, the value `void *rtheap->rtmemory` is a pointer to the beginning of the PCS memory heap on that processor. So, the offset of any arbitrary location from `rtheap->rtmemory` can easily be determined on that processor. Then, the offset value on this first processor can be written into the communication vector

on other processors. The other processors can then use this information to write data into the PCS memory heap on the first processor. This method requires some careful handshaking between the code on the various processors to make certain that the correct offset values are available when they are needed.

2. A scratch parameter data block is the most reasonable target buffer for a write of data from one CPU to another. The parameter data block is easy to create and, if necessary, a special byte alignment can be specified. There is an easy way to determine the offset of a parameter data block from the beginning of the PCS memory heap. The function `putblock_offsets` (Sec. 14.3.13) can be used in a control algorithm to create a parameter data block that contains a list of the offsets to each of the parameter data blocks on a specific virtual CPU being used by the algorithm. This parameter data block can be designated to be copied to any virtual CPU where it is needed by the algorithm. So, if an algorithm executing on multiple CPUs needs to use the interprocessor communication API, the required offset to the target buffer can be easily determined by looking it up in a parameter data block on the local CPU. The details of the function call are described in Sec. 14.3.13.

Appendix C

Restoring waveform vertices from legacy archives

The PCS software described in this document might be installed for use to replace or complement another control system implementation. For instance, a digital PCS is sometimes installed for use in a facility where an analog control system has already been in use. Typically, the first usage of the digital PCS would be to emulate the control behavior of the analog control system.

The concept of a "waveform" is probably common to both systems. In an analog control system, the waveform might be used to specify the time evolution of the output of an arbitrary waveform generator. The vertices for the analog control system waveforms would probably have been archived in the tokamak database. The digital PCS has its own format for archiving the setup data. It is desirable to be able to restore into the digital PCS the vertices of waveforms archived in the database for the analog control system. This section describes a simple facility that is provided to allow implementation of this capability.

In the data item descriptor (described in Sec. 4.7) there are several entries available that are not used in any specific way by the PCS infrastructure. So, these entries are available to provide the capability to restore from another waveform archive format.

- `restorepoint`: This entry is a string that can be used to specify the database entry in the legacy archive from which vertices for the waveform should be restored.
- `restorescale`: This entry is a floating point value by which the legacy waveform vertices can be multiplied before storing the vertices for a waveform in the digital PCS. This scale factor can be used to account for any desired changes in units.
- `restoreoffset`: This entry is a floating point value which can be added to the legacy waveform vertices before storing the vertices for a waveform in the digital PCS. This offset value can be used to account for any desired change in the waveform baseline.

- `archivepoint`: In early versions of the digital PCS software as used at DIII-D, the waveform vertices were archived in the legacy format used by the original analog control system. This string was used to specify where the waveform vertices were to be archived. Since the digital PCS now has its own method of archiving the PCS setup in the tokamak database, this entry doesn't really have a designated purpose. So, it can be used for any installation-specific purpose.

The code to restore waveform vertices from a legacy archive location must be located in the user interface in an installation-specific IDL routine. This routine can be invoked by the PCS user through a menu item on the "restore" menu. The restore menu can have installation-specific items added. The place where this is done is `waveinstall.pro`. See the example and comments in that file.

The installation-specific code would typically create a user interface window that would allow the user to restore vertices into the waveform presently selected on the waveform editor or, for instance, restore vertices into all waveforms in the presently selected shot phase. The restoration code must fetch the vertices from the legacy archive location and then send the vertices to the waveform server. The fields in the data item descriptor described above are available in the user interface so that they can be used to specify locations in the legacy archive database.

Example code that implements this type of restore of waveform vertices from a legacy archive is in the file `rstrwave.pro`. This is a reasonably straightforward routine that can be understood by reading the comments in this file. The code for the legacy restore function is included in the user interface in the same way that code is included for custom static data item editors (see the description in Sec. 2.11.8.12).

Appendix D

Obsolete features

As the PCS software evolves, some features have been changed to be more powerful, more generic, or easier to use. However, some code still exists that was created using older features with which the new features are still compatible. This chapter documents these obsolete features so that the existing code using old methods can be understood.

D.1 Unstructured parameter data

The original method of structuring parameter data was to simply allow an algorithm to have a single block of bytes that it could have organized in any way desired by the algorithm author. In other words, the parameter data block had no specific structure. This facility allowed for unanticipated needs but was quite difficult to program because the algorithm author needed to handle everything concerning writing the data block and subsequently locating the data in the block when it was needed later.

This unstructured parameter data organization has been replaced with block structured parameter data which is much more powerful, generic and much easier to use.

D.1.1 In the waveform server

An unstructured parameter data block is pointed to directly by `phase->parameters` and has the following structure.

- `int length`: Equal to the number of bytes of data not counting this integer.
- `struct stuff {}`: A structure, or some other data organization, containing the parameter data. There are `length` bytes.

Any routine that will replace the parameter data block must `malloc` the memory required for the new parameter data, use `free()` to deallocate the parameter data memory already in use (if `phase->parameters` is not `NULL`), and copy the pointer to the new parameter data block into `phase->parameters`.

D.1.2 For the real time processor

There must be a routine that returns the parameter data to be loaded into the real time computer's memory. This routine is called during setup for the shot when the host code for the real time computer is filling the memory of the real time computer.

Generally, this routine copies data from the block stored in the waveform server address space. However, the data provided can be formatted differently than the way it is stored in `phase->parameters` and the data used by the real time computer might be part or all of the data at `phase->parameters`. The data format is completely determined by the algorithm author.

The calling format is:

```
\ntt{void alg\_forhost(struct shotphase *phase, int vcpu,
                  char **data, int *length)}
```

Here the arguments are as follows.

- `char **data`: a pointer to the location to store the pointer to the data.
- `int *length`: a pointer to a location to store the number of bytes in the block of parameter data.
- `struct shotphase *phase`: pointer to the descriptor of the shot phase for which the data is being stored in the rtcpu's memory.
- `int vcpu`: the number of the virtual cpu into which the data is to be loaded. There could be different data for each cpu used by a given algorithm. Recall that virtual cpus are `CPUA`, `CPUB`, `CPUC`, etc.

Procedure for this function:

1. Determine the size and store it.

   ```
   *length = size;
   ```

2. Allocate the memory.

   ```
   mem = (char *)malloc(*length);
   *data = mem;
   ```

3. Copy the appropriate data to the allocated memory space.

The calling routine will deallocate the memory when it is done with it.

D.1.3 For the user interface

The algorithm author can give the PCS operator the option of modifying the parameter data. For this purpose a custom user interface using the X windows capability of IDL must be written. This code can request from the waveform server the current values of the parameter data. The user then makes any required changes and then the IDL code sends the data back to the waveform server.

The routine to provide the data for the user interface is:

```
\ntt{void alg\_foruser(struct shotphase *phase,
                  char **data, int *length)}
```

The arguments and outline of the procedure for this routine are the same as the "forhost" routine, except that the virtual cpu is not an input argument.

D.1.4 From the user interface

This routine receives the parameter data when the user interface sends it back and stores the data away in the region pointed to by `phase->parameters`.

The calling format is:

```
\ntt{void alg\_fromuser(struct shotphase *phase, char *data, int length)}
```

The function arguments are:

- `phase`: pointer to the shot phase descriptor
- `data`: pointer to the input data.
- `length`: number of bytes of data provided at the location pointed to by `data`.

The steps in this procedure are:

1. Optionally: free the memory allocated to parameter data if necessary.

   ```
   if(phase->parameters != NULL)
      free(phase->parameters);
   ```

2. If necessary, allocate the correct amount of memory for the new parameter data and store the address in `phase->parameters`.

3. Copy the data from the `data` input area to the `phase->parameters` area.

The calling routine deletes the memory allocated for the input data.

D.1.5 Data to be archived

The parameter data are archived in the "variable header" of the pointname "PCSSETUP". Because of the rules for writing to a DIII-D pointname, the data must be divided up into arrays of identical type: char, short, int, float, double.

So, this routine copies the data to be archived from the `phase->parameters` area into arrays of the correct type. Note that all, part, or none of the parameter data could be archived. This is determined by the algorithm author.

Calling format:

```
\ntt{void alg\_archive(struct shotphase *phase,\\
                       char **asciidata,    int *asciicount,\\
                       short **shortdata,   int *shortcount,\\
                       int **intdata,       int *intcount,\\
                       float **floatdata,   int *floatcount,\\
                       double **doubledata, int *doublecount)}
```

The function arguments are:

- `asciidata`, `shortdata`, `intdata`, `floatdata`, `doubledata`: pointers to locations to store the pointers to the data that is returned for archiving. Each points to an array of the given type. If there is no data of a given type, return the pointer equal to NULL.
- `asciicount`, `shortcount`, `intcount`, `floatcount`, `doublecount`: Pointers to locations to return the number of elements in each of the data arrays (not the number of bytes). If there is no data of a given type return the count equal to zero.

Steps in this procedure:

1. Determine the counts for each data type and write them into the correct return variables.
2. Allocate memory for each of the return data arrays using `malloc` and write the pointers into the return variables.
3. Copy the data to be archived into the correct array. e.g. All integers into intdata and all floats into floatdata.

The calling routine frees the memory that was allocated.

D.1.6 Restoring data

The PCS provides the parameter data being restored in the exact same format as it was provided to the PCS by the `alg_archive` routine. The algorithm author simply reverses the archive process in this routine: the data is provided as input arguments and the `alg_restore` routine copies it into `phase->parameters`.

Calling format:

```
\ntt{void alg\_restore(struct shotphase *phase,\\
                       char *asciidata, int asciicount,\\
                       short *shortdata, int shortcount,\\
                       int *intdata, int intcount,\\
                       float *floatdata, int floatcount,\\
                       double *doubledata, int doublecount)}
```

The function arguments are:

- `phase`: pointer to the descriptor for the shot phase.
- `asciidata`, `shortdata`, `intdata`, `floatdata`, `doubledata`: the arrays of input data.
- `asciicount`, `shortcount`, `intcount`, `floatcount`, `doublecount`: the counts of values of each data type

The calling routine takes care of any necessary freeing of memory.

D.1.7 Parameter data size routine

This routine returns the number of bytes that will be required on the specified real time computer for storage of the parameter data for the specified shot phase.

Calling format:

```
\ntt{int alg\_paramsize(struct shotphase *phase, int vcpu)}
```

The arguments are:

- `phase`: pointer to the descriptor of the shot phase.
- `vcpu`: number of the virtual real time cpu for which the data byte count is required.

Outline of the procedure:

1. Determine the count of bytes that will be returned in the same way as in `alg_forhost` (in fact, `alg_forhost` should actually call this routine to get the size of the parameter data area).
2. Return the count.

D.1.8 Accessing parameter data in real time

The algorithm author can arrange for a pointer to a parameter data block to be placed into an element of the integer step portion of the target vector. In real time this value can be accessed and used as a pointer. This is the preferred method for locating a parameter data block in real time because it is the fastest method.

Optionally, this real time pointer can point to a location at a specified offset within the parameter data block. This can be coupled to a waveform so, for instance, a user can specify which matrix in an array of matrices should be used as a function of time.

In the `alg_vectors` function, the macro to use is (Sec. D.1.10):

```
\ntt{case\_ptr\_param\_wvmp(int data\_entry\_number,
         int target\_vector\_index,
         int virtual\_cpu, char *waveform\_name,
         float offset, float scale\_factor)}
```

This generates values which will be interpreted as offsets into the block of parameter data. During shot setup the address of the parameter data is added to these values so they become pointers into the parameter data.

To specify a particular parameter data block, for the `offset` value specify the following function (Sec. D.1.9).

```
(float)pcs_paramoffset(phase,virtual_cpu, block_name)
```

This function generates the offset into the block of bytes used for storage of parameter data on the virtual CPU specified by `virtual_cpu` of the parameter data block specified by `block_name`.

The integer step target vector element specified by `target_vector_index` is calculated from the waveform given by `waveform_name` by multiplying the waveform vertices by `scale_factor` and then adding `offset`. So, for the example of the array of matrices, added to the value of `offset` needed to access the beginning of the parameter data block would be the offset in bytes into the parameter data block of the first matrix (normally 0), and `scale_factor` would be the size in bytes of a matrix. The waveform would have values 0, 1, 2 etc. to indicate that the first, second, third etc. matrix should be used.

If `waveform_name` is a null pointer then no waveform is used. Instead, a single integer step target vector value assigned to $t = 0$ (beginning of the shot phase) is generated pointing to the byte `offset` bytes into the parameter data. In this case `scale_factor` is ignored.

D.1.9 pcs_paramoffset

This (obsolete) function is intended for use in the `alg_vectors` routine for a control algorithm, executing within the waveform server process. It returns the offset in bytes into the region of memory on the real time computer that will be used to hold all of the parameter data for a particular phase, of the location where the data for the specified parameter block will be stored.

Function definition:

```
int pcs_paramoffset(struct shotphase *phase,
                    int vcpu, char *name)
```

Function arguments:

- `phase`: pointer to the descriptor for the shot phase.
- `vcpu`: the index of the virtual real time CPU for the category indicated in `phase` where the parameter data block will be located. Use the macros of the form `CPUx` where `x` is a letter `A`, `B`, etc.
- `name`: the name of the parameter data block.

D.1.10 case_ptr_param_wvmp

Calling format:

`case_ptr_param_wvmp(THE_DE,THE_T,CPU,WAVEFORM,OFFSET,FACTOR)`

Macro arguments:

- `THE_DE`: data entry index.
- `THE_T`: target vector index.
- `CPU`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.).
- `WAVEFORM`: string giving the waveform name. Must be used by the algorithm used in `phase`.
- `OFFSET`: the value to add to the waveform vertices.
- `FACTOR`: the value to scale the waveform vertices.

This macro causes integer data to be generated by multiplying the waveform by `FACTOR` and then adding `OFFSET`. The integer data is interpreted as an offset into the block of parameter data belonging to the shot phase specified by `phase`. The integer step target vector element given by `THE_T` on the virtual cpu given by `CPU` is set equal to a pointer to the data at this offset from the beginning of the parameter data. So, the integer step target vector element can be used to point to a specific portion of the parameter data and the portion of data that is pointed to can change in time.

If `WAVEFORM` is a null pointer, a waveform is not used and `FACTOR` is ignored. Instead, a single value assigned to $t = 0$ (relative to the start of the shot phase) is generated with integer value equal to `OFFSET`. This can be used to generate a constant (in time) pointer into the parameter data.

The pointer to the parameter data should be accessed as in the following example (here the block of parameter data to be accessed is assumed to be an array of floats).

```
float *mydata;
int *istargets;

istargets = (int *)rtheap->adtarget;
mydata = (float *)istargets[THE_T - 1];
```

D.2 case_phsseq_ptr

Calling format:

```
\ntt{case\_phsseq\_ptr(THE\_DE,THE\_T,CATIDENT,CATVCPU,WAVEFORM,CPU)}
```

Macro arguments:

- `THE_DE`: data entry index.
- `THE_T`: target vector index.
- `CPU`: virtual cpu number of the category given by `phase` on which the target vector is located.
- `WAVEFORM`: string giving the waveform name. Must be used by the algorithm used in `phase`. Each waveform vertex Y value must be equal to the identifying "number" of a phase sequence of the category given by `CATIDENT`.
- `CATIDENT`: the identifier of the category for which the waveform gives phase sequence numbers.
- `CATVCPU`: the virtual cpu index of the virtual cpu of the category CATIDENT on which the descriptor with the address to be calculated is located.

This macro causes entries in the integer step portion of the target vector to be generated. This macro takes the vertices of the waveform and converts each one into the address of the phase sequence descriptor with "phase sequence number" given by the Y value of the vertex. The phase sequence is one associated with the category given by `CATIDENT`. The address is the address of the descriptor in the memory of the virtual cpu given by `CATVCPU`. The address is placed in the integer step portion of the target vector (as an integer) in the element `THE_T` on the virtual cpu `CPU` of the category given by `phase`. If the waveform vertex is zero the descriptor pointer will be a null pointer to indicate a return to the previous phase sequence.

The pointer to the phase sequence descriptor should be accessed as in the following example.

```
int *istarget;
struct phs_tmg_seq *seq;

istarget = (int *)rtheap->adtarget;
seq = (struct phs_tmg_seq *)ist[THE_T - 1];
```

D.3 get_rt_param_block

A routine for use on one of the real time processors. This function returns a pointer to a particular parameter data block of a shot phase. The block is identified by its block order index value (Sec. 2.11.4.7).

Function definition:

```
void *get_rt_param_block(void *paramdata_start, int block_order_index)
```

Function arguments:

- `paramdata_start`: Pointer to the start of the parameter data for a given phase.
- `block_order_index`: Block identifier for the required parameter data block.

Function result:
Pointer to the beginning of the specified parameter data block returned as a `void *`.

D.4 Programming required for handling scratch memory

Scratch memory is simply a block of memory of a size specified by an algorithm. Each shot phase has its own scratch memory area allocated. Scratch memory should be used instead of allocating variables of type `static` in the algorithm's real time C code, especially if the storage space required for `static` variables is large. Because of the way the memory of the real time computer is organized, the amount of memory available for `static` variables is limited so the use of scratch memory helps ensure proper operation of the PCS.

Scratch memory is handled by the application programmer in a manner that is similar to the way that the parameter data is handled. The algorithm code requests that a specified amount of memory be made available and a position in the integer step portion of the target vector is used to point to the scratch memory. During setup for a shot the scratch memory is allocated and initialized to zero by the infrastructure code. During real time execution, the integer step target value is used to locate the scratch area.

To provide an area of scratch memory for execution of an algorithm, follow the following steps.

1. Define a C language structure in the `CONTROLDEF` portion of the algorithm master file (see Sec. 4.3), that will be used by the algorithm real time code to access the scratch memory. Choose a structure type name that includes the algorithm identifier to ensure that it is unique. Definitions included in the `CONTROLDEF` portion of the algorithm master file are available to all of the PCS code.

   Also in the `CONTROLDEF` portion of the algorithm master file, define a macro for the element of the integer step portion of the target vector that will be used to hold the pointer to the scratch memory.

2. Provide a function in the waveform server that returns the number of bytes of scratch memory required. This function can specify a different scratch memory size for each virtual real time computer on which the algorithm executes. The call format for this function is

   `void alg_scratch_size(struct shotphase *phase, int vcpu)`

   The function arguments are:

   - `phase`: pointer to the descriptor of the shot phase for which the scratch memory will be made available.
   - `vcpu`: the number of the virtual cpu on which the memory will be allocated. Use the macros `CPUA`, `CPUB` to compare to the value of `vcpu`.

   Typically, the size of the scratch memory will be equal to the size of the structure defined in the `CONTROLDEF` area of the algorithm. However, the function could use information from the descriptor (such as the content of parameter data) to determine how large the scratch space should be.

3. In the `alg_vectors` function, provide a `case` that will compute the value for the integer step target vector element that points to the scratch memory. The method in Sec. 2.11.7 for accessing parameter data in real time also applies to scratch memory, except for the `case` macro that should be used. To generate the pointer to scratch memory use the macro `case_ptr_scratch_wvmp` rather than `case_ptr_param_wvmp` which is used to generate a pointer to parameter data.

4. Be certain that the data entry number for the integer target vector element appears in a list that will cause the target vector element to be calculated (for instance in the

list associated with a waveform or in one of the lists in the `categories_algorithms` structure).

D.4.1 case_ptrt_scratch_wvmp

Calling format:

```
\ntt{case\_ptrt\_scratch\_wvmp(THE\_DE,THE\_T,CPU,WAVEFORM,OFFSET,FACTOR)}
```

Macro arguments:

- `THE_DE`: data entry index.
- `THE_T`: target vector index.
- `CPU`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.).
- `WAVEFORM`: string giving the waveform name. Must be used by the algorithm used in `phase`.
- `OFFSET`: a pointer to a data object descriptor (Sec. 2.11.4.5). The size in bytes of the data object is used as the value to add to the waveform vertices. If the pointer is `NULL`, no descriptor is expected and the size value is set to 0.
- `FACTOR`: a pointer to a data object descriptor (Sec. 2.11.4.5). The size in bytes of the data object the size in bytes is used as the value to scale the waveform vertices. If the pointer is `NULL`, no descriptor is expected and the size value is set to 0.

This macro causes the data for the specified element in the pointer target vector to be equal to a pointer to a location at a specified offset into the scratch data area for the particular shot phase. The offset is generated by multiplying the waveform by the byte size given by `FACTOR`, then adding the number of bytes given by `OFFSET`. The waveform vertices are converted to integers before performing this calculation.

Here, `OFFSET` and `FACTOR` are pointers to data object descriptors (or `NULL`). The `convert_size` function is used along with the processor type of the real time processor indicated by `CPU` to determine the size in bytes of the data object described by each of these descriptors. If the pointer is `NULL`, the corresponding value (offset or factor) is set to 0 (i.e. the pointer is to the beginning of the scratch memory area).

If `WAVEFORM` is a null pointer, a waveform is not used and `FACTOR` is ignored. Instead, a single value assigned to $t = 0$ is generated with integer value equal to `OFFSET`. This can be used to generate a constant (in time) pointer into the scratch memory area.

The change list entry is generated as type `PTRT_SCRATCH_CHANGE` to indicate that the integer data is an offset into the scratch memory area to which the address of the scratch memory area should be added to get the real time value of the pointer vector element.

The pointer to the scratch memory can be accessed as in the following example where the scratch memory contains an array of floats and `PTR_ELEMENT_NUMBER` is the macro for the pointer target vector element number.

```
float *mydata;
float **pointertargets;

pointertargets = (float **)rtheap->pointer_target;
mydata = (float *)pointertargets[PTR_ELEMENT_NUMBER - 1];
```

D.5 Macros for generating pointers in the integer step target vector

D.5.1 case_ptr_paramblock_wvmp

Calling format:

```
case_ptr_paramblock_wvmp(THE_DE,THE_T,CPU,WAVEFORM,
                         OFFSET,FACTOR,BLOCK)
```

Macro arguments:

- `THE_DE`: data entry index.
- `THE_T`: target vector index.
- `CPU`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.).
- `WAVEFORM`: string giving the waveform name. Must be used by the algorithm used in `phase`.
- `OFFSET`: the value to add to the waveform vertices.
- `FACTOR`: the value to scale the waveform vertices.
- `BLOCK`: the block order index assigned to the required parameter data block.

This macro causes integer data to be generated by multiplying the waveform by `FACTOR` and then adding `OFFSET`. The integer data are interpreted as offsets into the parameter data block specified by `BLOCK` in the shot phase specified by `phase`. The integer step target vector element given by `THE_T` on the virtual cpu given by `CPU` is set equal to a pointer to the data at this offset from the beginning of the parameter data block. So, the integer step target vector element can be used to point to a specific portion of the parameter data block and the portion of data that is pointed to can change in time.

If `WAVEFORM` is a null pointer, a waveform is not used and `FACTOR` is ignored. Instead, a single value assigned to $t = 0$ (relative to the start of the shot phase) is generated with integer value equal to `OFFSET`. This can be used to generate a constant (in time) pointer into the parameter data block.

The pointer to the parameter data block should be accessed as in the following example (here the parameter data block to be accessed is assumed to be an array of floats).

```
float *mydata;
int *istargets;

istargets = (int *)rtheap->adtarget;
mydata = (float *)istargets[THE_T - 1];
```

D.5.2 case_ptr_scratch_wvmp

Calling format:

```
\ntt{case\_ptr\_scratch\_wvmp(THE\_DE,THE\_T,CPU,WAVEFORM,OFFSET,FACTOR)}
```

Macro arguments:

- `THE_DE`: data entry index.
- `THE_T`: target vector index.
- `CPU`: virtual cpu index (e.g. `CPUA`, `CPUB`, `CPUC` etc.).
- `WAVEFORM`: string giving the waveform name. Must be used by the algorithm used in `phase`.
- `OFFSET`: the value to add to the waveform vertices.
- `FACTOR`: the value to scale the waveform vertices.

This macro causes integer data to be generated by multiplying the waveform by `FACTOR` and then adding `OFFSET`. The integer data is interpreted as an offset into the block of scratch memory belonging to the shot phase specified by `phase`. The integer step target vector element given by `THE_T` on the virtual cpu given by `CPU` is set equal to a pointer to the data at this offset from the beginning of the scratch memory block. So, the integer step target vector element can be used to point to a specific portion of the scratch memory and the portion of data that is pointed to can change in time.

If `WAVEFORM` is a null pointer, a waveform is not used and `FACTOR` is ignored. Instead, a single value assigned to $t = 0$ (relative to the start of the shot phase) is generated with integer value equal to `OFFSET`. This can be used to generate a constant (in time) pointer into the scratch memory.

The pointer to the scratch memory should be accessed as in the following example (here the block of scratch memory to be accessed is assumed to be an array of floats).

```
float *mydata;
int *istargets;

istargets = (int *)rtheap->adtarget;
mydata = (float *)istargets[THE_T - 1];
```

D.6 Arrays of Vector Element Information

The waveform server global variables that are a standard part of an algorithm are arrays of constants that provide definitions of data relevant to the algorithm. The arrays are referenced through the `categories_algorithms` structure (described in Sec. 4.6). These arrays are as follows.

1. Vector element information: Recall that each control category has assigned to it one or more blocks of elements in each real time vector. A given algorithm can make use of any of the vector elements that are assigned to its category. The vector element information arrays list which of the vector elements are used by the algorithm and give some information about each element. Much of this information is provided for the purpose of archiving the vector element data or for diagnostics. Background information on this is in Sec. 2.7.

   Each array of vector element information is an array of structures. Each structure provides information about one vector element. There is one array for each of the following vectors: target, pointer target, error, shape, and command. The array for the error vector is also used for the P vector and the array for the command vector is used for both the intcommand and fpcommand vectors.

   Each structure contains the following information.

   (a) A string of 1 to 10 characters giving a name to assign to the vector element. If the vector element data is archived, then the name becomes the pointname for that data. If the data is not archived, then the name should still be provided because it is used for diagnostics.

   By convention, the name has the following format.

   `AlgVectorElement`

   `Alg` is a two character abbreviation for the algorithm. For example, `DN` for the `dnull` algorithm.

`Vector` is a single character indicating the real time vector: `T` for floating point values in the target vector (i.e. the continuous or float step portions of the target vector), `I` for integer portions of the target vector, `R` for the pointer target vector, `E` for the error vector, `S` for the shape vector and `C` for the command vector. For the P vector, the letter `E` in the error vector name is replaced by `P` (automatically by the code when the data is archived) and the remainder of the name remains the same. For the command vector, the name is used as given for the intcommand vector (indicated by the `C`) and the `C` is replaced by `F` for the fpcommand vector, which is composed of floating point values.

`Element` is up to 7 arbitrary characters giving the remainder of the name. Ideally, the name is an abbreviation for something relevant to the algorithm.

Note that the name must be unique among all data archived in the tokamak database if it is to be used for archiving the data.

(b) The macro giving the number of the vector element. See Sec. 2.10.3 for a description of these macros. The macro should have been defined in the section of the algorithm master file containing the symbol definitions (see Sec. 4.3).

(c) The archive flag. Set this value to 1 to have the data archived in a pointname automatically after each discharge (Sec. 2.7), 0 otherwise.

(d) The inherent number. If the data is archived, this value is stored in the pointname header (Sec. 2.7).

(e) The physics zero. If the data is archived, this value is stored in the pointname header (Sec. 2.7).

An example of one of these structures is as follows.

```
{
        "DNTZPP",       /* element name */
         TA_ANLGSTD_ZPP, /* element number */
        1,      /* archive mask */
        1.0,    /* inherent number */
        0.0     /* physics zero */
},
```

Note that each array of vector information structures must be declared to have length equal to the maximum size possible for the category. This is done by using a predefined macro for the category that gives the total number of elements allocated to the category in a given vector. For example, for the shape category, the number of elements allocated for the target vector is given by `BL_TA_TOTAL_SHAPE` where `BL` stands for "block length", `TA` stands for "target vector on virtual real time computer A" and `SHAPE` indicates the category. Any elements of the array that are not explicitly initialized will be set to default values by the compiler (usually 0).

For an algorithm that executes code on multiple real time computers, there must be arrays of vector information for each of the computers (see Sec. 2.9).

By convention these arrays are named as follows, where `alg` is the algorithm identifier.

- `alg_target_info` for the target vector.
- `alg_pointer_target_info` for the pointer target vector.
- `alg_error_info` for the error vector.
- `alg_shape_info` for the shape vector.
- `alg_cmd_info` for the command vectors.

2. Function vector elements. This is an array of integers giving the list of all of the elements in the function vector that are used by the algorithm (see Sec. 2.5.2 for background on the function vector). There must be an array for each of the real time computers used by the algorithm. As with other real time vectors, the appropriate macro should be used to specify an element number. The length of the array must be declared equal to the number of elements assigned to the category (by using the appropriate macro) but the algorithm does not need to use all of the elements so all elements do not need to be specified. Any array elements not explicitly initialized are set to 0 by the compiler.

3. Function names. This is an array of strings giving the names of the functions that are to be associated with the corresponding function vector elements called out in the previous array (see Sec. 2.5.2 for background information on the function vector). A function that is written in the C language must have its name prefixed with an underscore.

   By convention this array is named `alg_fcn_map` where `alg` is the algorithm identifier because the elements of the array map the algorithm into the function vector.

D.7 Infrastructure options

The infrastructure builds pretty much the same for all installations. But as more installations use the PCS, some options were required to specify specific cases. These options could not be provided through the installation. Therefore, a new file was needed to specify installation-specific options needed by the infrastructure. This file is called `infra_installdefs_<PCSLOCATION>.h`. In it are macro definitions which define array sizes such as the maximum number of physical cpus, `MAX_PCPUS`, and the maximum number of categories, `MAX_CATEGORIES`. Also specified are the pointname size, `POINTNAME_SIZE`, and the time to wait for the lockout server to wait for a cpu to finish, `SHOT_COMPLETE_WARNING_DELAY`, among other definitions. The defaults are usually adequate.

This file can define a macro to specify which section a particular real time piece of memory is put: in the `cacheable` section or the `noncacheable` section (see Sec. 13.3). For example, the parameter data is put in the `noncacheable` section by default. If this does not work for some reason, then the macro `RTPARAMETERS_PREFERENCE` can be defined to be `CACHEABLE` to get it in the `cacheable` section.

D.8 Old parameter data block routines

The following functions are considered obsolete even though they will continue to work. The newer versions of these functions should be used instead (Sec. 14.3). The reason that these functions are obsolete is because they cannot deal with shared parameter data blocks (Sec. 2.11.4.8) due to limitations on the usage flags argument.

D.8.1 put_param_block

This function is used in the waveform server to create a parameter data block. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This is the most generic function available for parameter data block creation.

Calling format:

```
int put_param_block(
    char **parameters,
    char *name,
    int flags,
    int block_order_index,
    int alignment,
    STRUCT_DESCRIPTORS *descriptor,
    void *data)
```

Function arguments:

- `parameters`: The pointer to the location where the pointer to the parameter data block will be stored. The pointer to a parameter data block is always stored in the descriptor for a shot phase. So, given the descriptor for a shot phase,

  ```
  struct shotphase *phase
  ```

  this argument should always be

`&phase->parameters`

The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.
- `flags`: An integer bit mask that specifies the way the parameter data block is used. This argument is the logical `OR` of one or more bit mask macros, as described in Sec. D.8.27.
- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.
- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.
- `descriptor`: Pointer to the data object descriptor for the data that will be stored in the parameter data block. This argument is used to determine the number of bytes to copy into the parameter block. It is also used to determine how to copy the data to the real time processor or for archiving. Section. 2.11.4.5 has a detailed discussion of data object descriptors.
- `data`: Pointer to the block of bytes to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns the number of bytes copied into the parameter data block. If this value is 0, the parameter data block creation may have failed.

Example:
Section 2.11.3 contains a discussion of examples for various usages of parameter data blocks.

D.8.2 put_param_block_chars

This function is used in the waveform server to create a parameter data block that contains an array of type `char`. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This function performs the same task as the generic parameter data block creation function `put_param_block` (Sec. D.8.1) except that this function doesn't require a data object descriptor as an argument. Instead, this function takes an integer argument specifying the number of elements in the array to be copied to the parameter data block. This function simply creates the necessary data object descriptor and then calls `put_param_block`.

This function is best used to create a byte array or a single character string. Though it can be used to create a concatenated set of character strings all of the same size, it is better to use `put_param_block_strings` to create a block with multiple strings.

Calling format:

```
int put_param_block_chars(
    char **parameters,
    char *name,
    int flags,
    int block_order_index,
    int alignment,
    int count,
    char *data)
```

Function arguments:

- `parameters`: The pointer to the location where the pointer to the parameter data block will be stored. The pointer to a parameter data block is always stored in the descriptor for a shot phase. So, given the descriptor for a shot phase,

  ```
  struct shotphase *phase
  ```

  this argument should always be

  ```
  &phase->parameters
  ```

  The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.

- `flags`: An integer bit mask that specifies the way the parameter data block is used. This argument is the logical `OR` of one or more bit mask macros, as described in Sec. D.8.27.

- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.
- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.
- `count`: The number of bytes in the array to be copied into the parameter data block.
- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns 0.
Examples:

```
1. Create a parameter data block containing an array of 25 byte values.
{
    int ii;
    char barray[25];

    for(ii=0;ii<25;ii++) barray[ii] = ii;

    put_param_block_chars(&(phase->parameters), /* parameter area */
"BYTE BLOCK EXAMPLE",  /* name */
FORHOSTVCPUB_MASK,/* flags */
BB_ALG_DATA_BLOCK,/* block_order_index */
BLOCK_ALIGN_0BYTES,/* mem_align */
25,/* num bytes in data */
barray); /* pointer to data */
}

2. Create a parameter data block containing a single character string.
{
    char label = 'file_example.txt'

    put_param_block_chars(&(phase->parameters), /* parameter area */
"SINGLE STRING BLOCK EXAMPLE",  /* name */
FORHOSTVCPUB_MASK,/* flags */
BB_ALG_DATA_BLOCK,/* block_order_index */
BLOCK_ALIGN_0BYTES,/* mem_align */
strlen(label)+1,/* num bytes in data */
label); /* pointer to data */
}

3. Create a parameter data block containing multiple strings all of the
   same size.
{
    char labels[2][40];

    memset((char *)labels,0,2*40);

    strcpy(labels[1-1], "label number 1");
    strcpy(labels[2-1], "label number 2");

    put_param_block_chars(&(phase->parameters), /* parameter area */
"CHAR BLOCK EXAMPLE",  /* name */
FORHOSTVCPUB_MASK,/* flags */
BB_ALG_DATA_BLOCK,/* block_order_index */
BLOCK_ALIGN_0BYTES,/* mem_align */
2 * 40,/* num bytes in data */
labels); /* pointer to data */
}
```

D.8.3 put_param_block_shorts

This function is used in the waveform server to create a parameter data block that contains an array of type `short`. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This function performs the same task as the generic parameter data block creation function `put_param_block` (Sec. D.8.1) except that this function doesn't require a data object descriptor as an argument. Instead, this function takes an integer argument specifying the number of elements in the array to be copied to the parameter data block. This function simply creates the necessary data object descriptor and then calls `put_param_block`.

Calling format:

```
int put_param_block_shorts(
    char **parameters,
    char *name,
    int flags,
    int block_order_index,
    int alignment,
    int count,
    short *data)
```

Function arguments:

- `parameters`: The pointer to the location where the pointer to the parameter data block will be stored. The pointer to a parameter data block is always stored in the descriptor for a shot phase. So, given the descriptor for a shot phase,

  ```
  struct shotphase *phase
  ```

  this argument should always be

  ```
  &phase->parameters
  ```

  The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.

- `flags`: An integer bit mask that specifies the way the parameter data block is used. This argument is the logical `OR` of one or more bit mask macros, as described in Sec. D.8.27.

- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.

- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.

- `count`: The number of elements in the array to be copied into the parameter data block.

- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns 0.

Example:

```
{
    int ii;
    short sarray[25];

    for(ii=0;ii<25;ii++) sarray[ii] = 0;

    put_param_block_shorts(&(phase->parameters),/* parameter area */
"SHORT BLOCK EXAMPLE",  /* name */
FORHOSTVCPUB_MASK,/* flags */
BB_ALG_DATA_BLOCK,/* block_order_index */
BLOCK_ALIGN_0BYTES,/* mem_align */
25,/* num shorts in data */
sarray); /* pointer to data */
}
```

D.8.4 put_param_block_ints

This function is used in the waveform server to create a parameter data block that contains an array of type `int`. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This function performs the same task as the generic parameter data block creation function `put_param_block` (Sec. D.8.1) except that this function doesn't require a data object descriptor as an argument. Instead, this function takes an integer argument specifying the number of elements in the array to be copied to the parameter data block. This function simply creates the necessary data object descriptor and then calls `put_param_block`.

Calling format:

```
int put_param_block_ints(
    char **parameters,
    char *name,
    int flags,
    int block_order_index,
    int alignment,
    int count,
    int *data)
```

Function arguments:

- `parameters`: The pointer to the location where the pointer to the parameter data block will be stored. The pointer to a parameter data block is always stored in the descriptor for a shot phase. So, given the descriptor for a shot phase,

  ```
  struct shotphase *phase
  ```

  this argument should always be

  ```
  &phase->parameters
  ```

  The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.

- `flags`: An integer bit mask that specifies the way the parameter data block is used. This argument is the logical `OR` of one or more bit mask macros, as described in Sec. D.8.27.

- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.

- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.

- `count`: The number of elements in the array to be copied into the parameter data block.

- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns 0.
Examples:

1. Create a parameter data block containing a single integer. This might be, for instance, a block that is part of a static data item. This block might contain a value that indicates the size of a second block.

```
{
    int value;

    value = 0;

    put_param_block_ints(&(phase->parameters),/* parameter area */
"INT BLOCK EXAMPLE",  /* name */
FORHOSTVCPUB_MASK,/* flags */
BB_ALG_VALUE_BLOCK,     /* block_order_index */
BLOCK_ALIGN_OBYTES,/* alignment */
1,/* num ints in data */
&value);         /* pointer to data */
}
```

2. Store an array of 25 integers.

```
{
    int ii;
    int iarray[25];

    for(ii=0;ii<25;ii++) iarray[ii] = 0;

    put_param_block_ints(&(phase->parameters),/* parameter area */
"INT BLOCK EXAMPLE",  /* name */
FORHOSTVCPUB_MASK,/* flags */
BB_ALG_DATA_BLOCK,/* block_order_index */
BLOCK_ALIGN_OBYTES,/* mem_align */
25,/* num ints in data */
iarray); /* pointer to data */
}
```

D.8.5 put_param_block_longs

This function is used in the waveform server to create a parameter data block that contains an array of type `long`. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This function performs the same task as the generic parameter data block creation function `put_param_block` (Sec. D.8.1) except that this function doesn't require a data object descriptor as an argument. Instead, this function takes an integer argument specifying the number of elements in the array to be copied to the parameter data block. This function simply creates the necessary data object descriptor and then calls `put_param_block`.

Calling format:

```
int put_param_block_longs(
    char **parameters,
    char *name,
    int flags,
    int block_order_index,
    int alignment,
    int count,
    long *data)
```

Function arguments:

- `parameters`: The pointer to the location where the pointer to the parameter data block will be stored. The pointer to a parameter data block is always stored in the descriptor for a shot phase. So, given the descriptor for a shot phase,

  ```
  struct shotphase *phase
  ```

  this argument should always be

  ```
  &phase->parameters
  ```

  The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.

- `flags`: An integer bit mask that specifies the way the parameter data block is used. This argument is the logical `OR` of one or more bit mask macros, as described in Sec. D.8.27.

- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.

- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.

- `count`: The number of elements in the array to be copied into the parameter data block.
- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns 0.
Example:

```
{
    int ii;
    long larray[25];

    for(ii=0;ii<25;ii++) larray[ii] = 0;

    put_param_block_longs(&(phase->parameters),/* parameter area */
"LONG BLOCK EXAMPLE",  /* name */
FORHOSTVCPUB_MASK,/* flags */
BB_ALG_DATA_BLOCK,/* block_order_index */
BLOCK_ALIGN_OBYTES,/* mem_align */
25,/* num longs in data */
larray); /* pointer to data */
}
```

D.8.6 put_param_block_floats

This function is used in the waveform server to create a parameter data block that contains an array of type `float`. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This function performs the same task as the generic parameter data block creation function `put_param_block` (Sec. D.8.1) except that this function doesn't require a data object descriptor as an argument. Instead, this function takes an integer argument specifying the number of elements in the array to be copied to the parameter data block. This function simply creates the necessary data object descriptor and then calls `put_param_block`.

Calling format:

```
int put_param_block_floats(
    char **parameters,
    char *name,
    int flags,
    int block_order_index,
    int alignment,
    int count,
    float *data)
```

Function arguments:

- `parameters`: The pointer to the location where the pointer to the parameter data block will be stored. The pointer to a parameter data block is always stored in the descriptor for a shot phase. So, given the descriptor for a shot phase,

  ```
  struct shotphase *phase
  ```

  this argument should always be

  ```
  &phase->parameters
  ```

  The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.

- `flags`: An integer bit mask that specifies the way the parameter data block is used. This argument is the logical `OR` of one or more bit mask macros, as described in Sec. D.8.27.

- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.

- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.

- `count`: The number of elements in the array to be copied into the parameter data block.

- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns 0.
Example:

```
{
    int ii;
    float farray[25];

    for(ii=0;ii<25;ii++) farray[ii] = 0;

    put_param_block_floats(&(phase->parameters),/* parameter area */
"FLOAT BLOCK EXAMPLE",  /* name */
FORHOSTVCPUB_MASK,/* flags */
BB_ALG_DATA_BLOCK,/* block_order_index */
BLOCK_ALIGN_0BYTES,/* mem_align */
25,/* num floats in data */
farray); /* pointer to data */
}
```

D.8.7 put_param_block_doubles

This function is used in the waveform server to create a parameter data block that contains an array of type `double`. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This function performs the same task as the generic parameter data block creation function `put_param_block` (Sec. D.8.1) except that this function doesn't require a data object descriptor as an argument. Instead, this function takes an integer argument specifying the number of elements in the array to be copied to the parameter data block. This function simply creates the necessary data object descriptor and then calls `put_param_block`.

Calling format:

```
int put_param_block_doubles(
    char **parameters,
    char *name,
    int flags,
    int block_order_index,
    int alignment,
    int count,
    double *data)
```

Function arguments:

- `parameters`: The pointer to the location where the pointer to the parameter data block will be stored. The pointer to a parameter data block is always stored in the descriptor for a shot phase. So, given the descriptor for a shot phase,

  ```
  struct shotphase *phase
  ```

  this argument should always be

  ```
  &phase->parameters
  ```

  The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.

- `flags`: An integer bit mask that specifies the way the parameter data block is used. This argument is the logical `OR` of one or more bit mask macros, as described in Sec. D.8.27.

- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.

- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.

- `count`: The number of elements in the array to be copied into the parameter data block.

- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns 0.
Example:

```
{
    int ii;
    double darray[25];

    for(ii=0;ii<25;ii++) darray[ii] = 0;

    put_param_block_doubles(&(phase->parameters),/* parameter area */
"DOUBLE BLOCK EXAMPLE",  /* name */
FORHOSTVCPUB_MASK,/* flags */
BB_ALG_DATA_BLOCK,/* block_order_index */
BLOCK_ALIGN_0BYTES,/* mem_align */
25,/* num doubles in data */
darray); /* pointer to data */
}
```

D.8.8 put_param_block_strings

This function is used in the waveform server to create a parameter data block that contains an array of type `char`. It is usually called in an algorithm's `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) function.

This function performs the same task as the generic parameter data block creation function `put_param_block` (Sec. D.8.1) except that this function doesn't require a data object descriptor as an argument. Instead, this function takes an integer argument specifying the number of elements in the array to be copied to the parameter data block. This function adds up the lengths of the strings, combines them with NULL characters as separators, creates the necessary data object descriptor, and then calls `put_param_block`.

This function is best used to create multiple strings in a single block. If the string array ends with a NULL string, then the count can be 0 and this function will determine how many strings there are by counting until the NULL string.

Calling format:

```
int put_param_block_strings(
    char **parameters,
    char *name,
    int flags,
    int block_order_index,
    int alignment,
    int count,
    char **data)
```

Function arguments:

- `parameters`: The pointer to the location where the pointer to the parameter data block will be stored. The pointer to a parameter data block is always stored in the descriptor for a shot phase. So, given the descriptor for a shot phase,

  ```
  struct shotphase *phase
  ```

  this argument should always be

  ```
  &phase->parameters
  ```

  The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the parameter data block. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.

- `flags`: An integer bit mask that specifies the way the parameter data block is used. This argument is the logical `OR` of one or more bit mask macros, as described in Sec. D.8.27.

- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.

- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.

- `count`: The number of strings in the array to be copied into the parameter data block.

- `data`: Pointer to the string array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.

Return value:
The function returns 0.
Example:

```
{
    int ii;
    char strings[] = { "string1", "string2"};

    put_param_block_strings(&(phase->parameters), /* parameter area *\
                     "STRING BLOCK EXAMPLE", /* name *\
                     FORHOSTVCPUB_MASK,      /* flags *\
                     1,                      /* rtcpu_block_index *\
                     BLOCK_ALIGN_0BYTES,     /* mem_align *\
                     2,                      /* num strings in data *\
                     strings);               /* pointer to array *\
}
```

D.8.9 put_param_labeled_int_array

This function is used in the waveform server to create a group of parameter data blocks that are associated with the standard static data item (Sec. 2.12) of type `labeled int array`. It should be called in an algorithm's `alg_parameters` function.

A static data item (Sec. 2.10.1.2) should also be created which uses the `generic_editor` to display the data, and the waveform descriptor should be set to `SD_LABELED_INT_ARRAY` so that the data can be restored correctly even if the size of the array changes in the future.

Calling format:

```
int put_param_labeled_int_array(
    char **parameters,
    char *name,
    int rtflags,
    int archive_mask,
    int block_order_index,
    int alignment,
    int count,
    int *data,
    char **labels,
    int *substitute_data);
```

Function arguments:

- `parameters`: The pointer to the location where the pointer to the parameter data block will be stored. The pointer to a parameter data block is always stored in the descriptor for a shot phase. So, given the descriptor for a shot phase,

  ```
  struct shotphase *phase
  ```

this argument should always be

```
&phase->parameters
```

The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the data item which describes this set of parameter data blocks. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.
- `rtflags`: An integer bit mask that specifies which virtual real time computers the parameter data block is used. This argument is the logical `OR` of one or more bit mask macros, as described in Sec. D.8.27.
- `archive_mask`: An integer bit mask that specifies whether the parameter data blocks should be archived. This argument is either 0 or FORARCHIVE_MASK.
- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.
- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.
- `count`: The number of elements in the array to be copied into the parameter data block.
- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.
- `labels`: An array of labels, one per array value, which is used to match up data values when this data item is restored.
- `substitute_data`: An array of values that are used to substitute for values that are not in the archived data block.

Return value:
The function returns 0.
Example:

```
Define the static data item:

#ifdef WAVEFORMS
{/* StaticData */
 0,                         /* ignored */
 {0, 0, 0.0, 0.0},          /* ignored */
 "labeled int array",       /* label identifying param data, 20 chars max */
 "generic_editor",          /* name of idl routine to handle this data */
 " ",                       /* ignored */
 " ",                       /* ignored */
 1.0,                       /* ignored */
 0.0,                       /* ignored */
 " ",                       /* ignored */
 SD_LABELED_INT_ARRAY,      /* indicates not a waveform, 40 chars max */
 C_CODE,                    /* control category index */
 A_CODE,                    /* control algorithm index */
 {0},                       /* target vectors affected */
 0,                         /* ignored */
 0,                         /* ignored */
 {0.0},                     /* ignored */
 0,                         /* ignored */
 0,                         /* ignored */
 1,                         /* subset number */
 0                          /* ignored */
},
#endif

Create the parameter data in the alg_parameters function:
{
    int ii;
    int values[3];

    char *labels[1] = { "value 1", "value 2", "value 3" };

    for(ii=0;ii<3;ii++) values[ii] = 0;

    put_param_labeled_int_array(&(phase->parameters),/* parameter area */
                                "labeled int array",/* name */
                                FORHOSTVCPUA_MASK,/* rtflags */
                                FORARCHIVE_MASK,/* archive mask */
                                BA_ALG_DATA_BLOCK,/* block_order_index */
                                BLOCK_ALIGN_4BYTES,/* alignment */
                                3,/* count */
                                values,/* data */
                                labels,/* labels */
                                values);/* substitution_data */
}
```

D.8.10 put_param_labeled_float_array

This function is used in the waveform server to create a group of parameter data blocks that are associated with the standard static data item (Sec. 2.12) of type `labeled float array`. It should be called in an algorithm's `alg_parameters` function.

A static data item (Sec. 2.10.1.2) should also be created which uses the `generic_editor` to display the data, and the waveform descriptor should be set to `SD_LABELED_FLOAT_ARRAY` so that the data can be restored correctly even if the size of the array changes in the future.

Calling format:

```
int put_param_labeled_float_array(
    char **parameters,
    char *name,
    int rtflags,
    int archive_mask,
    int block_order_index,
    int alignment,
    int count,
    float *data,
    char **labels,
    float *substitute_data);
```

Function arguments:

- `parameters`: The pointer to the location where the pointer to the parameter data block will be stored. The pointer to a parameter data block is always stored in the descriptor for a shot phase. So, given the descriptor for a shot phase,

  ```
  struct shotphase *phase
  ```

  this argument should always be

  ```
  &phase->parameters
  ```

The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the data item which describes this set of parameter data blocks. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.
- `rtflags`: An integer bit mask that specifies which virtual real time computers the parameter data block is used. This argument is the logical `OR` of one or more bit mask macros, as described in Sec. D.8.27.
- `archive_mask`: An integer bit mask that specifies whether the parameter data blocks should be archived. This argument is either 0 or FORARCHIVE_MASK.
- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.
- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.
- `count`: The number of elements in the array to be copied into the parameter data block.
- `data`: Pointer to the array to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.
- `labels`: An array of labels, one per array value, which is used to match up data values when this data item is restored.
- `substitute_data`: An array of values that are used to substitute for values that are not in the archived data block.

Return value:
The function returns 0.
Example:

```
Define the static data item:

#ifdef WAVEFORMS
```

```
{/* StaticData */
 0,                        /* ignored */
 {0, 0, 0.0, 0.0},         /* ignored */
 "labeled float array",    /* label identifying param data, 20 chars max */
 "generic_editor",         /* name of idl routine to handle this data */
 " ",                      /* ignored */
 " ",                      /* ignored */
 1.0,                      /* ignored */
 0.0,                      /* ignored */
 " ",                      /* ignored */
 SD_LABELED_FLOAT_ARRAY,   /* indicates not a waveform, 40 chars max */
 C_CODE,                   /* control category index */
 A_CODE,                   /* control algorithm index */
 {0},                      /* target vectors affected */
 0,                        /* ignored */
 0,                        /* ignored */
 {0.0},                    /* ignored */
 0,                        /* ignored */
 0,                        /* ignored */
 1,                        /* subset number */
 0                         /* ignored */
},
#endif

Create the parameter data in the alg_parameters function:
{
{
    int ii;
    float values[3];

    char *labels[1] = { "value 1", "value 2", "value 3" };

    for(ii=0;ii<3;ii++) values[ii] = 0.0;

    put_param_labeled_float_array(&(phase->parameters),/* parameter area */
                                "labeled float array",/* name */
                                FORHOSTVCPUA_MASK,/* rtflags */
                                FORARCHIVE_MASK,/* archive mask */
                                BA_ALG_DATA_BLOCK,/* block_order_index */
                                BLOCK_ALIGN_4BYTES,/* alignment */
                                3,/* count */
                                values,/* data */
                                labels,/* labels */
                                values);/* substitution_data */
}
```

D.8.11 put_param_labeled_structure

This function is used in the waveform server to create a group of parameter data blocks that are associated with the standard static data item (Sec. 2.12) of type `labeled structure`. It should be called in an algorithm's `alg_parameters` function.

A static data item (Sec. 2.10.1.2) should also be created which uses the `generic_editor` to display the data, and the waveform descriptor should be set to `SD_LABELED_STRUCTURE` so that the data can be restored correctly even if the size of the structure changes in the future.

Calling format:

```
int put_param_labeled_structure(
    char **parameters,
    char *name,
    int rtflags,
    int archive_mask,
    int block_order_index,
    int alignment,
    int count,
    STRUCT_DESCRIPTORS *descriptor,
    int *data,
    char **labels,
    int *substitute_data);
```

Function arguments:

- `parameters`: The pointer to the location where the pointer to the parameter data block will be stored. The pointer to a parameter data block is always stored in the descriptor for a shot phase. So, given the descriptor for a shot phase,

  ```
  struct shotphase *phase
  ```

  this argument should always be

  ```
  &phase->parameters
  ```

  The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the data item which describes this set of parameter data blocks. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.
- `rtflags`: An integer bit mask that specifies which virtual real time computers the parameter data block is used. This argument is the logical `OR` of one or more bit mask macros, as described in Sec. D.8.27.
- `archive_mask`: An integer bit mask that specifies whether the parameter data blocks should be archived. This argument is either 0 or FORARCHIVE_MASK.
- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.
- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.
- `count`: The number of labels passed in. If the string array ends with a NULL string, then the count can be 0 and this function will determine how many strings there are by counting until the NULL string.
- `descriptor`: Pointer to the data object descriptor for the data that will be stored in the parameter data block. This argument is used to determine the number of bytes to copy into the parameter block. It is also used to determine how to copy the data to the real time processor or for archiving. Section. 2.11.4.5 has a detailed discussion of data object descriptors.
- `data`: Pointer to the structure to be copied into the parameter data block. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.
- `labels`: An array of labels, one per structure value, which is used to match up data values when this data item is restored.
- `substitute_data`: A structure of values that are used to substitute for values that are not in the archived data block.

Return value:

The function returns 0.

Current limitations:

- Embedded arrays (except char arrays) and structures are considered multiple values so there must be a label for each element of an array or structure.
- A single char value is considered a number rather than a string. IDL uses a range of 0-255 for its BYTE value so the C equivalent of unsigned char should be used as the variable type in the structure.
- A char array of at least size 2 is considered a character string. If the structure changes, the size of the char array can be increased, but should not decreased else archived data will not get restored.

Example:

```
Define the structure:

#ifdef CONTROLDEF
#define GEN_params \
SDSTART( struct params {     ,D_params      ) \
DGEN   (   int   value1;     ,INTTYPE,     1) \
DGEN   (   float value2;     ,FLOATTYPE,   1) \
DGEN   (   float value3;     ,FLOATTYPE,   1) \
SDEND  (               };                   )
GEN_params
#endif

Define the static data item:

#ifdef WAVEFORMS
{/* StaticData */
 0,                        /* ignored */
 {0, 0, 0.0, 0.0},         /* ignored */
 "labeled structure",      /* label identifying param data, 20 chars max */
 "matrix_editor",          /* name of idl routine to handle this data */
 " ",                      /* ignored */
 " ",                      /* ignored */
 1.0,                      /* ignored */
 0.0,                      /* ignored */
 " ",                      /* ignored */
 SD_LABELED_STRUCTURE,     /* indicates not a waveform, 40 chars max */
 C_CODE,                   /* control category index */
 A_CODE,                   /* control algorithm index */
 {0},                       /* target vectors affected */
 0,                         /* ignored */
 0,                         /* ignored */
 {0.0},                     /* ignored */
 0,                         /* ignored */
 0,                         /* ignored */
 1,                         /* subset number */
 0                          /* ignored */
},
#endif

Create the parameter data in the alg_parameters function:
{
    GEN_params
    struct params params;

    char *labels[1] = { "value 1", "value 2", "value 3" };

    params.value1 = 1;
    params.value2 = 1.0;
    params.value3 = 500.0;

    D_params[0].size = 1;
    put_param_labeled_structure(&(phase->parameters),/* parameter area */
                                "labeled structure",/* name */
                                FORHOSTVCPUA_MASK,/* rtflags */
                                FORARCHIVE_MASK,/* archive mask */
                                BA_ALG_DATA_BLOCK,/* block_order_index */
                                BLOCK_ALIGN_4BYTES,/* alignment */
                                3,/* count */
                                D_params,/* descriptor */
                                (void *)params,/* data */
                                labels,/* labels */
                                (void *)values);/* substitution_data */
}
```

D.8.12 put_param_labeled_matrices

This function is used in the waveform server to create a group of parameter data blocks that are associated with the standard static data item (Sec. 2.12) of type `labeled matrices`. It should be called in an algorithm's `alg_parameters` function.

A static data item (Sec. 2.10.1.2) should also be created which uses the `matrix_editor` to display the data, and the waveform descriptor should be set to `SD_LABELED_MATRICES` so that the data can be restored correctly even if the size of the matrix changes in the future.

Calling format:

```
int put_param_labeled_matrices(
    char **parameters,
    char *name,
    int flags,
    int block_order_index,
    int alignment,
    int num_matrices,
    int num_matrix_rows,
    int num_matrix_cols,
    char *matrix_names,
    char *matrix_row_names,
    char *matrix_col_names,
    float *matrix_data,
    int *substitute_data);
```

Function arguments:

- `parameters`: The pointer to the location where the pointer to the parameter data block will be stored. The pointer to a parameter data block is always stored in the descriptor for a shot phase. So, given the descriptor for a shot phase,

  ```
  struct shotphase *phase
  ```

  this argument should always be

  ```
  &phase->parameters
  ```

  The pointer to the shot phase descriptor is an argument to the `alg_parameters` (Sec. 2.11.4.1) or `alg_vectors` (Sec. 2.10.5) functions where this parameter data block creation function is normally called.

- `name`: A string that specifies the name of the data item which describes this set of parameter data blocks. There is a discussion of conventions for parameter data block names in Sec. 2.11.4.3.

- `flags`: An integer bit mask that specifies the way the parameter data block is used. This argument is the logical `OR` of one or more bit mask macros, as described in Sec. D.8.27.

- `block_order_index`: An integer value that is used primarily for locating the parameter data block on the real time processor. This argument has no use if the parameter data block is not copied to a real time processor. There is additional discussion of the usage of this value in Sec. 2.11.4.7.
- `alignment`: Specifies the required storage alignment of the parameter data block on the real time processor. This argument should be one of the macros described in Sec. 2.11.4.6.
- `num_matrices`: The number of matrices passed in.
- `num_matrix_rows`: The number of rows in the matrix.
- `num_matrix_cols`: The number of columns in the matrix.
- `matrix_names`: The names of the matrices that get displayed in the matrix editor. This is a 2-dimensional char array:

  ```
  char matrix_names[num_matrices][MATRIX_MAX_RC_NAME_SIZE]
  ```
- `matrix_row_names`: The names of the rows that get displayed in the matrix editor. This is a 2-dimensional char array:

  ```
  char matrix_names[num_matrix_rows][MATRIX_MAX_RC_NAME_SIZE]
  ```
- `matrix_col_names`: The names of the columns that get displayed in the matrix editor. This is a 2-dimensional char array:

  ```
  char matrix_names[num_matrix_cols][MATRIX_MAX_RC_NAME_SIZE]
  ```
- `data`: Pointer to the float array. Because these data are copied, the memory located at the region indicated by this pointer isn't needed after the parameter data block is created.
- `substitute_data`: A float array the size of one matrix of values that are used to substitute for values that are not in the archived matrices.

Return value:
The function returns 0.

Example:

```
Define the static data item:

#ifdef WAVEFORMS
{/* StaticData */
 0,                        /* ignored */
 {0, 0, 0.0, 0.0},         /* ignored */
 "C matrices",             /* label identifying param data, 20 chars max */
 "matrix_editor",          /* name of idl routine to handle this data */
 "(f12.4)",                /* format for matrix numbers */
 " ",                      /* ignored */
 1.0,                      /* ignored */
 0.0,                      /* ignored */
 " ",                      /* ignored */
 SD_LABELED_MATRICES,      /* indicates not a waveform, 40 chars max */
 C_CODE,                   /* control category index */
 A_CODE,                   /* control algorithm index */
 {0},                      /* target vectors affected */
 0,                        /* ignored */
 0,                        /* ignored */
 {0.0},                    /* ignored */
 0,                        /* ignored */
 0,                        /* ignored */
 1,                        /* subset number */
 0                         /* ignored */
},
#endif

Create the parameter data in the alg_parameters function:
/*
-------------------------------------------------------------------------------
PARAMETER:  "C Matrix ..."
-------------------------------------------------------------------------------
*/
{
#define NUM_MATRICES 2
#define NUM_MATRIX_ROWS 4
#define NUM_MATRIX_COLS 3
#define MATRIX_MAX_RC_NAME_SIZE 40
char matrix_names[NUM_MATRICES][MATRIX_MAX_RC_NAME_SIZE];
char row_names[NUM_MATRIX_ROWS][MATRIX_MAX_RC_NAME_SIZE];
char col_names[NUM_MATRIX_COLS][MATRIX_MAX_RC_NAME_SIZE];
```

```
float matrices[NUM_MATRICES][NUM_MATRIX_ROWS][NUM_MATRIX_COLS];
int ii,jj,kk;

/*
initializations
*/
    memset((char *)matrix_names,0,NUM_MATRICES*MATRIX_MAX_RC_NAME_SIZE);
    memset((char *)row_names,0,NUM_MATRIX_ROWS*MATRIX_MAX_RC_NAME_SIZE);
    memset((char *)col_names,0,NUM_MATRIX_COLS*MATRIX_MAX_RC_NAME_SIZE);
    for(ii=0;ii<NUM_MATRICES;ii++)
    {
        for(jj=0;jj<NUM_MATRIX_ROWS;jj++)
    for(kk=0;kk<NUM_MATRIX_COLS;kk++)
        matrices[ii][jj][kk] = 0.0;
    }
/*
initialize matrix names
*/
    for(jj = 0; jj < NUM_MATRICES; jj++)
        sprintf(matrix_names[jj],"Matrix %d",jj);
/*
initialize matrix row names
*/
    strcpy(row_names[1-1],"row 1
    strcpy(row_names[2-1],"row 2");
    strcpy(row_names[3-1],"row 3");
    strcpy(row_names[4-1],"row 4");
/*
initialize matrix column names
*/
    strcpy(col_names[1-1],"col 1");
    strcpy(col_names[2-1],"col 2");
    strcpy(col_names[3-1],"col 3");
/*
write to waveserver's memory
*/
    put_param_labeled_matrices(
&(phase->parameters),  /* address of start of paramter data */
"C matrices", /* matrix label */
FORHOSTVCPUA_MASK|FORARCHIVE_MASK|FORUSER_MASK, /* usage flag */
BA_CMATRIX, /* rtcpu block index */
BLOCK_ALIGN_4BYTES, /* mem alignment for real time computer */
NUM_MATRICES, /* num s matrices */
NUM_MATRIX_ROWS, /* num s matrix rows */
NUM_MATRIX_COLS, /* num s matrix cols */
(char *)matrix_names, /* matrix names */
(char *)row_names, /* matrix row names */
(char *)col_names, /* matrix column names */
(float *)matrices, /* matrix data */
        (float *)NULL); /* substitute data is all zeros */

#undef NUM_MATRICES
#undef NUM_MATRIX_ROWS
#undef NUM_MATRIX_COLS
#undef MATRIX_MAX_RC_NAME_SIZE
}
```

D.8.13 put_param_block_offsets

This function creates a "byte offset list" parameter data block. The parameter data block created contains an array of integers, each of which is the byte offset of a parameter data block from the beginning of the PCS memory heap. The list of offsets is for the parameter data blocks of the phase containing the byte offset list block that are located on a specified virtual CPU. The virtual CPU is one of the CPUs used by the category containing the phase for which the byte offset block is created.

The values in a byte offset list parameter data block are useful with the interprocessor communication functions (Sec. B). These functions take an argument that specifies the offset from the beginning of the PCS memory heap (Sec. 2.9.2) of the location to which data should be copied. It is convenient to create a scratch parameter data block to serve as a buffer to receive data from another real time CPU. A byte offset list parameter data block can be used on the CPU that is transmitting the data to determine the offset to the buffer in the scratch parameter data block. Further discussion is in Sec. B.

Note that a byte offset list parameter data block is not an ordinary block where the algorithm programmer provides the data to be stored there. Also, the algorithm programmer doesn't provide the parameter data block name. Instead, the name "block offsets for CPUx" is used, where "x" is replaced with the appropriate virtual CPU letter. This is a special parameter data block name. The infrastructure code recognizes this block name and fills the block with the appropriate offset values. The following is the content of the block in the waveform server.

- The `io_handler` copy of the parameter data block contains a single integer equal to 0.
- The work queue copy of the parameter data block is a model of the actual block that

will be on the real time processor. The number of integers in the block is the same as will be on the real time processor but the integer values are offsets from the beginning of the phase's parameter data area rather than the complete offset from the beginning of the PCS memory heap.

So, the content of the parameter data block is really only useful on the real time processor.

Calling format:

```
int put_param_block_offsets(char **parameters,
                            int vcpu,
                            int flags,
                            int rtcpu_block_num)
```

Function Arguments:

- `parameters`: Pointer to the location in the shot phase descriptor where the pointer to the parameter data for that phase is stored. For example, if the function calling `put_param_block_offsets` has a variable `struct shotphase *phase`, the pointer to the shot phase descriptor, this argument should be `&phase->parameters`. The pointer to the shot phase descriptor is a function argument to the `alg_parameters` function (Sec. 2.11.4.1) which is where `put_param_block_offsets` would normally be called.
- `vcpu`: The virtual CPU where the parameter data blocks are located for which the offset block will contain byte offset values. The virtual CPU should be specified using macros of the form `CPUx`, where `x` is an uppercase letter, `A`, `B`, etc (Sec. 2.9.1).
- `flags`: The standard parameter data block bit mask (Sec. D.8.27) that indicates how the block should be used. For a byte offset list parameter data block, usually this mask would be used to indicate the virtual CPUs on which the parameter data block should be placed.
- `block_order_index`: The block order index (Sec. 2.11.4.7) of the parameter data block to be created.

Return values:

This function returns 0.

Here is some example code. In this example, data are to be transmitted from CPU B to CPU A. The algorithm identifier is `example` and the category identifier is `mycategory`.

1. In the `CONTROLDEFS` section of the algorithm master file, create the block order index macro for the byte offset list parameter data block which will be placed on CPU B.

   ```
   #define BB_EXAMPLE_BLOCK_OFFSETS_CPUA (3)
   ```

Create the block order index macro of the scratch parameter data block to which the data will be copied. This block will be on CPU A.

```
#define BA_EXAMPLE_SCRATCH_BUFFER (5)
```

2. In the function `example_parameters`, create the byte offset list parameter data block. Also, in the `example_parameters` function create the scratch parameter data block that will receive the data. The `alg_parameters` function (Sec. 2.11.4.1) is the appropriate place to create these blocks because the blocks only need to be created once when the shot phase is initialized. Their contents don't depend on any raw data that could be modified by the PCS user. Here, the scratch parameter block is the correct size to hold a structure of type `scratch_buffer`. The structure and its descriptor (Sec. 2.11.4.5) should be defined in the `CONTROLDEFS` section of the algorithm master file.

```
GEN_scratch_buffer

put_param_block_offsets(&(phase->parameters),
CPUA,
FORHOSTVCPUB_MASK,
BB_EXAMPLE_BLOCK_OFFSETS_CPUA);
put_param_block(&(phase->parameters),
"scratch area CPU A",
FORHOSTVCPUA_MASK | SCRATCHDATA_MASK,
BA_EXAMPLE_SCRATCH_BUFFER,
BLOCK_ALIGN_OBYTES,
D_scratch_buffer,
NULL);
```

3. On CPU B, copy data to CPU A. Use the byte offset list parameter block to obtain the offset to the target buffer.

```
int offset,error_flag;
int *block_offsets;

block_offsets =
   (int *)rtheap->current_parameter_data[CATB_MYCATEGORY-1]\
                  [BB_EXAMPLE_BLOCK_OFFSETS_CPUA-1];
offset = block_offsets[BA_EXAMPLE_SCRATCH_BUFFER - 1];
error_flag = rtcipc_write(rtheap,
           CPUA_MYCATEGORY,offset,data_to_be_copied,
                   data_length,NULL);
```

D.8.14 get_param_block

This routine can return all the information for the parameter data block given the block's name. All the return arguments can be set to `NULL` if a value is not desired.

Calling format:

```
block_ptr = (block_type *)get_param_block(phase->parameters,
                                          block_name,
                                          &blkstart,
                                          &blksize,
                                          &blkdatasize,
                                          &flags,
                                          &insert_order,
                                          &mem_align,
                                          &num_descriptors,
                                          &descriptors);
```

Function Arguments:

- `phase->parameters`: The pointer to the parameter data for the phase.
- `char *block_name`: The name of the parameter data block.
- `int *blkstart`: Pointer to an integer returning the byte offset into the memory where the block starts, can be NULL.
- `int *blksize`: A pointer to an integer returning the number of bytes that the block uses including header information, can be NULL.
- `int *blkdatasize`: A pointer to an integer returning the number of bytes of data in the block, can be NULL.
- `int *flags`: A pointer to an integer returning the block flags (See Sec. D.8.27), can be NULL.
- `int *insert_order`: A pointer to an integer returning the block storage order (See Sec. 2.11.4.7), can be NULL.
- `int *mem_align`: A pointer to an integer returning the alignment of the block (See Sec. 2.11.4.6), can be NULL.
- `int *num_descriptors`: A pointer to an integer returning the number of descriptors used to describe the block, can be NULL.

- `STRUCT_DESCRIPTORS **descriptors`: A pointer to return the pointer to the location in the block where the array of block descriptor structures start (See Sec. 2.11.4.5), can be NULL.

Return value:

- The return value of the routine is a pointer to the data in the parameter data block. If the parameter data block couldn't be located, a null pointer is returned. The programmer needs to cast the return value to the desired variable type.

Note that the parameter data block pointer returned can become invalid if any changes are made to the parameter data in the phase in which the parameter data block is located. (See Sec. 2.11.4.9 for more discussion of this point.)

D.8.15 get_param_block_data_ptr

This routine returns the pointer to the data in the parameter data block given the block's name. It is the equivalent of calling `get_param_block` with all the return arguments set to NULL's.

Calling format:

```
block_ptr = (block_type *)get_param_block_data_ptr(
                                phase->parameters,block_name);
if(block_ptr != NULL)
{
    ...
}
```

Function Arguments:

- `phase->parameters`: The pointer to the parameter data for the phase.
- `char *block_name`: The name of the parameter data block.

Return value:

- The return value of the routine is a pointer to the data in the parameter data block. If the parameter data block couldn't be located, a null pointer is returned. The programmer needs to cast the return value to the desired variable type.

Note that the parameter data block pointer returned can become invalid if any changes are made to the parameter data in the phase in which the parameter data block is located. (See Sec. 2.11.4.9 for more discussion of this point.)

D.8.16 get_param_block_data_copy

This routine returns a pointer to a COPY of the data in the parameter data block given the block's name. The pointer to the memory must be freed when finished.

Calling format:

```
block_ptr = (block_type *)get_param_block_data_copy(
                          phase->parameters,block_name);
if(block_ptr != NULL)
{
   ...
   free(block_ptr);
}
```

Function Arguments:

- `phase->parameters`: The pointer to the parameter data for the phase.
- `char *block_name`: The name of the parameter data block.

Return value:

- The return value of the routine is a pointer to memory containing a copy of the data in the parameter data block. If the parameter data block couldn't be located, a null pointer is returned. The programmer needs to cast the return value to the desired variable type. When done, the pointer to the memory must be freed.

This routine is useful when adding or modifying the parameter data in a phase because it returns a copy of a parameter data block rather than a pointer to the data in the block. Adding, deleting, or replacing blocks will not affect the validity of this pointer.

D.8.17 replace_param_block

This routine replaces a given parameter data block. The block MUST exist. Most of the attributes of the block, the block order index, the memory alignment, and the flags, remain unchanged. The array of descriptors, however, can change and an argument is provided for such a case; if no changes are needed to the descriptors, this argument is set to NULL.

Calling format:

```
replace_param_block(&(phase->parameters),
                    block_name,
                    descriptors,
                    data_ptr);
```

Function Arguments:

- `phase->parameters`: The pointer to the parameter data for the phase which is returned. This pointer will change if the block is successfully replaced.
- `char *block_name`: The name of the parameter data block.
- `STRUCT_DESCRIPTORS *descriptors`: A pointer to an array of block descriptor structures describing the block (See Sec. 2.11.4.5), can be NULL. If NULL, then the existing descriptors are unchanged, in which case, the data length remains the same.
- `char *data_ptr`: The pointer to the char data.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`. (See Sec. 2.11.4.9 for more discussion of this point.)

D.8.18 replace_param_block_chars

This routine replaces a given parameter data block of type `char`. The block `MUST` exist.

Calling format:

```
replace_param_block_chars(&(phase->parameters),
                          block_name,
                          num_chars,
                          carray);
```

Function Arguments:

- `phase->parameters`: The pointer to the parameter data for the phase which is returned. This pointer will change if the block is successfully replaced.
- `char *block_name`: The name of the parameter data block.
- `int num_chars`: The number of chars in the data.
- `char *carray`: The pointer to the char data array.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

D.8.19 replace_param_block_shorts

This routine replaces a given parameter data block of type `short`. The block `MUST` exist.

Calling format:

```
replace_param_block_shorts(&(phase->parameters),
                          block_name,
                          num_shorts,
                          sarray);
```

Function Arguments:

- `phase->parameters`: The pointer to the parameter data for the phase which is returned. This pointer will change if the block is successfully replaced.
- `char *block_name`: The name of the parameter data block.
- `int num_shorts`: The number of shorts in the data.
- `short *sarray`: The pointer to the short data array.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

D.8.20 replace_param_block_ints

This routine replaces a given parameter data block of type `int`. The block `MUST` exist.

Calling format:

```
replace_param_block_ints(&(phase->parameters),
                        block_name,
                        num_ints,
                        iarray);
```

Function Arguments:

- `phase->parameters`: The pointer to the parameter data for the phase which is returned. This pointer will change if the block is successfully replaced.
- `char *block_name`: The name of the parameter data block.

- `int num_ints`: The number of ints in the data array.
- `int *iarray`: The pointer to the int data array.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

D.8.21 replace_param_block_longs

This routine replaces a given parameter data block of type `long`. The block `MUST` exist.

Calling format:

```
replace_param_block_longs(&(phase->parameters),
                          block_name,
                          num_longs,
                          larray);
```

Function Arguments:

- `phase->parameters`: The pointer to the parameter data for the phase which is returned. This pointer will change if the block is successfully replaced.
- `char *block_name`: The name of the parameter data block.
- `int num_longs`: The number of longs in the data array.
- `long *larray`: The pointer to the long data array.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

D.8.22 replace_param_block_floats

This routine replaces a given parameter data block of type `float`. The block `MUST` exist.

Calling format:

```
replace_param_block_floats(&(phase->parameters),
                           block_name,
                           num_floats,
                           farray);
```

Function Arguments:

- `phase->parameters`: The pointer to the parameter data for the phase which is returned. This pointer will change if the block is successfully replaced.
- `char *block_name`: The name of the parameter data block.
- `int num_floats`: The number of floats in the data array.
- `char *farray`: The pointer to the float data array.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

D.8.23 replace_param_block_doubles

This routine replaces a given parameter data block of type `double`. The block `MUST` exist.

Calling format:

```
replace_param_block_doubles(&(phase->parameters),
                            block_name,
                            num_doubles,
                            darray);
```

Function Arguments:

- `phase->parameters`: The pointer to the parameter data for the phase which is returned. This pointer will change if the block is successfully replaced.
- `char *block_name`: The name of the parameter data block.
- `int num_doubles`: The number of doubles in the data array.
- `char *darray`: The pointer to the double data array.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

D.8.24 replace_param_block_strings

This routine replaces a given parameter data block which is made up of an array of strings. The block `MUST` exist.

Calling format:

```
replace_param_block_strings(&(phase->parameters),
                            block_name,
                            num_strings,
                            char *strings[]);
```

Function Arguments:

- `phase->parameters`: The pointer to the parameter data for the phase. This pointer will change if the block is successfully replaced.
- `char *block_name`: The name of the parameter data block.
- `int num_strings`: The number of strings in the data array.
- `char *strings[]`: The pointer to the strings array.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

D.8.25 copy_param_block

This routine copies an existing parameter data block to a new parameter data block. The new block may or may not exist. It is the equivalent of getting the block given by the first name and all its attributes and creating the block with the second name using those attributes.

Calling format:

```
int copy_param_block(
    &(phase->parameters)
    name,
    new_name)
```

Function Arguments:

- `phase->parameters`: The pointer to the parameter data for the phase. This pointer will change if a the new block is successfully created.
- `char *name`: The name of the parameter data block to copy.
- `int new_name`: The name of the block to create.

Return value:
The function returns 1 if the block does not exist, otherwise it returns 0.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

D.8.26 delete_param_block

This routine deletes a parameter data block if it exists.

Calling format:

```
void delete_param_block(
     &(phase->parameters)
     &new_size,
     name)
```

Function Arguments:

- `phase->parameters`: The pointer to the parameter data for the phase. This pointer will change if the new block is successfully deleted.
- `int *new_size`: The address of an int which receives the new size of the parameter data.
- `int name`: The name of the block to delete.

Note that after this routine is called, any pointers to parameter data blocks will be invalid (except those returned by `getblock_data_copy`). (See Sec. 2.11.4.9 for more discussion of this point.)

D.8.27 Parameter data block flags

All parameter data block creation routines take an integer mask value that defines how the parameter data block is used. The mask value should be created from the logical OR of the following values.

- `FOR_EMPTY_MASK`: Use this value to indicate that no flags are set. This is the appropriate value for parameter data blocks that are used simply as temporary storage within the waveform server process.
- `FORUSER_MASK`: When parameter data are requested by the user interface, this block should be provided. If this bit is not set, this is an indication that the data in the parameter data block are not available to be edited by the user.
- `FORARCHIVE_MASK`: Indicate that the parameter data block should be included in the PCS setup that is archived for each shot.
- `FORHOSTVCPUA_MASK`, `FORHOSTVCPUB_MASK`, `FORHOSTVCPUC_MASK`, `FORHOSTVCPUD_MASK`, `FORHOSTVCPUE_MASK`, `FORHOSTVCPUF_MASK`, `FORHOSTVCPUG_MASK`, `FORHOSTVCPUH_MASK`: Each of these bits indicates that the parameter data block should be copied to one of the real time computers. The letters A, B,... specify the virtual CPU numbers for the category.
- `FORHOSTPARAMA_MASK`, `FORHOSTPARAMB_MASK`, `FORHOSTPARAMC_MASK`, `FORHOSTPARAMD_MASK`, `FORHOSTPARAME_MASK`, `FORHOSTPARAMF_MASK`, `FORHOSTPARAMG_MASK`, `FORHOSTPARAMH_MASK`: Each of these bits indicates that the parameter data block should be copied to one of the host_cpu processes. The letters A, B,... specify the virtual CPU numbers for the category.
- `NORESTORE_MASK`: Indicates that even if a parameter data block with the same name as the block being created exists in an archived PCS setup, the block should not be restored when the archived setup is restored. This flag controls the restore of individual parameter data blocks. So, it can be used to control a parameter data block that is part of a static data item. Even if the remainder of the parameter data blocks belonging to the static data item are to be restored, this flag can prevent an individual block from ever being restored. See Sec. 2.11.5.2 for more information.
- `SCRATCHDATA_MASK`: Indicates that the block is a scratch data block (Sec. 2.11.10). The block has no associated data. When the block is created on the real time processor it is filled with zeros. The real time algorithm code can use the block for any scratch storage purpose. This flag has no meaning for a parameter data block that isn't created on a real time processor. Because no data need to be provided to fill the block, the argument to the function that creates the parameter data block that usually points to the data is specified as a null pointer.
- `EXTERNALDATA_MASK`: Use this value to indicate that the data in the parameter data block should be stored in an "parameter data block external file." See Sec. 2.17.4 for a discussion of this feature.