Plasma Control System Software Architecture Guide (preliminary draft)

January 21, 2011

Contents

Obtaining this document

This document is available online with a WWW browser at:
`http://web.gat.com/pcs/architecture/architecture.html`.
The Postscript file for printing can be obtained at:
`https://diii-d.gat.com/DIII-D/comp/analysis/pcs/architecture/architecture.ps`.
The pdf file for printing can be obtained at:
`https://diii-d.gat.com/DIII-D/comp/analysis/pcs/architecture.pdf`.

1 Introduction

This manual provides information on the plasma control system (PCS) software at the level of detail appropriate for a system programmer. Those interested in modifying the software architecture or porting the PCS software to new computer systems will benefit from reading this guide. Those interested only in adding control algorithms to the system should read the `Application Programmer's Guide` instead. Readers of this manual should also be familiar with the content of the Application Programmer's Guide, particularly the section "Background information for application code authors" because the necessary background given there is not repeated here.

The reader will detect quickly that this manual is far from complete. Sections will be added and updated at what will probably be irregular intervals.

2 Limitations

The number of realtime computers is limited to a maximum of 24. This is due to the parameter data block functions which all have an argument called `flags` which is of type `int` so is limited to 32 bits of information. This argument is a bit mask. Eight of the 32 bits are used for general information like whether the parameter data block is archived or whether it is sent to the user interface. The other 24 bits are used to indicate whether the block goes on the host_cpu or realtime computers.

The original usage of these bits limited the number of realtime computers to 12 because 12 bits were used to indicate the block should go on a host_cpu computer and the other 12 bits were used to indicate the block should go on a realtime computer. This is modified for installations where the number of computers is greater than 12 by using two of the eight general bits, one to indicate a host_cpu computer and one to indicate a realtime computer, so that the 24 other bits can indicate which computer. This means that a parameter data block cannot be put only on virtual host_cpu

computer A and only on virtual realtime computer B. If the block must go on both types of computers, it must go on the same virtual ones, in this case, both A and B.

3 The waveform server

3.1 Structure of the waveform server code

The code for the waveform server is divided into three portions: infrastructure, installation-specific, and application specific.

1. The following files are part of the infrastructure:

   (a) `wavemain.c` contains the main routine for the waveform server process. It includes `control.h`.

   (b) `control.h` is the file that "gathers" all the installation code together into the waveform server process. It also defines some global variables used in the waveform server.

   (c) `waveserver.c` contains the core routines for the waveform server.

   (d) `get_vector_maps.c` contains routines used to print out information about data compiled into the waveform server. For instance, the indices of elements of the target vector or point names used for the data archived in the shot file.

   (e) `return_data.c` contains routines that are used to return data from the database maintained by the waveform server to a requesting process.

   (f) `waverestore.c` contains routines used to "restore" a shot setup. That is, these routines read a shot setup from either a future shot file or from an archived setup from a previous shot. Either all or part of a shot setup can be restored.

   (g) `targetvectors.c` contains the routines that manage the database of "processed" data within the waveform server. Processed data are created from raw data and are usually copied to a real time cpu for use during a shot.

   (h) `rtmemory.c` contains the routines that precalculate the arrangement of the memory in the real time cpu according to what will be required by the shot setup held in the waveform server database.

   (i) `parameterdata.c` and `waveparams.c` contain the standard routines used to manage the block structured "parameter" data (also called "static" data) defined for a given shot phase.

   (j) A group of files contains the low level code that manages the structure of the file (or shot database entry) that records the setup for a given shot.

These files handle both archiving and restoring of a shot setup. These routines are called by the routines in `waverestore.c`. (This group of files also contains routines used to archive data acquired and calculated during a shot. These routines are used by the host_cpu processes.) The files in this group are:

- `ckstatus.c`
- `clear_setup_data.c`
- `cstring_to_field.c`
- `parcel_setup_data.c`
- `read_shotsetup.c`
- `rstcom.c`
- `setup_driver.c`
- `setup_read_handler.c`
- `setup_write_handler.c`
- `unparcel_setup_data.c`

Note that there is an even lower level set of files containing the installation-specific routines for writing into the shot database file. For DIII-D, these routines are part of the general library `libd3`.

2. The following files are installation specific. These files provide the various definitions that are specific to a particular control system installation but which are not specific to any control algorithm.

   (a) `installdefs.h` contains macro definitions needed by both the infrastructure and the installation. The demo version of this file contains all the required definitions. This file is included in `control.h`.

   (b) `input_data.h` contains macro definitions used for raw data. By convention, this file is used for the raw data and digital data pointnames. It is usually included in `installdefs.h`.

   (c) `pcsshotfile.c` contains low level routines for archiving the shot setup, archiving data after the shot, and restoring data for the simserver. These routines are the direct interface to the local shot database routines.

   (d) `categories.h` is the starting place for including categories. This file must exist and is included multiple times in `control.h`.

(e) `cpus.h` is the starting place for including the real time cpu information. This file must exist and is included multiple times in `control.h`. It includes the definitions necessary for each real time cpu used in the PCS. If the program `common/config.c` is used (highly recommended), then the file it creates `definitions.h` gets included here.

(f) `waveinstall.h` contains installation-specific code for the `waveserver` program. Currently, only the function `waverestore_install` which handles specific restore issues is required. This file must exist and is included in `wavemain.c`.

3. All files with a name ending in `_algorithms.h` are files that include the algorithm master files for a particular category.

4. All files with a name ending in `_master.h` are files that define either a real time cpu, a control category, or a control algorithm. Some of these master files will include code from other category or algorithm specific files. There may be a few other files containing code or definitions that have been separated out for some reason. Examples are `filter_programs.c` which contains DIII-D specific routines and `shape_includes.h` which contain common code that gets included in multiple algorithms.

3.2 Defining the dma_region

In general, in order to do feedback control it is necessary to acquire data from sensors on the controlled system. The `waveserver` process can set aside memory in order to store the acquired data away, and the data can also be archived by the `host_cpu` process after the shot.

The PCS infrastructure can handle an unlimited number of channels of data from A/D converter modules. The digital data can be set to 1, 2, or 4 bytes per value by defining a `dma_region` descriptor. By default, this descriptor is defined as follows:

```
/*
Descriptor for a complete dma\_region. The count of floats is filled
in when the descriptor is used.
*/
#define GEN_dma_region \
DSTART(                                              D_dma_region)\
DGEN  ( float rawdata[dummy_NUMCHANNELS];            ,FLOATTYPE,1)\
```

```
DGEN  ( unsigned int times[2];                    ,INTTYPE,  2)\
DGEN  ( unsigned int diodata[1];                  ,INTTYPE,  1)\
DGEN  ( unsigned int dummy;                       ,INTTYPE,  1)\
DEND
GEN_dma_region
```

The installation can define a different descriptor as long as it has the same three "pieces" as above:

- `rawdata`: An array of NUMCHANNELS of any of 4 types: `char`, `short`, `int`, or `float`. The value of NUMCHANNELS can be zero.
- `times`: Must be `unsigned int` and be at least 2 in size.
- `diodata`: An array of any of 4 types: `char`, `short`, `int`, or `float`. The descriptor size can be set to zero. This field is used for digital data and each bit will be archived as a separate pointname.

To pass this new descriptor to the infrastructure, the installation code (usually the `acquisition` category) must call the function `fill_dma_region_constants(D_install_dma_region, rtcpu)` for each real time processor (rtcpu is a value from 0 to rtcpu_count-1). This will load the sizes of these three fields into each real time processor. The installation can add fields or times as desired, but the infrastructure will not do anything with them.

The amount of memory set aside for the dma_region for a particular real time processor is the descriptor size (the default is 48 bytes) multiplied by the number of samples set in the `operating setup data` parameter data block for that processor. Extra space is added equal to two samples times the descriptor size plus 24 bytes which was required by the SuperCard version of the DIII-D PCS.

The dma_region exists only to make it easier for the installation code to acquire data on the real time processors. The `host_cpu` process will archive data if the sizes of the `rawdata` and/or `diodata` are non-zero.

An example of an installation version of the dma_region and the code needed to set it up follows.

```
#ifdef CONTROLDEF
/*
Descriptor for the installation version of the dma_region.
This differs from the default structure because the rawdata
```

```
and the diodata are both shorts instead of ints.  The
sizes will get set later in the parameters function.
*/
#define NUM_DIODATA 4
#define GEN_install_dma_region \
DSTART(                                          D_install_dma_region)\
DGEN  ( short rawdata[1];                        ,SHORTTYPE,1)\
DGEN  ( unsigned int times[2];                   ,INTTYPE,  2)\
DGEN  ( unsigned short diodata[NUM_DIODATA];     ,SHORTTYPE,NUM_DIODATA)\
DEND
GEN_install_dma_region
#endif

In the baselinedata_parameters function:
/*
Define the installation version of the DMA_region.
The number of channels and diodata could be different on each cpu.
*/
{
static GEN_install_dma_region /* must be declared static */

    for(j=0; j<rtcpu_count; j++)
    {
        D_install_dma_region[0].size = NUMCHANNELS; /* number of rawdata */
        D_install_dma_region[2].size = NUM_DIODATA; /* number of diodata */
        fill_dma_region_constants(D_install_dma_region,j);
    }
}
```

3.3 Defining the install_buffer

In general, when multiple processors are used in the PCS, there requires some communication between them. What data needs to be sent is wholly up to the installation. The PCS provides the communication buffer for use during the shot, but data might need to be passed `before` the shot. Or the communication buffer may not suit the needs of inter-processor communication.

Therefore, the infrastructure allows space to be set aside for a buffer that is completely defined by the installation called the `install_buffer`. The waveserver

needs to know the descriptor for this buffer so it can set aside the required space in the memory of each real time processor. An algorithm that is always running in the PCS (like in the acquisition or system category) needs to define this structure. By default, the install_buffer is set to NULL.

An example of an installation version of the install_buffer and the code needed to set it up follows.

```
#ifdef CONTROLDEF
/*
Structure of installation buffer.
*/
#define GEN_install_buffer \
SDSTART(struct install_buffer {     ,D_install_buffer    )\
DGEN   (  int cpu_status[16];       ,INTTYPE,16          )\
                /* used to synchronize cpus */\
SDEND  (                      };                         )
GEN_install_buffer
#endif

In the baselinedata_parameters function:
/*
Set the infrastructure descriptor for the install_buffer.
*/
{
extern STRUCT_DESCRIPTORS *install_buffer_descriptor;
static GEN_install_buffer /* must be declared static */

    install_buffer_descriptor = D_install_buffer;
}
```

4 The user interface

4.1 The logo window

4.1.1 Creating the logo file

The PCS "logo window" has a bit map graphics widget that displays a logo image. This image can be created specifically for each PCS installation. The image is in

an X windows bit map file called `logoimage.bm`. This file can be created using any convenient tool. For instance, the X windows drawing program `xfig` will create output files in the correct format. The program `xv` is also useful. Here is a brief set of steps for using `xv`.

1. Capture (or create) an image to use. A JPG format is suitable but other formats can be used as well.
2. Start up `xv`. Right click on the start-up window and a GUI will come up.
3. Load the image file.
4. Modify the size of the image using `xv` if desired.
5. Save the image in X11 Bitmap format with the name `logoimage.bm`.

5 Communicating PCS information and status

5.1 The message server (msgserver program)

The message server is a program that runs in the background and gets started when the waveform server and other PCS processes get started.

Messages are sent from each of the other PCS processes to the message server. These messages may be error messages, information messages, or just a heartbeat to indicate that a process is still alive and kicking.

These messages are then passed to all processes which have logged themselves as clients. These clients would include the user interface server (uiserver) (see Sec. 5.2) processes that are created whenever a view log is started in a PCS user interface. Other clients could be external programs that need to know what the status is of the PCS. Any such program would need to parse the messages to extract pertinent data.

5.2 The user interface server (uiserver program)

The user interface server (or uiserver) runs as a background process whenever a view log is started. This program logs itself to the message server. Whenever there are messages buffered in the message server, they are sent to each uiserver client. The uiserver clients then pass these messages to the IDL user interface so the messages can be put into the view log. The IDL user interface requests information from the uiserver every two seconds.

The user interface server also receives messages from the waveform server whenever a change is made to any data items, phases, phase sequences, etc. These messages are also passed to the IDL user interface so they can be interpreted. If the user is affected by the change, a message specifying what the change was is written to the plot region of the user interface. If the current data item is modified, then the "apply" and "cancel" buttons will change to indicate that the current data item was modified.

The UI server has a maximum number of messages that it holds in a circular buffer. The current size of this buffer is 600 messages.

Note that the uiserver runs on the same host as the IDL user interface so the regular communication with IDL is not over the network.

6 Accommodating a mixture of CPU types

Here is a list of the places where customization is needed in order to allow the various processes of the PCS to run on a mixture of processor types.

1. Conversion of data formats is done using routines in `convert_functions.c`. All routines should use these functions so that new processor types can be added simply by updating these functions.

2. The data type chosen for the `typedef` values `processor_type_mask` and `message_length_value` is an integer that has the same length on all processor types. Macros to convert these two data types between network (big endian) and local format are specially defined.

3. In `fill_sc_rtheap.c`, the function `phase_change` determines whether 1 or 2 change list entries are required in order to move a pointer into a vector element. Also, in `rtmemory.c`, the function `phase_changelist_count` must also determine whether 1 or 2 change list entries are required in order to move a pointer into a vector element.

4. Variables of type `int` are used in coding wherever an integer is needed for consistency of variable size. However, this isn't required. Variables of type `long` could be used as long as the possible variation in size of the variable between the various processor types doesn't introduce problems.

5. In several places a very large integer value is required. Presently, the macro used is `MAXINT` which is the maximum possible positive value that can be held in a signed `int`. This value is defined in

`<values.h>`

. This works because presently the `int` is the same size on all processor types that are used. If this is ever not the case, this macro will need to be changed so that the large integer value is the same independent of the type of processor on which the code executes.

6. The type of processor is given by the bit mask data type `processor_type_mask`. The value for this mask is determined by the routine `get_processor` in `get_processor.c`.

7. The data type `char` is a single byte on all machines. This is likely to stay this way, I suppose. A redefinition of this C language data type would be required to change this. So, the operation to convert the data format for the `char` data type is a simple copy and it seems useless to insist that the `convert_data` function be called to convert character data. Using this function, though, provides consistency in the way data conversion is treated. We have ended up being inconsistent about converting character data. In cases where the character data are embedded into a structure, conversion of the structure will include conversion of the character data. In some cases, though, messages sent between processes include a block of character data. In this case, sometimes we follow the formality of calling `convert_data` and sometimes no "conversion" is done. If we tried to change the PCS sometime in the future to use an international character set (2 byte characters) this would require using different code for handling character strings. Even if we included the call to `convert_data` for all character strings in the present code this wouldn't really help in this character set conversion task.

8. Messages sent between processes usually consist of various pieces of data which are often first collected into one block of bytes and then transmitted. Even if data are transmitted in several blocks, the data are always received as one big block of bytes. This can present 2 types of problem.

   (a) When copying the data into one large block of bytes, there could be a problem with alignment. An attempt could be made to copy a data type into a location in the message which isn't properly aligned because of the amount of data preceding it in the message. For instance, if a character string is at the beginning of the message and it is followed by an integer, this could cause an alignment problem when copying the integer if the string has arbitrary length. This problem can be handled by defining a

single structure to hold the complete message, or by ensuring that data types with the most severe alignment are located at the beginning of the message.

(b) When the message is received, its data format must be converted. Often this big block of bytes isn't converted by constructing a single descriptor and doing the conversion with one call to `convert_data`. For instance, if a message consists of a few integers followed by a character string, the integers are converted separately from the character string. This problem can be avoided by defining a single descriptor for the message and converting the entire message with one call to `convert_data`. Or, if the data with the most severe alignment are located at the beginning of the message there should be no problem.

The designs of the various message formats vary greatly and we haven't followed a consistent method to avoid these problems.

9. The IDL routines that are used to access the S file need definitions of the various structures contained in the real time processor's memory. These definitions need to vary depending on the type of processor that created the file and the type of processor reading the file. These IDL structure definitions should be generated automatically in order to keep them consistent with the C code definitions, but at present these definitions are maintained by hand.

7 Interprocess communication

The installation-specific file `rtcipc.h` has the functions that are required for communication between real time processors. These routines must also handle "stand-alone" mode. It is left up to the installation to fill in the details. For example, the DIII-D PCS uses a Myrinet network so these functions interface to the Myrinet library of functions. Another example would be if all real time processors were in the same computer box, shared memory regions could be used for processor to processor communication. Or if a network of computers all had reflective memory, these functions would need to know how to map and access that memory so that one computer could communicate with another.

The following is a brief summary of the interprocessor communication functions.

- `rtcipc_write`: Write into a buffer on any of the PCS processors at an arbitrary location in the PCS memory heap.

- `rtcipc_write_commbuff`: Copy an array of values to a specified location in the communication vector on any of the PCS processors.
- `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.
- `rtcipc_write_install_buffer`: Copy data to a general installation-specific location on any of the PCS processors.
- `rtcipc_write_timetostopflag`: Write a value to the timetostop_flag on a given PCS processor.
- `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.
- `rtcipc_write_paramdata`: Write data to a parameter data block on a given PCS processor.
- `rtcipc_write_rtmessage`: This is the most general function that copies data from one processor to another.

These functions are described more in the Application Programmer's Guide, as is the "real time message facility" which is a set of higher level standardized routines that use the first and last functions listed above.

8 The PCS shot cycle

8.1 Introduction

Most of the time the PCS is idle. During this idle time it collects data on the shot setup from the operators and waits for the signal to start a shot cycle. When this signal is received the PCS runs through a complete procedure to execute a shot cycle. In this section a summary of this PCS shot cycle procedure is given. Many of the processes that make up the PCS participate in the shot cycle. This section provides an overview of the interaction between the PCS processes. The sections on the individual processes provide more details.

The shot cycle can be divided into three portions.

Setup: In the first part of a shot cycle, the host computer does most of the work. Data from the waveform server are collected for each real time cpu and arranged in the memory of the cpus.

Execution: The second part of the shot cycle is when the tokamak discharge is actually executed and plasma is created. The real time computers do the work in this part of the cycle except for some incidental, non-real-time work by the host computer.

Cleanup: In the final part of the shot cycle the PCS cleans up after the discharge. This involves, primarily, archiving the data acquired during the shot. Most of the work here is done by the host computer.

In the first and last parts of the discharge cycle when the host computer is doing most of the work, the PCS work is coordinated by the lockout server. The lockout server keeps track of the progress of each of the PCS processes and indicates when to proceed to the next step in the shot cycle. The lockout server receives notice of "events" and uses its programmed logic to determine what to do based on the events that occur. The lockout server receives its event information from two sources: messages from other processes and, optionally, by monitoring a set of hardware triggers.

Part of the "Execution" portion of the shot cycle is standardized by the PCS infrastructure code. However, most of the procedures during real time are determined by the application specific control algorithms and the way that the operator has programmed the shot setup.

The procedure in each of the three portions of the shot cycle is described in the following sections.

8.2 Shot Setup

8.2.1 First lockout

The lockout server begins a new shot cycle either when it receives a message of type `SET_FIRSTLOCK` or detects that the hardware "get ready for shot" bit is set. At the start of a new shot cycle, the waveform server is informed (by a `LOCKOUT_IN` message from the lockout server) that changes are "locked out" (no more changes should be accepted for the shot setup). Then the lockout server sets its local variable that stores the PCS state to `FIRSTLOCKOUT`.

The `LOCKOUT_IN` message received by the waveform server is placed in its work queue. The waveform server continues to process messages on the queue until the lockout message is found. At that point the waveform server stops processing messages from the queue, but will still add to the queue if input is received from a user interface. These messages sitting on the queue are not processed until the waveform server receives an "unlock" message.

The host real time processes periodically poll the lockout server to find out the PCS state. When a change in the state to `FIRSTLOCKOUT` is detected the host real time processes begin preparation for the discharge (see Sec. 10). Before the host real time processes can begin loading data for the discharge they must wait for the waveform server to complete processing of all messages that were received before lockout. The host real time processes poll the waveform server to obtain its lockout flag. When the proper lockout flag is detected, the host real time processes prepare their associated cpus for the discharge and then notify the lockout server that shot preparation is complete. Preparation for the discharge includes loading the memory of the real time cpu and executing application specific initialization routines for the control algorithms chosen by the operator.

The lockout server tracks the progress of the host real time processes. When all of the host real time processes have completed the work for shot preparation, the PCS is ready to move to the "final lockout" state.

8.2.2 Unlocking

Before the final lockout, the PCS operator can issue an "unlock" command by sending the `SET_UNLOCK` message to the lockout server. The lockout server sends a corresponding unlock command to the waveform server. The waveform server then begins processing messages on its work queue again. This allows all messages sent to the waveform server to be processed until the "relock" command is received (a `SET_RELOCK` message to the lockout server). After an unlock/relock operation, all of the prepara-

tion for the discharge by the host real time processes must be done again. In this way, any last second changes in the shot setup that the operator needed to make can be incorporated into the data loaded into the memory of the real time cpus. Note that the shot setup work by the host real time processes is not interrupted by an unlock message. An unlock message received while any of the host real time processes have not yet completed shot setup is flagged as pending.

Because the host real time processes are only clients which receive their PCS state information by polling the lockout server, there is an interval between polls in which the lockout server could make several state changes, e.g. unlock/relock could occur more than once. The host real time processes cannot track this rapid activity. Instead, a "lockout counter" is maintained by the lockout server. This counter is used by the host real time process to determine whether it has caught up with all of the necessary work. When waiting for the waveform server to process all messages received prior to the lockout, the host real time process tracks the lockout count value for the lockout message processed by the waveform server. In this way it detects whether there are more lockout messages yet to be processed by the waveform server (e.g. in the case of a rapid sequence of unlock/relock commands). Also, if the lockout count increases after the host real time process has completed shot setup, it knows that it must do the shot setup again.

8.2.3 Final lockout

The lockout server changes to the `FINALLOCKOUT` state either after receiving a message of type `SET_FINALLOCK` or after detecting that the final lockout hardware trigger bit has been set. In the final lockout state no more shot setup changes can be made by the operator.

The host real time processes detect the transition to the `FINALLOCKOUT` state and responds by first sending a `REALTIME_ROUTINE_STARTING` message to the lockout server and then starting the real time code on the real time cpus. The real time cpus can, at this point, do some application specific initialization code that is defined by the control algorithms programmed to be used during the discharge.

At this point the lockout server checks each of the watchdog timer bits (see Sec. 12) to be certain that the corresponding real time cpu is operating properly. The "PCS ready for shot" hardware bit is then set to indicate to the main tokamak control computer that the PCS is ready.

8.2.4 Aborting the shot

At any time between the change to the lockout state and the start of the real time control cycles the shot can be aborted. This means that setup for this shot is stopped and the PCS returns to the `NULL` state.

The lockout server can abort the shot at any time during the `FIRSTLOCKOUT` state. This occurs if either an `ABORT_SHOT` message is received or if the hardware abort bit is set. In either of these cases, the lockout server sets the PCS state to `ABORTED`. In response, the host real time processes each send a `SHOT_FINISHED_AND_ABORTED` message to the lockout server. This message indicates that the host real time process has recognized that the shot is complete. When all host real time processes have indicated shot completion in this manner, the lockout server follows the shot cleanup procedure.

This method is followed because it is possible that, depending on when the abort command arrives, one or more of the real time cpus might have already started the real time code. So, this allows for a mixture of shot completion status from the real time cpus (shot completed normally or shot aborted). The lockout server treats the shot as having been aborted if any of the real time cpus report that the shot was aborted.

The shot may also be aborted if one or more of the host real time processes is unable to properly complete setup for the shot. In this case, the host real time process sends an `ABORT_SHOT` message to the lockout server during the shot setup procedure. The host real time process then tells the lockout server that it has completed setup. When the lockout server responds by switching to the `ABORTED` state, the host real time process follows the normal procedure for aborting the shot.

8.3 Shot Execution

Once the real time computers begin executing their real time routines, the host real time processes and the other PCS processes have a short time during the discharge when they don't have much to do other than to wait for the real time computers to report that the shot is complete. During this pause, the first steps in creating the database archive file for the shot are taken. The file is created and the waveform server is instructed to write its database entry with a complete description of the setup for the shot.

When the real time code is started on the real time cpus, the first thing done is to simply wait for a hardware trigger that indicates that the shot has started. This trigger is synchronous with other timing signals generated by the main tokamak time

system. For the DIII-D PCS this trigger is called `6A` and comes at $t = -10$ secs. This trigger is also used to clear the counter that the PCS uses to track the time during a shot. Thus, after this trigger arrives the PCS knows that its time counters contain valid values. The real time cpu then monitors the time until $t = -9$ s when it calls the main routine that runs the control cycle. The 1 s pause here is simply to account for any delays in recognizing that the synchronous trigger has arrived. Until $t = -9$ s the shot can be aborted.

The real time cpus then run the shot cycle routine continuously until it is recognized that the shot is over. On the DIII-D PCS, the control system simply stops executing the control cycle at a time programmed by the operator. This is simply a time somewhat after the plasma is programmed to end.

8.4 Shot Cleanup

At the end of the discharge, the shot cycle routine returns and the real time cpus then execute any shot cleanup routines that are specific to the control algorithms that were programmed to be used during the shot. The real time cpu's routine then exits and then the host real time routine takes over again.

The host real time routines execute any shot cleanup code that is defined by the algorithms programmed to be used during the shot and then notifies the lockout server that particular real time cpu is finished with the discharge.

The lockout server waits until all real time cpus have completed the shot and then makes the transition to the `WRITINGFILES` state. While the PCS is in the `WRITINGFILES` state it writes the data from the real time cpus into the database archive file. One at a time, to avoid file access conflicts, the real time cpus are given permission to write to the archive file. In addition, the real time cpus optionally write the "s" file which contains an image of the area of the memory of the real time cpu used to store PCS data.

After all of the host real time cpus have completed archiving data, the lockout server returns the PCS to the `NULL` state and waits for the start of another shot.

9 The lockout server

Files used by the lockout server program (which is called lockserver).

Files used from the infrastructure:

- `lockmain.c` contains the main function which mostly just calls the `lockserver` function. The installation-specific file `lockinstall.h` is then included. This is

followed by all the default functions that are needed by the program. See below for a description of how these default functions can be replaced by installation-specific functions.

- `lockserver.c` contains the function `lockserver` which is the "guts" of the lockout server program.

Required installation-specific files:

- The file `lockinstall.h` contains any installation-dependent functions which do tasks differently than the default versions found in the file `lockmain.c`. Each of these default functions are compiled into the lockserver image unless a macro of the form `REPLACE_routinename`, e.g., `REPLACE_LS_GET_SHOT_NUMBER` is defined in `lockinstall.h`. At least one function needs to be included in `lockinstall.h` which correctly gets the shot number.

- `close_shotfile.c` contains low level routines for the `lockserver` to handle the end of shot archiving. (See 15.3).

10 The host real time process

Each of the real time computers in the PCS has, in addition to the code that runs in real time, a set of routines that provide the interface to the remainder of the PCS. These interface routines can run on the real time processor if the necessary networking software is available or the interface routines can run on a host processor that is closely coupled with the real time processor.

In the former version of the PCS, the interface routines ran on a host computer. There was a different process and executable from the real time process and executable, one for each of the real time computers. The executables were called `host_cpu%` where `%` indicates the physical cpu number. These separate processes are still used for the `stand alone` version of the PCS (`runsa`).

In the current version of the PCS, the host real time and the real time routines are compiled separately, but they are linked together into one executable called `pcs_rt_cpu%` where `%` indicates the physical cpu number.

The `host_cpu` code is written using the model that the process on the host can directly control the real time processor. It is assumed that the host computer can:

- Write or copy data into an area of memory on the real time computer.

- Determine the values for pointers to this data in the address space of the real time computer and include these pointers as part of the data copied into the real time computer's memory.
- Start a function running on the real time computer with an argument that is a pointer into the data that were written to the real time computer's memory.
- Synchronize with the operation of the real time cpu by waiting for the real time function to finish.

If the real time processor had the necessary networking support, then all of the interface routines and the real time code could run on the same processor and these conditions would be easily satisfied.

In the current version of the DIII-D system, the host process and the real time process are linked together so as to satisfy these conditions. However, to describe how these two parts work, we will speak separately of the `host_cpu` process and the `realtime` process.

The real time interface routines implement a client process in the client/server architecture of the PCS. This client process has the following small number of responsibilities.

1. Most of the time the process is essentially idle. Periodically, the `host_cpu` client polls the lockout server to determine the current PCS processing state. When a change in state is noted the process takes appropriate action.
2. The `host_cpu` process detects that a shot cycle is beginning from a change to the `FIRSTLOCKOUT` processing state. In response, the process communicates with the waveform server to load all of the data required on that particular real time cpu and arrange it in the memory of the real time processor (Sec. 10.4). When this is complete, the `host_cpu` process informs the lockout server that its real time cpu is ready for the shot.
3. When the change to the `FINALLOCKOUT` state is detected, the host real time process does what is necessary to start the main real time routine on the real time cpu. In the case of separate host and real time computers, this involves calling a host process function that can start a function running on the real time cpu.
4. The host process then waits while the real time cpu participates in controlling the discharge. When the real time cpu function completes the host process assumes that the discharge is complete.

5. After the discharge, each of the `host_cpu` processes is given permission, one at a time, to archive data in the shot database. By cycling through the processes one at a time during data archiving, conflicts in access to a single data file are avoided. During this phase the PCS is in the `WRITINGFILES` state.

6. During preparation for the discharge, there could be a transition to the `ABORT` state. This indicates that either the lockout server or a real time cpu has detected a command to terminate the shot cycle early. The `host_cpu` process responds by confirming recognition of the `ABORT` state to the lockout server. The PCS then passes through the `WRITINGFILES` state, with no files actually written, to reach the `NULL` state.

Each of these steps in the work of the `host_cpu` process will be discussed in more detail in the remainder of this section.

10.1 Structure of the `host_cpu` code

The source code for the `host_cpu` process is separated into 2 portions. Most of the code is in the infrastructure file `host_cpu.c`. Any code that is category or algorithm specific is added to the host real time process by including the code in the category or algorithm master file.

The file `host_cpu.c` contains all of the generic code that implements the client process and handles the real time cpu. The file `hostmain.c` contains the `main` function and is compiled multiple times, once for each `host_cpu` process that is created.

When `hostmain.c` is compiled, the installation-specific code is included. There is a set of installation-specific files that is required to implement tasks that depend on the type of real time processor in use. These functions are described in Sec. 10.2.

Also part of the host real time generic code is the infrastructure file `fill_sc_rtheap.c` ("fill supercard real time heap"). This file contains the code that is responsible for obtaining data from the waveform server and arranging it in the memory of the real time computer during setup for a discharge (Sec. 10.4).

The infrastructure file `writertfiles.c` contains most of the code for archiving data into the tokamak database from the real time processor.

The installation-specific file `pcsshotfile.c` contain additional code which is required for archiving data to the DIII-D database.

The installation-specific file `filter_programs.c`, contains the code for programming the anti-aliasing filters that are part of the DIII-D PCS.

10.2 Installation-specific functions

The `host_cpu` process requires a set of installation-specific functions that are called by the infrastructure code described in Sec. 10.1. These are functions that know how to perform tasks required by the infrastructure code that are specific to the particular real time processor hardware used in the PCS. Each of these required functions is described in this section. By convention, these functions are contained in the file `hostinstall.h` in the installation-specific code area.

10.2.1 host_install_x_hp

This function converts a pointer in the address space of a real time processor into a pointer in the address space of the `host_cpu` process. This task assumes that virtual address space in the `host_cpu` process has been mapped into memory located on the real time processor.

Calling format:

```
void *host_install_x_hp(void *imemory, void *address)
```

Arguments:

- `imemory`: An installation-specific pointer that is the value of the global variable `imemory` that is set by the initialization function `host_install_inits` (Sec. 10.2.3). This pointer, for instance, might indicate which of several real time processors is indicated by the `address` argument. If this argument isn't needed, it can be ignored.
- `address`: The pointer in the address space of the real time processor. This will usually be a pointer into the real time heap memory area.

Return value:
The function returns the value of the pointer in `address` after the address space conversion.

10.2.2 host_install_x_p

This function converts a pointer in the address space of the `host_cpu` process into a pointer in the address of a real time processor. This task assumes that virtual address space in the `host_cpu` process has been mapped into memory located on the real time

processor and that, if there are multiple real time processors to choose from, the value of the input pointer indicates which real time processor is referred to.

Calling format:

```
void *host_install_x_p(void *address)
```

Arguments:

- `address`: The pointer in the address space of the `host_cpu` process. This will usually be a pointer into the real time heap memory area.

Return value:
The function returns the value of the pointer in `address` after the address space conversion.

10.2.3 host_install_inits

This function performs any installation-specific initializations that are required when the `host_cpu` process is started. The expectation is that this function will attach the `host_cpu` process to a real time processor. This usually will involve mapping some virtual address space onto the memory located on the real time processor. As part of this mapping process, the function is required to initialize 3 global variables. The descriptions here of these variables are taken directly from the comments in the infrastructure file `hostmain.c`.

- `void *RTMEMSTART`: This is the pointer in the address space of the realtime process or processor to the start of the real time memory heap. This variable should not be accessed as a global in other functions. Routines in other files should use the function `host_heap_pointer` to get this pointer value.

- `void *HOSTRTMEMSTART`: This is the pointer in the address space of the `host_cpu` process to the start of the real time heap. This variable should not be used as a global in other files. Routines in other files should use the function `host_host_heap_pointer` to get this pointer value.

- `void *imemory`: In the stand alone mode of PCS operation or in the online mode on self-hosted real time processors, this is `NULL` to serve as a flag to indicate a self-hosted real time processor.

In the online version of the PCS for real time processors located on a separate CPU board, this pointer has a value that depends on the type of real time processor. Its usage is strictly for real time processor-specific functions. Typically this is as an argument to routines that convert between pointers in the address space of the real time process or processor and pointers in the address space of the `host_cpu` process or to identify a particular real time processor to the installation specific code.

If this value isn't needed for address space conversion routines or other installation specific functions, it should be set to be the pointer in the address space of the `host_cpu` process to the start of the real time heap (i.e. same as `HOSTRTMEMSTART`). It should never be NULL in this case.

Calling format:

```
void *host_install_inits()
```

Arguments:
None.
Return value:
The function returns the value of `imemory`.

10.2.4 host_install_init_rt_memory

This routine is responsible for setting all the bytes to be used in the real time memory heap to zero. It is called just after the PCS enters the `FIRSTLOCKOUT` state each time setup begins for a new shot.

Calling format:

```
int host_install_init_rt_memory(int *hssc_imemory,int count)
```

Arguments:

- `hssc_imemory`: Pointer in the address of the `host_cpu` process to the memory that should be initialized.
- `count`: The number of bytes that should be set to 0.

Return value:
The function returns a completion status, 0 indicating good, other than 0 indicates an error.

10.2.5 host_install_preshot_setups

This routine is called each time the PCS enters the `FIRSTLOCKOUT` state. This function performs any installation-specific shot setup that is required but which isn't configured as part of the shot setup code for a control category or a control algorithm. This function is called after all of the infrastructure code executes that performs the shot setup.

Calling format:

```
int host_install_preshot_setups(int software_test,int hardware_test)
```

Arguments:

- `software_test`: If nonzero, this flag indicates that the PCS is in software test mode.
- `hardware_test`:If nonzero, this flag indicates that the PCS is in hardware test mode.

Return value:
The function returns a completion status, 0 indicating good, other than 0 indicates an error.

10.2.6 host_install_start_realtime

This routine is responsible for starting up the real time function on the real time processor. This function is called when the PCS enters the `FINALLOCKOUT` state.

Calling format:

```
int host_install_start_realtime(struct rt_heap_misc *hssc_rtheap)
```

Arguments:

- `hssc_rtheap`: Pointer in the address space of the `host_cpu` process to the standard structure that describes the real time memory heap area.

Return value:
The function returns a completion status, 0 indicating good, other than 0 indicates an error.

10.2.7 host_install_stop_realtime

This function is called after `host_install_start_realtime`. It is responsible for waiting until the real time function completes execution on the real time processor.

Calling format:

```
int
host_install_stop_realtime(void *imemory, struct rt_heap_misc *hssc_rtheap)
```

Arguments:

- `imemory`: The installation-specific pointer returned by the function `host_install_inits` (Sec. 10.2.3). This argument might be used, for instance, to identify the real time processor where the real time function is executing.
- `hssc_rtheap`: Pointer in the address space of the `host_cpu` process to the standard structure that describes the real time memory heap area.

Return value:
The function returns a completion status, 0 indicating good, other than 0 indicates an error.

10.2.8 host_install_shot_cleanups

This routine is run at the end of each shot to execute any installation specific code that is required each time a shot is completed.

Calling format:

```
void host_install_shot_cleanups(struct rt_heap_misc *hssc_rtheap)
```

Arguments:

- `hssc_rtheap`: Pointer in the address space of the `host_cpu` process to the standard structure that describes the real time memory heap area.

Return value:
None.

10.2.9 host_install_get_function_ptr

This function returns a pointer to a function that executes on the real time processor. The returned pointer is to a location in the address space of the real time processor. This function is used to obtain values that will be written in real time into the function vector.

Normally this function would cause a routine to execute on the real time processor to return the required pointer. This routine would take advantage of the infrastructure function `get_function_ptr` that is located in the infrastructure file `realmain.c`. `get_function_ptr` has a table of names of all of the functions that could be listed in the function vector and a corresponding table of the pointers to these functions. `get_function_ptr` searches in the list of names for a match and returns the corresponding pointer. The list of names is generated during the PCS build procedure by executing the waveform server process in a mode that causes a list of all of the real time functions listed by the control algorithms to be generated. The code for the corresponding list of function pointers is also generated and the correct pointers are filled in when the real time processor code is built.

Calling format:

```
void *host_install_get_function_ptr(
void *imemory,
char *input_function_name,
int SOFTWARE_TEST_MODE)
```

Arguments:

- `imemory`: The installation-specific pointer returned by the function `host_install_inits` (Sec. 10.2.3). This argument might be used, for instance, to identify the real time processor where the real time function is executing.
- `input_function_name`: The name of the function for which the pointer is required.
- `SOFTWARE_TEST_MODE`: This argument is set to a nonzero value when the PCS executes in one of the software test modes. Otherwise this value is 0.

  This argument can be used in an installation-specific manner to substitute one function for another in software test mode. For example, presently in software test mode, the PCS always makes the following function substitutions.

- The function called `get_new_data_master` is replaced with the function `get_new_data_test_master`.
- The function called `get_new_data_slave` is replaced with the function `get_new_data_test_slave`.
- The function called `get_new_data_slave_no_dad` is replaced with the function `get_new_data_test_slave_no_dad`.

These substitutions are really made for historical reasons for the DIII-D PCS. However, this provides an example of what the `SOFTWARE_TEST_MODE` argument might be useful for. The idea was that in software test mode, all code that executes is identical to what would normally execute in real time except for the few functions used to obtain the input digitizer data. These function name substitutions replaced the functions that would obtain data directly from the digitizers with functions that obtain data from the simulation server. Probably a better procedure would be to simply test the software test mode flag during execution of the standard data acquisition function and call the appropriate code.

Return value:
The return value is the pointer to the desired function cast as a pointer to a `void`. (Admittedly, the function should really return a proper pointer to a function.)

10.2.10 host_install_update_the_state

This function causes the function `update_the_state` to be executed on the real time processor. This function returns when the execution on the real time processor is complete. The function `update_the_state` is called multiple times after the shot in order to reconstruct the target vector content for archiving.

Calling format:

```
void host_install_update_the_state(struct rt_heap_misc *hssc_rtheap)
```

Arguments:

- `hssc_rtheap`: Pointer in the address space of the `host_cpu` process to the standard structure that describes the real time memory heap area.

Return value:
None.

10.2.11 host_install_software_setups

This routine is called each time the PCS enters the `FIRSTLOCKOUT` state when the PCS is executing in a software test mode. It is responsible for running any installation specific routines to prepare the PCS for software test mode. For instance, this function would normally do any preparation that is necessary to allow the real time processor to request data from the simulation server.

Calling format:

```
int host_install_software_setups(struct rt_heap_misc *hssc_rtheap,
 int software_test_mode)
```

Arguments:

- `hssc_rtheap`: Pointer in the address space of the `host_cpu` process to the standard structure that describes the real time memory heap area.
- `software_test_mode`: The value of the software test mode flag.

Return value:
The function returns a completion status, 0 indicating good, other than 0 indicates an error.

10.2.12 host_install_write_simserver_data

This function is called by `host_read_data_set_simserver`, the routine that receives data from the simulation server. The data from the simulation server come in a standard format that must then be interpreted and written into the correct locations on the real time processor. Writing the data to the real time processor is the job of this function.

Calling format:

```
void host_install_write_simserver_data(
    struct rt_heap_misc *hssc_rtheap,
    struct sim_exchange_info_out *from_sim_info)
```

Arguments:

- `hssc_rtheap`: Pointer in the address space of the `host_cpu` process to the standard structure that describes the real time memory heap area.
- `from_sim_info`: The structure containing the data obtained from the simulation server.

Return value: None.

10.2.13 host_install_parse_command_line

This function parses the arguments on the command line used to start the `host_cpu` process. There are 7 standard values that this function would be expected to find on the command line. Normally, this function would call an infrastructure function to obtain these values as follows.

```
host_parse_command_line(argc,argv,runmode,
    lockhost, lockport, wavehost, waveport, msghost, msgport);
```

Then, `host_install_parse_command_line` can obtain any installation-specific values from the command line. These values would normally be stored in global variables declared in the file `hostinstall.h` where the code for the installation-specific functions for the `host_cpu` process is located.

Calling format:

```
void host_install_parse_command_line(int argc,char **argv,char *runmode,
     char *lockhost, int *lockport, char *wavehost, int *waveport,
     char *msghost,  int *msgport)
```

Arguments:

- `argc`: The standard value that passes the number of command line arguments.
- `argv`: The standard array of pointers to the command line argument strings.
- `runmode`: The PCS operations mode string obtained from the command line.
- `lockhost`: The name of the processor where the lockout server process is executing.

- `lockport`: The number of the input communications port for the lockout server process.
- `wavehost`: The name of the processor where the waveform server process is executing.
- `waveport`: The number of the input communications port for the waveform server process.
- `msghost`: The name of the processor where the message server process is executing.
- `msgport`: The number of the input communications port for the message server process.

Return value:
None.

10.3 Starting the shot cycle

The host real time process periodically sends a message to the lockout server requesting the present processing state of the PCS. Between shots this state is `NULL`.

When the shot is started, the state becomes `FIRSTLOCKOUT`. Each host real time process starts the shot by filling in the real time cpu memory, then calls any host specific category or algorithm initialization routines. The host real time process then sends a message back to the lockout server indicating that it is ready for the final lockout.

10.4 Filling the real time cpu memory

One of the biggest jobs of the host real time process is to load the memory of the real time processor in preparation for a discharge. A description of this process is given in this section.

10.4.1 Access to the real time cpu

In the DIII-D control system, the host real time process has access to the real time cpu because the two parts are linked together into one executable image. In a case where the two parts are separate, the host real time must access the memory of the real time cpu by mapping its virtual address space to the memory that is mapped to the

physical memory of the real time cpu. Thereafter, the host process can transparently access the memory of the real time cpu by using the pointer `imemory` which is a global variable for the host real time process.

The memory available could vary between the real time cpus. The maximum allowed amount of memory for each cpu is given by the macro definition `CPU_MEMORY_MAX` which is automatically generated by the `config.c` program in its output file `definitions.h`. This value is compiled into the `config.c` program through the macro `GEN_CPU_CONSTANTS` which is found in the installation specific file `config.h`. The order in which to load data into the real time cpu's memory is determined by the memory management code in the waveform server (Sec. 11). The memory management code ensures that the host process will not try to access data memory on the real time cpu outside the allocated area.

10.4.2 Filling memory

The process of loading the memory of the real time cpu begins with the line

```
hssc_rtheap = fill_sc_rtheap(cpu_num,&exfile_info,&setup_infra);
```

in `host_cpu.c`. This routine returns a pointer to the `rtheap` ("real time heap"), a structure which contains all of the information necessary to locate any other data on the real time cpu. The `rtheap` is the key for data access in real time. The pointer to this structure is an argument to all key real time routines.

The code that does the work of loading the memory of the real time cpu is located in the infrastructure file `fill_sc_rtheap.c`.

The first step in loading the memory is to initialize all of the memory to be used to hold data to 0. This ensures that there will be no uninitialized memory and that unused data locations will default to 0.

Then, the `fill_sc_rtheap.c` code steps through each of the data structures that are part of the real time code, obtains the required data from the waveform server, and copies the data into locations offset from `imemory`.

As long as the interface between the host process and the real time cpu's memory can be reduced to a single pointer in `imemory`, the code for loading data should be transparently portable to other real time hardware.

If it is not possible to use a single pointer to transparently access real time cpu memory from the host, an alternate approach is to `malloc` sufficient memory in the host address space, point to this location with `imemory`, and load all of the data there. Then, copy this entire block of data into the real time computer's memory using whatever method is available.

10.5 Executing the real time code

When the state becomes `FINALLOCKOUT`, each host real time process starts the real time code by instructing the real time process to execute the function `run_realtime` found in the infrastructure file `realmain.c`.

This is done directly for the `pcs_rt_cpu` process since the real time code is shared with the host real time code. In stand-alone mode, where the two are separate programs, the host real time process communicates with the real time process through a shared memory segment instructing it to run the function.

This function performs the following actions.

1. initializes all phase sequences for all categories.
2. executes each of the category and algorithm specific initialization functions.
3. executes the installation specific function `set_time_zero` which synchronizes all the real time processors.
4. executes the installation specific function `wait_for_start_signal` which waits for the master trigger to begin the shot. This function would turn off interrupts for the operating system and start toggling a watchdog if one is implemented.
5. once the trigger has arrived, executes the infrastructure function `time_critical` (see Sec. 13.2.5) which loops through the function vector calling each function until the value `rtheap->return_status` is non-zero.
6. executes the function `realtime_cleanup` which should perform functions to clean up after the shot such as zeroing any D/A outputs.
7. executes each of the category and algorithm specific cleanup functions.

10.6 Archiving data

When the shot is finished, each host real time process is given permission to archive data by the lockout server. This is done in series to avoid problems with synchronization that might occur if a single file is used for storage.

This behavior can be modified by specifying the full path of a file using the environment variable `ARCHIVE_ORDER_FILE` in the `pcs_config_info` file used for operations. This file, if found, should have the count of processors that can concurrently archive

on the first line. Only set this to a value greater than 1 if the system can handle simultaneous archiving. The second line in the file is the archiving order of the physical cpus. The default is numerical order.

When a host real time process gets permission to archive from the lockout server, it does the following actions.

1. executes each of the category and algorithm specific host archival functions. These functions can archive data in a nonstandard way.

2. executes the infrastructure function `write_archive_files` found in the file `writertfiles.c`. This function gathers up the data for each pointname and then executes the installation specific function `write_pcs_pointname` which writes the given pointname. This function is found in the installation specific file `pcs_shotfile.c`.

3. optionally, dumps the real time memory to the "S" file (see Sec. 14).

The saved data for the vector elements in the shape, fpcommand, intcommand, error, and pvector vectors are archived only if pointnames are provided in the algorithm code. Target vector elements are recomputed after the shot and then archived. Raw data and any digitial data are also archived.

If a pointname is shared between algorithms and multiple phases are used during the shot, then the data from all phases that use a pointname are combined together. It is possible to have a gap in the data where an algorithm that doesn't use the pointname runs.

Time pointnames are created separately from the data pointnames. The main time pointname for each processor is `TIMEPCSA<cpu>` where ¡cpu¿ is the processor number. This time pointname is used for all the vector elements except possibly the target vectors, as well as the raw data and digital data. The time pointname for the target vectors can be different if multiple phases are used through the shot and a pointname is not used in one of the phases. These time pointnames would be something like `TIMEPCS<number-1>` where ¡number¿ is the processor number multiplied by 100. This allows for 100 of these per processor.

For operations and test modes, if the file `write_archives.dat` exists in the data directory, then archiving can be turned off if this file contains a "0" as the only character. A value of "1" in the file turns archiving on. Archiving can be turned off while testing to speed up the shot cycle.

11 Management of memory on the real time computer

The memory on the PCS real time processor cannot be treated in the same way that the memory resource is handled in a typical program executing on a multitasking, virtual memory operating system. There are several key differences.

1. The operating system on the real time processor probably doesn't allow for expansion of the addressable virtual memory through paging to a swap file on a disk. Even if this were possible, the real time application probably can't tolerate the processing delays that would occur during disk operations. So the memory on the real time processor must be treated as having a fixed size.

2. The physical memory available on the real time processor could be limited to a relatively small amount.

3. It isn't practical to have the real time code simply allocate (e.g. with malloc) additional memory when it is required. The memory resource is limited on the real time processor and it is difficult to predict the maximum amount of memory that might be required when malloc is used in multiple locations in a program. So, the real time application could run out of memory in an unpredictable way. This would likely result in the failure of the real time task.

4. The amount of memory allocated for the stack will probably also be limited and fixed. So, if functions in the real time code make extensive use of local variables (which are usually placed on the stack), the real time application could fail in an unpredictable way because it runs out of stack space.

5. Depending on the particular real time processor system, the memory available for global and static variables could be limited.

In the PCS, all memory required by a control algorithm must be preallocated and the malloc function is never used in real time. This avoids the first three problems listed above. Preallocation of memory by a control algorithm is done by creating parameter data blocks to hold precalculated data and scratch parameter data blocks for temporary storage.

The last 2 problems in the list above are difficult to avoid except by careful programming practice. It is advisable to avoid the use of local variables that require significant amounts of storage space. Storage space in parameter data blocks should

be used instead. It is also advisable to avoid the use of global and static variables which introduce problems in addition to the amount of memory required. There is further discussion of this in the section "Global and static variables on the real time processor" in the "Application Programmer's Guide."

On each real time processor the PCS uses a single contiguous block of memory called the "PCS memory heap." This memory area holds all of the PCS real time data structures. The size of each data structure within the PCS memory heap is fixed during each discharge. There is no adjustment of data structure size or allocation of memory in real time. The PCS data structures in this memory heap area are accessed by code on the real time processor through the structure of type `rt_heap_misc`, commonly called `rtheap`, that is an argument to the real time functions as discussed in the "Application Programmer's Guide."

During setup for a discharge, the waveform server determines the layout and required size of the PCS memory heap for each of the real time processors. The required size will depend on the PCS setup, e.g. which algorithms are in use and how much memory is required for data archival.

The maximum amount of memory available on each real time processor for the PCS memory heap is set to a maximum value. (see Sec. 10.4) After the required size for the memory heap for the next discharge is determined, it is compared to the maximum available memory. If the required size of the heap is too large, a fatal raw data problem is registered. The PCS will not execute a discharge if a fatal raw data problem exists.

The method actually used to make available the maximum available memory for the PCS memory heap on the real time processor is installation dependent. This functionality is implemented in routines in `hostinstall.h`. Usually the location of the memory for the PCS memory heap is determined in the function `host_install_inits` that is called to initialize the `host_cpu` process. This function initializes three global variables in order to point the PCS to the appropriate memory area.

1. `void *RTMEMSTART`: This is the pointer in the address space of the realtime process or processor to the start of the PCS memory heap. This variable should not be accessed as a global in functions other than `host_install_inits`. Functions requiring this pointer call the function `host_heap_pointer`.

2. `void *HOSTRTMEMSTART`: This is the pointer in the address space of the `host_cpu` process to the start of the PCS memory heap. This variable should not be accessed as a global in functions other than `host_install_inits`. Functions requiring this pointer call the function `host_host_heap_pointer`.

3. `void *imemory`: In the stand alone mode of the PCS or in the online mode on self-hosted real time processors, this is NULL to serve as a flag to indicate a self-hosted real time processor.

   In the online version of the PCS for real time processors located on a separate CPU board, this pointer has a value that depends on the type of real time processor. Its usage is strictly for real time processor-specific functions. Typically this is used as an argument to routines that convert between pointers in the address space of the real time process or processor and pointers in the address space of the `host_cpu` process or to identify a particular real time processor to the installation specific code. If this value isn't needed for address space conversion routines or other installation specific functions, it should be set to be the pointer in the address space of the `host_cpu` process to the start of the real time heap (i.e. same as `HOSTRTMEMSTART`). It should never be NULL in this case.

In this description of these 3 variables, there is reference to a "self-hosted" real time processor versus a processor located on a separate CPU board. A self-hosted processor is one where the `host_cpu` process and the real time process run on the same processor. An example is the current DIII-D system with real time processors running Linux. The alternative is a system with a host processor and one or more real time processors attached by some expansion bus such as VME. An example is a Sky system. In both cases the assumption is made by the PCS infrastructure code that the address space on the real time processor holding the PCS memory heap is mapped into the address space of the `host_cpu` process. (This mapping would usually be done during the initialization performed by the function `host_install_inits`.) Code in the functions in `fill_sc_rtheap.c` that perform the preshot initialization of the PCS memory heap access it directly through pointers in the address space of the `host_cpu` process. It may be necessary to convert between pointers in the address space of the `host_cpu` process and the address space of the real time processor. Usually installation-specific functions will be available for this purpose. The PCS functions `host_install_x_hp` and `host_install_x_p` (defined in `hostinstall.h`) are generic interfaces to the installation-specific functions.

In stand alone mode operation of the PCS the memory heap for all of the processors is located in one large shareable memory area on the processor hosting the stand alone operation. The size of the shareable memory region is the sum of the maximum memory values for each processor.

A layout of the memory heap is actually calculated each time the work queue in the waveform server becomes empty. This preliminary memory heap size and layout

result can be listed at any time using the options in the user interface "diagnostics" menu. The option "memory usage summary" brings up a window with a listing of the size of the memory heap on each real time processor. Also listed is a breakdown of the total required memory into the amount of memory used for various purposes. The option "RT processor memory layout" brings up a window with a listing of the size and location of each of the blocks of memory allocated within the PCS memory heap. Note that the "preliminary" memory layout will be identical to the final layout used during the discharge if no PCS changes are made.

In the waveform server, the PCS memory heap layout is calculated in terms of byte offsets from the beginning of the PCS memory heap. The "RT processor memory layout" window shows the calculated offset and size for each PCS data structure. These calculations are performed by the functions in the infrastructure file `rtmemory.c`. The main function there is `fill_rtheap`. This function places most of the information about offsets into the PCS memory heap of the various structures into a structure of type `rt_heap_misc_offsets`. During setup for a discharge this structure is copied to the `host_cpu` process. This process converts the offset values into pointers during the process of loading the data into the memory heap on the real time processor. This memory initialization is performed by functions in the infrastructure file `fill_sc_rtheap`.

The PCS memory heap is divided into 2 portions: "cacheable" and "noncacheable." This attribute is specified by the code in `rtmemory.c` and is indicated for each data structure on the "RT processor memory layout" window. This attribute is somewhat historical but still may possibly be useful in some systems. Memory that is "cacheable" is intended to contain data that can be usefully brought into the real time processor's cache memory. Memory that is "noncacheable" contains data that is not desirable to have in cache. These are data that, for instance, are known to be needed only briefly when accessed or data in a large structure that is read completely and would overwrite everything in the cache. In some systems it may also be necessary to take care that the content of memory that can be written by more than one processor is never brought into cache. Some processors (e.g. the i860 used originally for the DIII-D PCS) have capability to control whether a data access is copied to cache and so can take advantage of this memory organization. The `rtmemory.c` functions don't do anything specifically to have a particular affect on the cache functionality. These routines simply divide the PCS memory heap into 2 portions and group the cacheable structures together in one portion and the other structures together in the other portion. Separating the 2 memory areas ensures that when cacheable memory is accessed, adjacent noncacheable memory is not accessed simultaneously by accident.

In order to set the cacheable preference the routine `set_cache_preference` can be called from a function in any algorithm which is always running (like the parameters function in the system algorithm). This function takes two arguments: a macro representing the section (like `PIDLOOKUP_PREFERENCE`) and an argument giving either the macro `NONCACHEABLE` or the macro `CACHEABLE`. All these macros can be found in the file `infra/serverdefs.h`. The only three sections that are NONCACHEABLE by default are `ADCOMBUFFER`, `DMABUFFER`, and the `TIMETOSTOP_FLAG`.

12 The watchdog timer

13 The real time procedure

The code that executes on the real time processor is compiled in an installation-specific manner. The basic methods to get code to execute on the real time processor will be strongly dependent on the real time platform and its associated software.

There is actually very little code for the real time processor that is part of the infrastructure.

13.1 Structure of the real time code

The main file for the real time code is the infrastructure file `realmain.c`. This file is compiled when the installation-specific code is built. `realmain.c` has `include` statements that bring in all of the real time code that is part of the control algorithms.

`realmain.c` can be compiled with or without the macro `NOMAIN` defined. This macro determines whether a function `main()` is created. This function is required for the executable used in the PCS stand alone mode. Depending on the type of real time processor, it may or may not be required for the code that actually executes on the real time processor.

`realmain.c` includes a function called time_critical which contains the main function that is executed during the real time control portion of the discharge. There are also other functions like `startup_state` and `update_the_state` and a number of others. All of these functions can be replaced by installation-specific code in the file `realinstall.h` by setting a macro beginning with `REPLACE_routinename`, e.g., `REPLACE_UPDATE_THE_STATE`.

The infrastructure file `phasetick_c.c` contains the code that executes at the end of each control cycle to determine whether a phase clock tick has elapsed and, if so,

to update the target vector and associated other real time data structures that are handled by the infrastructure.

13.2 Infrastructure functions

13.2.1 new_shot_phase_tick

The PCS uses two time intervals during the real time control: a "fine scale clock tick" which is usually 1 microsecond and is the units of `rtheap->currenttime` and `rtheap->update_time` and a "shot phase clock tick" which is the smallest interval between vertices and is set by the installation (1 millisecond is a good value). The shot phase clock tick should not be any smaller than the cycle time of the fastest processor.

This function is called whenever `rtheap->update_time` is greater than or equal to the next shot phase clock tick. Its purpose is to switch phase sequences if necessary, switch phases if necessary, and make changes to non-continuous target vector elements using the change list. If a phase switch is found to be neccessary, the `phasebegin` or `phaseenter` function is called.

13.2.2 new_continuous_target_c

The purpose of this function is to compute the new values of the continuous target vector elements from the values in the `mb` and `deltat` arrays. The values in `deltat` are then incremented.

Each of the "continuous" target vector elements is specified by a series of connected line segments. Each line segment is defined by specifying the slope and intercept of the line segment so that:

```
new target vector value = slope * deltat + intercept

(or y = m * x + b  in shorthand, thus the slope/intercept pair
                                 is an mb pair).
```

Here, `deltat` is the delta time from the vertex that defines the start of a line segment in units of shot phase clock ticks. `intercept` is, then, the Y value at that vertex.

This method of defining the line segment avoids having unreasonably large or small values for the intercept since the Y value of the vertex will always be a value

on the target vector waveform. The slope will never be exceptionally large since the time step is, at the least, one shot phase clock tick.

The slope/intercept (`mb`) array contains a pair of slope,intercept values for each continuous target vector element. The deltat array contains the time in shot phase clock ticks (e.g. 1 ms) since the previous vertex for each target vector element. The `deltat` array element is initialized to zero when a new vertex is reached. This routine increments deltat by 1.0 when a new target vector element is calculated. This method avoids having to subtract two possibly large numbers as would happen if deltat = (current time) - (time of vertex) was used in the computation.

13.2.3 startup_state

This function is called once before the real time control portion begins. It initializes data structures to their correct states. It then executes the `new_shot_phase_tick` function which sets the phase sequence for each category and initializes the first phase that is active in each phase sequence. Initial values for all target vector elements are also set and the `phasebegin` function for each phase is called.

13.2.4 update_the_state

The PCS uses two time intervals during the real time control: a "fine scale clock tick" which is usually 1 microsecond and is the units of `rtheap->currenttime` and `rtheap->update_time` and a "shot phase clock tick" which is the smallest interval between vertices and is set by the installation (1 millisecond is a good value). The shot phase clock tick should not be any smaller than the cycle time of the fastest processor.

This function determines whether the time has reached the time of the next shot phase clock tick. If so, then the `new_shot_phase_tick` function is called to make any necessary phase sequence changes, phase changes, and non-continuous target changes.

This function also determines whether new target values need to be computed by calling the `new_continuous_target_c` function. The target vector is recomputed at intervals of one shot phase tick, on the half-tick. The target vector is also recomputed if any phase changes were made.

Its possible that it is necessary to execute these two tasks more than once if the the value of `rtheap->update_time` is greater than one shot phase clock tick later than the last time this function was called. In that case, this function will loop, incrementing the next shot phase tick, until this value is greater than `rtheap->update_time`.

13.2.5 time_critical

This function does the real time control during the plasma discharge. It starts executing when the primary start time has been reached.

It first calls the `startup_state` function to initialize phase sequences, phases, and the target vector elements. It then calls the function which starts data acquisition if such a function is defined (this was needed for the Supercard hardware).

This function then goes into a while loop until either an error occurs or the time to stop has been reached. This loop executes the functions in the function vector.

After each function executes, the values of `rtheap->currenttime` and `rtheap->update_time` are compared. If the current time is greater, then the `update_the_state` function is called to make any phase sequence or phase changes, and to update the target vector. After all the functions in the function vector have been executed, the value of `rtheap->update_time` is compared to the value of `rtheap->currenttime` and if less, then `rtheap->update_time` is set equal to `rtheap->currenttime`. The `update_the_state` function is then called.

The time at which the call to `update_the_state` is made is important. It was originally called only after all functions in the the function vector were executed. This assumed that the value of `rtheap->currenttime` was being set to the next expected value in the last function. The `update_the_state` function then looked at the current time to make its updates.

But with the move away from the Supercards, the function that sets the current time was moved to a different location in the function vector. The desire was to get the state updated at the end of the function vector when processors have time leftover before the next set of data are available. A new value `rtheap->update_time` was added which could be set to the next expected time. Then the state could be updated to this new time while the processor is expected to be waiting to start the next cycle.

Then, if the value of `rtheap->currenttime` is set to a value that is later than the expected time, `rtheap->update_time`, the state must be updated again. Thus, the check of these two times after each function is executed. This check would be sufficient in itself, but the computation time needed to get the state updated would delay the execution of the rest of the functions. So the most efficient method is to set `rtheap->update_time` to the expected next time in the last function in the function vector so the computational time needed to update the state does not delay the real time control.

If `SAVE_TIMING` is used, then this function saves the number of processor clock ticks before and after each function and puts these values in a scratch parameter data

block pointed to by the global variable `save_timing_ptr`.

13.3 Installation-specific functions

The real time code requires a set of installation-specific functions that are called by the infrastructure code described in Sec. 13.1. These are functions that know how to perform tasks required by the infrastructure code that are specific to the particular real time processor hardware used in the PCS. Each of these required functions is described in this section. By convention, these functions are contained in the file `realinstall.h` in the installation-specific code area.

13.3.1 real_install_init

This routine contains any initializations code which is needed at the time the realtime process starts up. This code is called only once upon the start up of a PCS real-time process. This function is called only if a `main()` function is compiled (see Sec. 13.1).

Calling format:

```
int real_install_init(int argc, char **argv)
```

Arguments:

- `argc`: The standard value that passes the number of command line arguments.
- `argv`: The standard array of pointers to the command line argument strings.

Return value:
Any value returned is ignored.

13.3.2 real_install_main

This routine contains any initialization code which is needed at the time the realtime process starts up. This code is called only once upon the start up of a PCS real-time process. This function is called only if a `main()` function is not compiled (see Sec. 13.1).

Calling format:

```
int real_install_main(int argc, char **argv)
```

Arguments:

- `argc`: The standard value that passes the number of command line arguments.
- `argv`: The standard array of pointers to the command line argument strings.

Return value:
Any value returned is ignored.

13.3.3 set_time_zero

In a multi-CPU PCS configuration this routine synchronizes all CPUs and sets the value in `rtheap->timezero` to a reference start time.

In a single CPU PCS configuration this routine simply sets the value of `rtheap->timezero` to 0.

Calling format:

```
void set_time_zero(struct rt_heap_misc *rtheap,int cpu_num)
```

Arguments:

- `rtheap`: Pointer to the standard structure that describes the real time memory heap area.
- `cpu_num`: The PCS physical CPU number of the real time processor on which the code is executing.

Return value:
None.

13.3.4 wait_for_start_signal

This function is called after a real-time process has finished all of it's preshot initializations and is at the point where it is ready to begin the shot. The function should wait until an installation specific event indicating the time of the start of shot has arrived. When this function returns the time should be approximately `PRIMARY_START_TIME`. The function `time_critical` that executes all of the real time control algorithms during the discharge is called immediately after this function returns.

Calling format:

```
int wait_for_start_signal(struct rt_heap_misc *rtheap)
```

Arguments:

- `rtheap`: Pointer to the standard structure that describes the real time memory heap area.

Return value:
If the function completes without error, the function should return 0. If there is an error, an installation-specific, positive, error code should be returned. If the return value isn't 0, the time critical code is not executed and the PCS completes the shot immediately.

13.4 Data acquisition

In general, in order to do feedback control it is necessary to acquire data from sensors on the controlled system. These data can be either analog values that have been converted to digital form with a digital to analog converter module or the data can come from digital input/output boards directly in digital format.

The PCS infrastructure handles an unlimited number of channels of data from A/D converter modules as well as digital bit data. See Sec. 3.2 for details of the dma_region.

The infrastructure makes a few assumptions about how the digitizer data will be handled, but most of the data acquisition functions are performed by installation specific code.

This section describes how the installation specific code and the infrastructure combine to acquire the digitizer and digital i/o data.

NOTE: THIS IS AN EXPLANATION OF OLDER VME BASED HARDWARE. SEE Sec. 17 FOR NEWER LINUX BASED SYSTEMS.

The data acquisition is divided into two portions.

- The "start acquisition" routine, normally placed near the end of the function vector, is used to set up the data acquisition and, possibly, trigger the digitizers. The assumption is that there will be a delay after the digitizers are triggered before the data are available. During this delay the PCS can finish the work of the current cycle. So, acquisition of data for a control cycle is started near the end of the previous cycle.

The digitizer data normally would be written directly to the processor memory by the acquisition circuits using DMA hardware. The start acquisition routine determines the location of the DMA buffer and loads the DMA hardware with the address.

The data archiving facility of the PCS is based on the procedure that when data in a buffer should be saved for archival, the pointer to the buffer is changed to a new buffer leaving behind the data to be saved. All PCS buffers that contain data for archival are saved at the same time. So, this is the procedure used for saving the raw data from the digitizers.

The DMA buffer is reused until it is time to move to another buffer leaving a set of data behind. Most of the work of changing buffer pointers to save data is performed after all function vector routines are called. But, in order to decide which DMA buffer to use for data for the next cycle, the decision about whether it is time to archive data must be made by the start acquisition routine.

- The first routine called from the function vector is normally the "get new data" routine that collects the new set of data to be used on the control cycle that is just starting. Presumably, the data acquisition was started by another routine (the start acquisition routine), so the get new data routine first waits for a flag that the new data are available. The new data are expected in the DMA buffer. When the data arrive, the get new data routine copies the digitizer values, subtracts the appropriate baseline value and writes result in a buffer with a fixed location. It is from this buffer (`rtheap->datavector`) that all of the control algorithms read the data they need.

  The get new data routine also copies the communication vector. The communication vector is a region of memory where other processors can write data. Because these data are written with DMA operations, the processor doesn't know when the data change. The get new data routine reads the communication vector with instructions that know not to look in the cache memory for the communication vector values. This guarantees that the correct values are obtained directly from the processor memory. The communication vector values are copied into the "control cycle loop" copy of the communication vector (`rtheap->combufloop`). This is a buffer that can be safely accessed by C code without disturbing new DMA writes of new data into the communication vector. All algorithm code uses the control cycle loop communication vector copy which will not change during a control cycle.

  Finally, the get new data routine determines the current time. In the DIII-D

PCS, the data that arrive from the digitizers are accompanied by a value giving the time that the data were acquired. This value is written to `rtheap->currenttime`. Algorithms use this value for control work, and the infrastructure code uses this value to determine when to change target vector values.

The get new data routine compares the time value to the "time to stop" value provided by the operator to determine if the shot should be over. If so, the master data acquisition processor sends a flag to the slave processors telling them to stop. Then the get new data routine causes a complete return from the `time_critical` routine.

13.5 Software testing modes

The PCS has 3 special modes of operation referred to as software testing modes. The PCS operator can choose one of these modes on the "operating setup data" window in the user interface. The real time code can detect that a software test mode has been chosen by testing the value of `rtheap->software_test_mode`. The possible values are the following.

- `NORMAL_OPERATION_MODE`: No software test mode has been selected.
- `SOFTWARE_TEST_MODE`: The mode called "software test mode" has been selected. This is an installation-specific test mode. See Sec. 13.5.1 for a discussion of how to implement this mode.
- `SIMULATION_TEST_MODE`: In this operation mode the PCS communicates with a separate simulation server process to obtain input data. See Sec. 13.5.2 for a discussion of how to implement this mode.
- `SETUP_TEST_MODE`: In this mode the PCS executes an entire shot cycle normally, except that the time critical portion is not executed. That is, the function `time_critical` is not called. This mode is intended to be used to obtain an image in the S file of the PCS real time memory heap as it is configured at the start of a shot. This is useful for debugging problems that result from errors in configuring the real time processor's memory during shot setup. There is no special installation-specific code required to implement this testing mode.

13.5.1 Implementing software test mode

In software test mode, the PCS acquires its input data from an installation-specific function. A typical application of this mode is to provide 0 for all of the input data

and simply step the current time value forward on each cycle. This mode is useful for quick tests that don't involve the actual input data.

In order to implement this mode of operation, some installation-specific "real time" code is required. The function that normally acquires the PCS input data from digitizers and/or other sources during the real time execution, should detect that the PCS is operating in software test mode. When this mode is detected, an installation-specific function is called. The function should copy the desired input data into the appropriate locations on the real time processor in order to emulate the procedure followed when data are acquired from digitizers.

To detect that the PCS is operating in software test mode, the data acquisition function should test the value of `rtheap->software_test_mode`. This flag has the value `SOFTWARE_TEST_MODE` when the PCS is operating in software test mode.

The simplest way to implement the installation-specific function to provide the dummy data in this operation mode is to call a function that executes on the real time processor. However, it may be desirable to have the function that provides the data execute as part of the `host_cpu` process on the host processor. In this case, the same type of procedure that is discussed in Sec. 13.5.2 for implementing simulation mode could be used.

13.5.2 Implementing simulation mode

In simulation mode, the PCS acquires its input data from the simulation server process. Also, the PCS calculation results from each cycle are sent to the simulation server to allow, if desired, a simulation of the plasma response to the PCS commands.

In order to implement this mode of operation, some installation-specific "real time" code is required. The function that normally acquires the PCS input data from digitizers and/or other sources during the real time execution, should detect that the PCS is operating in simulation mode. When simulation mode is detected, the real time data acquisition function should cause the function `host_read_data_set_simserver` to be called instead. This function communicates with the simulation server to exchange the necessary data. It then calls the installation-specific function `host_install_write_simserve` (Sec. 10.2.12) which is responsible for copying the data from the simulation server to the correct locations on the real time processor in order to emulate the procedure followed when data are acquired from digitizers.

To detect that the PCS is operating in simulation mode, the data acquisition function should test the value of `rtheap->software_test_mode`. This flag has the value `SIMULATION_TEST_MODE` when the PCS is operating in simulation mode.

The functions `host_read_data_set_simserver` and `host_install_write_simserver_data`

are both part of the `host_cpu` process and are executed on the host computer. (`host_read_data_set_simserver` is located in the infrastructure file `hostmain.c`.) So, there must be some method available for the real time processor to cause the `host_cpu` process to call `host_read_data_set_simserver`. The real time processor must then wait for this function to complete before it continues. If nothing more sophisticated is available, the `host_cpu` process and the real time processor could simply use a couple of variables to implement handshaking flags. The `host_cpu` process could poll one flag as part of the code in `host_install_start_realtime` or `host_install_stop_realtime` to detect when `host_read_data_set_simserver` should be called. Then, the real time processor could poll the other variable to detect when the function has completed. Since in simulation mode the PCS isn't really executing in real time, the speed of this handshaking process isn't important (except for the affect it has on the total time to run a simulation mode shot).

13.6 Hardware test mode

The PCS operator can choose on the "operating setup data" window that the operating mode should be "hardware test." Hardware test mode is an installation-specific mode of operation. It is intended for use in testing the PCS as it normally would be used during tokamak operations, with all data being acquired through the standard digitizer and digital input/output hardware. In this test mode, though, the PCS isn't operated in synchronization with the tokamak timing system and, because there is no plasma, the diagnostic data are meaningless. This mode of operation simply allows a test of the basic operation of the PCS.

A separate hardware testing mode is defined to allow this operation without synchronization with the tokamak shot. Installation-specific triggering, etc. that is monitored during normal operation of the PCS might prevent operation without synchronization unless there is some special code to do something else during testing mode. For instance, if the PCS waits for a hardware trigger before proceeding during normal operations, this trigger wouldn't appear during a hardware test. So, installation-specific code is necessary to detect that hardware test mode is in effect so that the delay for the hardware trigger can be skipped.

The PCS code can detect hardware test mode by monitoring the value of `rtheap->hardware_test_mo`
The value is 0 during normal operations and is a nonzero value during hardware test mode.

Hardware test mode can be used to do a hardware simulation if data are loaded before the shot and this data OVERWRITES the data from the digitizers and other hardware. It is customary to preload the data in the data acquisition's algorithm

(usually called "baselinedata") in the preshot "host_init" function. The same functions used in the simserver to load shot data can also be used here. Then in real time the correct set of data can be found by matching the current time with the time of the data set and the data values can be written to the datavector.

14 Accessing the S file data

The "S" file is optionally written after the discharge with a copy of the real time computer's memory used for data storage for the PCS. A separate file is written for each real time cpu. A set of IDL routines is available for use in interactive examination of the content of an S file. The routines know how to locate the data structures that are particular to the PCS.

These routines are compiled into the file `s_file_access`. The first routine to call is `getrtdata`. For example

```
IDL> getrtdata,900002,1
% Compiled module: GETRTDATA.
shotnumber =       900002
cpunumber  =        1

**********************************************************************
This is the setup routine for reading data from PCS S shotfiles
For a complete set of idl routines which can be used here enter
 listget
at the idl prompt

The s file search directory is: /link/ops/store_v10/
NOTE: the directory to be searched can be specified in the
environmental variable SFILEDIR.
**********************************************************************

Reading filename = /link/ops/store_v10//s900002_cpu1.dat
processor type            1
rtheap start address =   1110834688

endrtmem             =   1129571680
startrtmem           =   1108553728
```

```
size of all memory   =     21017952
size of file         =     25753448
**************************************
finished loading structure: rtheap
**************************************
```

There are many routines that can look at the structures found in the real time heap. Type the command `listget` to get a listing of these routines. Some of the more useful routines are the following:

- dumpsfile[,filename] dumps everything in the S file.
- dumplayout[,filename] dumps the layout of the real time heap.
- dumprtheap[,filename] dumps the content of all the structures in the real time heap.
- dumpextra dumps the content of the `extra_file_info` structure.
- dumphost_info dumps the content of the `host_info_offsets` structure.
- getrtheap,rtheap returns the `rtheap` structure.
- printrtheap,rtheap prints the `rtheap` structure.
- getttime,times gets the times that were saved during the shot.
- dumpw,address,count[,filename] prints the content of the S file starting at the given address as integer and float numbers. `count` is the number of values to print. The address can be an offset into the S file, or a memory address given by a pointer found in one of the real time structures. This is particularly useful to look at parameter data.
- getphase,phases[,category] returns the phases used in real time. If `category` is not given, then all phase structures are returned. Note that the `category` value is 0 based.

There is a routine to read every structure that is found in the real time heap. These routines are very useful for finding problems that appear to be related to memory allocation.

15 Archiving and restoring PCS data

The PCS produces data that should be archived into a permanent database for use in evaluating the performance of the PCS and for reproducing previous PCS setups for new discharges.

The PCS archives two types of data.

1. The PCS setup is archived by the waveform server process.
2. The `host_cpu` processes archive data acquired and calculated during the discharge. These data are primarily values versus time. The data values can be either integers or floats.

The PCS restores data from the database for two purposes.

1. All or a portion of the PCS setup that was archived for a given discharge can be restored to be used as part of the setup for another discharge.
2. Diagnostic data are restored by the simulation server.

These archiving and restoring functions are managed by code within the infrastructure. The infrastructure code calls several low level routines that implement the functions required to add or restore data in the installation-specific database format. To implement the required archive and restore capability at a new PCS installation it is only necessary to write the required low-level installation-specific functions. These functions are described in this section.

15.1 Overview of installation-specific database functions

The installation-specific functions for PCS archiving and restoring are located in the files `pcsshotfile.c` and `close_shotfile.c`. The functions in `pcsshotfile.c` are linked into the the waveserver and the host_cpu programs. The functions in `close_shotfile.c` are linked into the lockserver program. Each function is responsible for different tasks. Here is an outline of the tasks. (The term "pointname" is used here to represent a set of archived data for a given shot.)

Here are the functions required in the file `pcsshotfile.c`. Note that the function `close_shotfile` is found here as well as in the file `close_shotfile.c`. This function used to be shared between all the processes needing it, but this proved to be too difficult to maintain. Thus, it was duplicated in both files.

1. The waveserver needs to be able to read a PCS setup from a shot archive. The function `read_shotsetup_shotnum` is used for this. This function is called by the waveform server whenever a PCS restore is performed by the PCS operator.

2. At the start of a discharge, in response to a command from the lockout server, the waveserver saves the PCS setup by calling the function `write_pcs_pointname` which is responsible for writing the PCS setup data to the new shot archive.

3. After the shot, the `host_cpu` processes write pointnames to the shot archive using the function `store_data`. Each `host_cpu` process writes data in turn in response to a command from the lockout server. The shot file would need to be opened when the first pointname is written. A global variable could be used to indicate the file is open and this variable could be checked before each pointname is written.

4. When all pointnames have been written, the `close_shotfile` function is called. The shot file should then be closed and the global variable reset to indicate this. Note that each `host_cpu` process finishes writing its pointnames before the next one starts so it is not neccessary to do any file locking.

5. The restore interface has the ability to display the phase sequence stack and the phase stack pointnames that are archived in a shot (e.g., pointnames `PHASESTK1` and `SEQSTK1`). In order to display this information the waveserver needs to retrieve these pointnames. The function `read_pointname_shotnum` is used for this task.

Here are the functions required in the file `close_shotfile`. Note that the function `close_shotfile` is found here as well as in the file `pcsshotfile.c`. This function used to be shared between all the processes needing it, but this proved to be too difficult to maintain. Thus, it was duplicated in both files.

1. In the lockout server process at the time of final lockout:

   - The function `delete_shotfile` is called to delete any existing shot file for the given shot. Deleting the existing shot file can be used to avoid problems that arise if the same pointname is used multiple times in a given shot file. This problem can arise, depending on the database implementation, if the same shot number is used more than once, a common occurrence when testing.

- Also, any installation-specific preparation for a new shot archive could be done at this time. This might involve something simple like the creation of a shot file. Or it may involve more time-consuming tasks such as the "registering" of a new shot (e.g. MDSplus archives require the creation of new "trees" for data storage).

2. After all archiving is complete, the lockserver process calls `close_shotfile` which is responsible for "closing out" the shot archive. The necessary work here is defined by the installation. This might include writing a flag in the archive indicating that the shot is "closed" to new data, or copying the file to another location.

15.2 Functions found in file pcsshotfile.c

15.2.1 write_pcs_pointname

This function is called to archive the PCS setup.

Call format:

```
int write_pcs_pointname(
    char *ptname,           /* name of the data being archived */
    struct pfi42 *pthead)   /* a structure defined in the infrastructure
                               (in file pfi.h) */
```

The setup data are "parcelled" by the infrastructure code into separate groups based on the data type. These groups are char (or ascii), short int (16-bit), int (32-bit), float (32-bit), and double (64-bit). In addition, a checksum array is added which keeps track of the counts in each group as the setup gets parcelled. The type of data in the checksum array is specified by a flag in the function arguments. The installation-specific function is responsible for taking these groups of data and storing them and their sizes in an archive.

One possible way of storing this data is to store the pfi42 header structure, followed by the data. This is how the setup is written out by the waveserver to the NextShot and future shot archive files (see function `store_shotsetup` in `setup_routines.c` in the infrastructure).

Function arguments:

- `ptname`: The pointname to use for archiving the data. This is the same as the definition of the macro `pcs_setup_pointname` in the `infra_installdefs_*.h` file that is used to hold installation-specific tuning parameters for the infrastructure.
- `pthead`: The pointer to a structure that contains the pointers to the data to be archived. The fields in the pfi42 structure that are used to pass input to this function are the following:
  - `snum`: the shot number.
  - `nwascii`: the count of 16-bit words of ascii data.
  - `nwint16`: the count of 16-bit words of 16-bit integer data.
  - `nwint32`: the count of 16-bit words of 32-bit integer data.
  - `nwreal32`: the count of 16-bit words of 32-bit real data.
  - `nwreal64`: the count of 16-bit words of 64-bit real data.
  - `nwdata`: the count of 16-bit words of checksum data.
  - `nbdata`: the number of bits in the checksum data type. For new data this is always 32. (For DIII-D there is some data archived where this value was 16.)
  - `vhascii`: the pointer to the ascii data.
  - `vhint16`: the pointer to the int16 data.
  - `vhint32`: the pointer to the int32 data.
  - `vhreal32`: the pointer to the real32 data.
  - `vhreal64`: the pointer to the real64 data.
  - `ptdata`: the pointer to the checksum data.

  Note that the counts are given in 16-bit increments (for historical reasons).

15.2.2 read_shotsetup_shotnum

This function is called to restore the PCS setup data. It is the companion to the function `write_pcs_pointname` (Sec. 15.2.1). The function is responsible for finding the setup data in the location into which it was archived by the function `write_pcs_pointname` and returning the data to the calling function. The data are returned in the same parcelled out groups that were provided to `write_pcs_pointname`

when the data were archived. The locations and amount of data are indicated by filling in the same fields in the `struct pfi42` function argument that are provided to `write_pcs_pointname`. If the data are stored as suggested in Sec. 15.2.1, then it can easily be read (see the function `read_shotsetup_setupfile` in file `setup_routines.c` in the infrastructure).

Calling format:

```
int read_shotsetup_shotnum(
    int     shotnum,        /* shot number */
    struct pfi42 *ptinit)   /* a structure defined in the infrastructure
                               (in file pfi.h) */
```

Function arguments:

- `shotnum`: The shot number for which the PCS setup should be fetched.
- `pthead`: The pointer to a structure into which the information about the restored setup data should be placed. The fields to fill in are the following.
  - `nwascii`: the count of 16-bit words of ascii data.
  - `nwint16`: the count of 16-bit words of 16-bit integer data.
  - `nwint32`: the count of 16-bit words of 32-bit integer data.
  - `nwreal32`: the count of 16-bit words of 32-bit real data.
  - `nwreal64`: the count of 16-bit words of 64-bit real data.
  - `nwdata`: the count of 16-bit words of checksum data.
  - `nbdata`: the number of bits in the checksum data type.
  - `vhascii`: the pointer to the ascii data.
  - `vhint16`: the pointer to the int16 data.
  - `vhint32`: the pointer to the int32 data.
  - `vhreal32`: the pointer to the real32 data.
  - `vhreal64`: the pointer to the real64 data.
  - `ptdata`: the pointer to the checksum data.

This function manages its own memory allocation. On the first call to the function the pointers in `pthead` are `NULL`. On subsequent calls the pointers will be preserved at the values written on the previous call. So, `read_shotsetup_shotnum` must handle decisions about either freeing any memory previously allocated or reusing the same memory.

15.2.3 store_data

This is the function responsible for archiving PCS data that are obtained during the real time processing. This function archives one set of data values versus time each time it is called. (For DIII-D, this means that one "pointname" is written for each call to the function.)

Calling format:

```
int store_data(
    int      chnl,                      /* channel index            */
    int      shotnum,                   /* shot number              */
    char    *ptname,                    /* pointname                */
    char    *timebase,                  /* timebase pointname       */
    char    *ptdesc,                    /* pointname description    */
    char    *data_type,                 /* data type                */
    float    inhnum,                    /* inherent number          */
    float    phys_zero,                 /* physics zero or baseline */
    int      delta_bytes,               /* bytes between time values */
    int      nblocks,                   /* contiguous data blocks   */
    int     *nsamples,                  /* number of samples        */
    int     **first_data)               /* first data value         */
```

Function arguments:

- `chnl`: This argument is relevant only for data of type "raw" or "dio". It is the index in the `struct archive_vector_info` of the pointname of the data to be archived. See below for further discussion.
- `shotnum`: The number of the shot for which data are being archived.
- `ptname`: The pointname for the data being archived.

- `timebase`: The name of the pointname containing the timebase data corresponding to the data being archived. See below for further discussion.
- `ptdesc`: A character string indicating where the data originated in the PCS.
- `data_type`: This argument indicates the type of data being archived. The possible values are "timebase", "raw", "dio", "int", or "float". See below for further discussion.
- `inhnum`: The "inherent number" for the data. The inherent number is the value by which the data should be multiplied to obtain a desired units conversion. A simple example is a conversion from volts to amps. This value comes from the `vector_info` structure in the algorithm master file. It can optionally be archived with the data (depending on the installation-specific requirements).
- `phys_zero`: The "physics zero" value from the `vector_info` structure in the algorithm master file. This value can optionally be archived with the data (depending on the installation-specific requirements). For "raw" data this is the baseline value acquired by the PCS.
- `delta_bytes`: The count of bytes separating the values to be archived in the input array. See below for further discussion.
- `nblocks`: The count of blocks of values in which the input data are provided. See below for further discussion.
- `nsamples`: An array giving the number of values in each block. See below for further discussion.
- `first_data`: An array of pointers to the first data value in each block. See below for further discussion.

This function is used to archive all of the various types of data saved during the real time processing. The `data_type` argument indicates the type of data to be archived. The meaning of the various data types is as follows.

- `timebase`: These are values of type `double`. The values are the sample times appropriate for one or more pointnames. The timebase data are typically archived in a pointname and the name of the appropriate timebase pointname is then recorded with the actual data values. Then, software that accesses a set of data values would read the data from the data pointname, also find the timebase

pointname from the data pointname, and then read the timebase data from its pointname.

- `dio`: Raw data obtained from a digital input/output source. This is always an array of type `short`.
- `int`: An array of values of type `int`. For instance, these values might come from the "intcommand" vector.
- `float`: An array of values of type `float`. For instance, these values might come from the "shape" vector.
- `raw`: Values acquired from the analog digitizers. See below for further discussion.

The "raw" data are contained within data structures of type `dma_region`. The function arguments point to the data as values of type `float`. However, the actual format of the data and how it is treated during archival will be installation-specific. For DIII-D, for instance, the original integer values obtained from the digitizer are extracted from the data and archived as values of type `short`.

The `chnl` argument is only relevant for "raw" and "dio" data. This value is only needed if some installation-specific values must be obtained for archival from an array of values for each input data channel. The value of `chnl` is the index into the `struct archive_vector_info` array which gives the pointnames of the data to be archived. This structure array exists as a parameter data block in the `host_cpu` process if any raw data are to be archived. A second parameter data block indicating the size of this array must also exist. Similar blocks must exist for the "dio" data if desired. The names of all these blocks are given by definitions in the `installdefs.h` file.

The `inhnum` and `phys_zero` arguments are the values specified in the `vector_info` structure initialized in the algorithm master files. The "inherent number" is the multiplication factor to use when converting the data to physics units and the "physics zero" is the zero offset. These values can be stored with the pointname if desired. Also specified as arguments are the "pointname description", which may be archived, and the shot number.

The `nsamples` and `first_data` arguments are arrays because they represent counts and pointers, respectively, of the phases used throughout the shot. The `nblocks` argument indicates how many phases. So to create a one- dimensional array of data, the total size must first be found by counting the number of samples in each block:

```
for(iblock=0, nstot = 0; iblock<nblocks; iblock++)
```

```
        nstot += nsamples[iblock];
```

Here is an example of how to treat data of type "float".

```
int iblock;                             /* block  index                */
int isample;                            /* sample index                */
int nstot;                              /* total samples in all blocks  */
int nbtot;                              /* total bytes for all values   */
int bytes_per_value;                    /* bytes per data value         */
void *memptr = NULL;                    /* malloc result                */
char *outptr;                           /* output data pointer          */
char *inptr;                            /* input  data pointer          */

/*
count total samples
*/
    for(iblock=0, nstot=0; iblock<nblocks; iblock++)
        nstot += nsamples[iblock];

    if (strcmp(data_type,"float") == 0)
        bytes_per_value = sizeof(float);

    nbtot = nstot * bytes_per_value;
    memptr = malloc(nbtot);

/*
gather data into data buffer
*/
    outptr = (char*)memptr;

    for(iblock=0; iblock<nblocks; iblock++)
    {
        inptr = (char*)first_data[iblock];

        for(isample=0; isample<nsamples[iblock]; isample++)
        {
            if (strcmp(data_type,"float") == 0)
                *((float *) outptr) = *((float *) inptr);
```

```
            outptr += bytes_per_value;
            inptr  += delta_bytes;
        }
    }
    free(memptr);
```

Here is an example for the "raw" data. It is here that the installation is allowed to change the size of the data to be archived. For example, if the raw data are really 12 bit digitizer data, it may be desired to store it as 16 bit shorts instead of 32 bit floats. We will do just that in the following example.

Malloc memory for the data using this total count and the size of the archive data. Then loop through the data again moving the sample values into the data block.

```
    bytes_per_value = delta_bytes/2;      /* data comes in as 4 byte floats */
    memptr = (char *) malloc(nstot * bytes_per_value);
    outptr = (char *) memptr;

    for(iblock=0; iblock<nblocks; iblock++) {
        inptr = (char *)first_data[iblock];
        for(isample=0; isample<nsamples[iblock]; isample++) {
            *((short *) outptr) = *((short *) inptr);
            outptr += bytes_per_value;
            inptr  += delta_bytes;
        }
    }
    free(memptr);
```

The timebase argument is a character string indicating the pointname of the time array. The PCS was designed to store the timebases as separate pointnames. If the installation wishes to archive the time values with the data values, then timebases will need to be stored in static memory, by name, so the data can be accessed when a non-timebase pointname is archived. Note that a timebase cannot be assumed to be a simple array of times evenly spaced apart.

15.2.4 close_shotfile

```
void close_shotfile(
     int      shotnum,        /* shot number                  */
     int      flag)           /* installation dependent flag */
```

The host_cpu process calls this function so it can close the shot file (if one is used) so that another host_cpu process can open and write to the file.

Note that the same function exists in the file close_shotfile.c. That version of this function used to be linked into the waveserver and host_cpu processes. But the linking of three different programs needing this function proved to be too difficult to maintain so a duplicate function was added to this file.

15.2.5 read_pointname_shotnum

```
int read_pointname_shotnum(
     char *ptname,            /* pointname, e.g. "PHASESTK1" */
     int ishot,               /* shot number                 */
     void **data,             /* data to return              */
     int *nbytes)             /* number of bytes returned    */
```

This function is called by the waveform server whenever a request is made through the restore interface to display the phase sequence stacks or the phase stacks for a given shot. This information is archived in the pointnames PHASESTK% and SEQSTK% where % is the cpu number. These pointnames are all character strings giving names of categories, phase sequences, and phases and the times when the phase sequences and phases were active during the shot.

15.3 Functions found in file close_shotfile.c

15.3.1 close_shotfile

```
void close_shotfile(
     int      shotnum,        /* shot number                          */
     int      flag)           /* flag indicating the calling process */
```

This function is called by the lockserver program. The lockserver's task is to "close out" the shot. This can mean a variety of things depending on the installation. Marking the shot as finished, copying it to another area, starting a compression program, etc. are a few possibilities.

15.3.2 delete_shotfile

```
void delete_shotfile(
     int      shotnum)        /* shot number                          */
```

This function is called by the lockserver to delete any existing shot files for the given shot. This is very useful for test shots. Also, this function can be used to "setup" for the shot (e.g. MDSplus archives require the creation of new "trees" for data storage).

The lockserver calls this function before it enters into its final lockout state. This state causes the waveserver to archive its setup. So there is no need to worry about the timing of the delete and the writing of the setup.

16 System Category

A system category is a good addition to the plasma control system and is highly recommended. In this category a single algorithm with known default values can be made to be active throughout the discharge. This algorithm can, for instance, always zero out command outputs in the real time cleanup function. This algorithm can have parameter data blocks that will always be available to algorithms in other categories because a phase that uses the algorithm will always be active. This algorithm can pass a mixture of category data from one processor to another to minimize communication. In general, a system category allows features to be present in the plasma control system which involve more than one category or are part of the overall system.

17 Data Acquisition Category

The PCS requires certain steps regarding the acquisition of raw data and its use. This includes the acquisition of the raw data, the calculation of baselines, the saving of raw and processed samples, the conversion from raw units to physical units, and the passing of data from one processor to another. These steps are fairly standard and there are infrastructure and common functions that can be used to accomplish these tasks. The Data Acquisition category and its baselinedata algorithm are usually used for these purposes. Some notes about this category and how it is used.

The digitizer data normally would be written directly to the processor memory by the acquisition circuits using DMA hardware. The function `wait_for_new_time` reads the time value from a digitizer until it changes. At that point, the real "time" is known and the system is in sync.

The function `get_new_data` copies the digitizer values, subtracts the appropriate baseline value, and writes the result in a buffer with a fixed location. It is from this buffer (`rtheap->datavector`) that all of the control algorithms read the data they

need.

The function `get_new_data` then sets the current time. This value is written to `rtheap->currenttime`. Algorithms use this value for control work, and the infrastructure code uses this value to determine when to change target vector values.

Finally, the function `get_new_data` compares the time value to the "time to stop" value provided by the operator to determine if the shot should be over. If so, then `rtheap->return_status` is set to TIMETOSTOP_RETURN which causes a complete return from the `time_critical` routine.

The `get_new_data` function also copies the communication vector. The communication vector is a region of memory where other processors can write data. Because these data are written asynchronously, the data can change at any time. Two buffers are provided: one to write to (`rtheap->combuffer`) and one to read from (the "control cycle loop" `rtheap->combufloop`). The data are copied from the write buffer to the read buffer. All algorithm code uses the control cycle loop communication vector copy which will not change during a control cycle.

The data archiving facility of the PCS is based on the procedure that when data in a buffer should be saved for archival, the pointer to the buffer is changed to a new buffer leaving behind the data to be saved. All PCS buffers that contain data for archival are saved at the same time. So, this is the procedure used for saving the raw data from the digitizers.

The DMA buffer is reused until it is time to move to another buffer leaving a set of data behind. Most of the work of changing buffer pointers to save data is performed by the `save_samples` function after all other function vector routines have completed.

17.0.1 wait_for_new_time

This real time function is always put into the function vector as the first function. It is meant to wait until a new set of data are available, but it does not update the real time heap. The function may repeatedly read a digitizer to get the time until the time changes. It may read a sharable memory address that has the current time until the value changes. It may read a memory address that another processor writes the current time to until that value changes. Whatever this function does, it will loop until a time value changes. It is important to always have a timeout during the read just in case the value does not change. Also, it is important to declare any memory address being written to from the outside as `volatile` so the compiler does not eliminate the read loop (this tells the compiler that the value will change).

Note that this function keeps the processor in sync with the real time during the shot (and thus the other processors). If, upon entering, the first read of the time

gives a different value than the last read, then the true time is unknown. The true time could be anything between the time read and the next time that will be read. A decision must be made whether to wait until the time changes (thus, skipping a cycle), or whether to continue and run the cycle for the current value of the time. The danger of running the cycle is that the processor is not in sync with the other processors in the PCS.

```
/*
*****************************************************************************
SUBROUTINE: wait_for_new_time

wait for a new time - this is always the first function in the function vector

Global variables needed here (all are set in the init function):
    cycle_times            int array giving the cycle times for each
                           cpu in microseconds
    number_of_digitizers   the number of digitizers on this processor
    minimum_cycle_time     cycle time of digitizer or if no digitizer,
                           cycle time of fastest processor that provides
                           this cpu its data
*****************************************************************************
*/
void wait_for_new_time(struct rt_heap_misc *rtheap)
{
#ifdef PCS_RT_CPU

#if defined(LINUXOS)
unsigned long long timer1, timer2, tdiff;
#endif
unsigned int new_time;
volatile unsigned int currenttime;

    if(!rtheap->software_test_mode)
    {
/*
If the cycle time for this cpu is 0, then wait
for a new remote time to arrive.
*/
```

```
       if(cycle_times[CPU_NUM-1] == 0)
       {
          wait_for_new_remote_time(rtheap);
          if(rtheap->return_status != 0) return;
       }
/*
If there are any digitizers on this processor,
then get the time value.  When the time value is
different from the currenttime, then continue.
The assumption is that the time to complete a
cycle is less than the cycle time for the processor.
The value of minimum_cycle_time is the rate at
which the digitizer cycles in microseconds.
*/
       if(number_of_digitizers > 0)
       {
#if defined(LINUXOS)
       readticks64(&timer1);
#endif
          currenttime = rtheap->currenttime;
          new_time = rtheap->currenttime + minimum_cycle_time;
          while(currenttime < new_time)
          {
              currenttime = (volatile unsigned int *)<time address>[0];
#if defined(LINUXOS)
              readticks64(&timer2);
              tdiff = timer2 - timer1;
              if(tdiff/ticks_per_usec >= timeout_in_usec)
              {
                 rtheap->return_status = TIMEOUT_RETURN;
                 <cleanup>
                 return;
              }
#endif
          }
       }
    }
#endif /* #ifdef PCS_RT_CPU */
```

```
/*
Save timing data before currenttime is changed.
*/
#ifdef SAVE_TIMING
    save_timing(rtheap);
#endif
}
```

There are several things to point out about the above code.

- The image built to run on the real time processors can be a combination of the host_cpu and the realtime code. This was found to be the most efficient method and avoids the necessary communication between two separate programs. A macro PCS_RT_CPU indicates this combination and is used around any code that only executes on the real time processor.

- The cycle_times array is an array of ints giving the cycle times of each processor in microseconds. A cycle time of 0 is a special case which means that the time of the cycle varies. It is used when a cycle time is long (like the slow loop rtEFIT). The function `wait_for_new_remote_time` needs to wait until the current value of the time (wherever it comes from) changes so that the processor can get back in sync. So it would read the time upon entering, then wait for that value to change.

- The minimum_cycle_time value cannot be zero. It needs to be the actual cycle time of any digitizer that exists. This allows the cycle time of the processor to be zero, yet there can also be a digitizer which is cycling at a fixed rate.

- readticks64 is a LINUX function that reads the cpu's processor clock and returns the value as a 64-bit number. This code requires a value that gives the number of ticks per microsecond which can be retrieved before the shot by calling the function `get_cpu_speed`.

- The value of the current time is checked in a while loop until it changes. A timeout needs to be specified so that the shot won't hang up if something goes wrong and the time is not updated.

- The SAVE_TIMING macro includes a standard function (found in the file common/save_timing.h) which calculates the amount of time it takes to execute

each function in the function vector and the total amount of time it takes to execute a cycle.

17.0.2 get_new_data

This real time function is usually the second function in the function vector. Its purpose is to read and prepare the next set of data that is collected on the processor. `rtheap->currenttime` is set to the time that was previously waited for in `wait_for_new_time`. The raw data is put into `rtheap->dmabuffer`, transferred to `rtheap->datavector` where a baseline should be subtracted. The value from `rtheap->datavector` can also be transferred to `rtheap->physicsvector` where it gets converted to physics units.

```
/*
*****************************************************************************
SUBROUTINE: get_new_data

Get the time and a new set of data.  Subtract the baseline
and fill the datavector and the physicsvector.

Global variables needed here (all are set in the init function):
    cycle_times            int array giving the cycle times for each
                           cpu in microseconds
    number_of_digitizers   the number of digitizers on this processor
    digitizer_indices      int array giving indices into the datavector
                           for each digitizer channel on this processor
    baseline_scratch       scratch parameter data block filled in by the
                           baselinedata algorithm
    btop                   parameter data block giving the factors to
                           convert each channel from bits to physics units
*****************************************************************************
*/
void get_new_data(struct rt_heap_misc *rtheap)
{
char *dma_region;
int chnl, i, offset;
unsigned int previous_cycle_time;
unsigned int convert_cycle_time;
```

```
    dma_region = (char *)rtheap->dmabuffer;
    offset = rtheap->offset_times;
    dmadata = (short *)rtheap->dmabuffer;
/*
get the time and a new set of data
*/
    /*
    save the previous cycle time
    */
    previous_cycle_time = rtheap->currenttime;

    if(rtheap->software_test_mode==SOFTWARE_TEST_MODE)
    {
        convert_cycle_time = previous_cycle_time + cycle_times[CPU_NUM-1];
        if(cycle_times[CPU_NUM-1] == 0)
        {
           get_remote_time(&convert_cycle_time, rtheap);
           if(rtheap->return_status != 0) return;
        }

for(chnl=0;chnl<rtheap->numrawdata;chnl++)
{
    dmadata[chnl] = 1000;
}

        memcpy(&dma_region[offset+DMA_CONVERTCYCLETIME*sizeof(int)],
               &convert_cycle_time,sizeof(int));
        memcpy(&dma_region[offset+DMA_PREVIOUSCYCLETIME*sizeof(int)],
               &previous_cycle_time,sizeof(int));
    }
    else if(rtheap->software_test_mode==SIMULATION_TEST_MODE)
         /* simserver mode, executes command on host process */
         /* two times in DMA region are filled in */
    {
#ifdef STANDALONE
strcpy(shmctlseg->command,"GET_DATA_FROM_HOST");
sem_set(sem_id,0); sem_wait(sem_id,1);
```

```
#else
        host_read_data_set_simserver();
/* note: dmadata and CONVERTCYCLETIME set by simserver */
#endif
        memcpy(&dma_region[offset+DMA_PREVIOUSCYCLETIME*sizeof(int)],
               &previous_cycle_time,sizeof(int));
    }
    else /* if(rtheap->hardware_test_mode==1) or normal operations mode */
    {
#ifdef PCS_RT_CPU
convert_cycle_time = previous_cycle_time + cycle_times[CPU_NUM-1];
        if(cycle_times[CPU_NUM-1] == 0)
        {
           get_remote_time(&convert_cycle_time, rtheap);
           if(rtheap->return_status != 0) return;
        }

if(number_of_digitizers > 0)
{
         int status = 0;

   status = dtacq_read(rtheap,&convert_cycle_time,dmadata);
           if(status != 0)
           {
              rtheap->return_status = status;
              return;
           }
}
#endif

        memcpy(&dma_region[offset+DMA_CONVERTCYCLETIME*sizeof(int)],
               &convert_cycle_time,sizeof(int));
        memcpy(&dma_region[offset+DMA_PREVIOUSCYCLETIME*sizeof(int)],
               &previous_cycle_time,sizeof(int));
    }
/*
Set the previoustime and the currenttime in the rtheap.
*/
```

```
    memcpy(&rtheap->previoustime,
           &dma_region[offset+DMA_PREVIOUSCYCLETIME*sizeof(int)],
           sizeof(int));
    memcpy(&rtheap->currenttime,
           &dma_region[offset+DMA_CONVERTCYCLETIME*sizeof(int)],
           sizeof(int));
    (rtheap->alldata)->time = rtheap->currenttime;
/*
Subtract the baseline from the dmadata array (which has values
in raw counts) and stick the value into the datavector.
baseline_scratch is a parameter data block that has the baseline
values in raw counts for each channel.
*/
    i = 0;
    while(digitizer_indices[i] != -99)
    {
        chnl = digitizer_indices[i];
        if(chnl > -1)
        {
           rtheap->datavector[chnl] =
               dmadata[i] - baseline_scratch->sdata[i];
        }
        i++;
    }
/*
Convert the raw data value from the datavector to physics units
and store into the physicsvector.  Only locally acquired data
need to be moved since any data from other processors gets sent
already in physics units.
digitizer_indices is a parameter data block holding the indices
into the datavector for each channel collected on this processor.
btop is a parameter data block that has conversion factors
form bits to physics units for each channel.
*/
    i = 0;
    while(digitizer_indices[i] != -99)
    {
        chnl = digitizer_indices[i];
```

```
        if(chnl > -1)
           rtheap->physicsvector[chnl] = btop[chnl]*rtheap->datavector[chnl];
        i++;
    }
/*
Determine offset into pid lookup table based on delta of
current and previous cycle times.
*/
    default_set_pid_index(rtheap);
/*
Copy the communication buffer contents.
*/
    default_copy_combuffer(rtheap);
/*
Check if another cpu has aborted the shot.  When it
does, a message is sent to all cpus and the
cpu_return_status flag for this cpu will be non-zero.
*/
    if(<another cpu has aborted the shot>)
    {
        rtheap->return_status = TIMETOSTOP_RETURN;
        return;
    }
}
```

There are several things to point out about the above code.

- The values of previoustime and currenttime in the rtheap structure are set differently depending on what the operating mode is: software test, hardware test, or operations.
- Functions `host_data_set_simserver` and `dtacq_read` need to put the raw data into the dmaregion. The former also has to set DMA_CONVERTCYCLETIME in the dmaregion.
- The image built to run on the real time processors can be a combination of the host_cpu and the realtime code. This was found to be the most efficient method and avoids the necessary communication between two separate programs. A macro PCS_RT_CPU indicates this combination and is used around any code that only executes on the real time processor.

- The cycle_times array is an array of ints giving the cycle times of each processor in microseconds. A cycle time of 0 is a special case which means that the time of the cycle varies. It is used when a cycle time is long (like the slow loop rtEFIT). The function `get_remote_time` will return the time (wherever it comes from). There is no waiting here since that was done previously.

- The function `default_set_pid_index` is found in realmain.c. It sets the value of `rtheap->dtoffset` which is used in the older and obsolete pid functions.

- The function `default_copy_combuffer` is found in realmain.c. It copies the values of `rtheap->adcombuffer` which may have been updated by other processors into `rtheap->adcombufloop` which is where values should be read from.

17.0.3 baselinedata

This real time function follows the function `get_new_data`. Its purpose is to sum up the raw data for each channel starting at one or more "trigger" times. These times are usually before the discharge starts. Then when enough samples have been collected, the average value for each channel is calculated and stored as a baseline value.

This algorithm should always be running as the default algorithm in the Data Acquisition category. Only one phase should be used in order to avoid problems with phase specific parameter data which is needed by the algorithm.

This algorithm is usually the location of the data item `operating setup data` which is parameter data required by the infrastructure.

This algorithm is usually the location of waveforms that specify the sample interval for each cpu over the course of the shot. The sample interval is the interval at which raw data and processed data (errors, shape vector data, commands, etc.) are stored for archiving after the shot. These waveforms replace the single values of sample interval and first sample time found in the data item `operating setup data`. These waveforms are useful when the cycle time of a processor is so fast that all samples cannot be saved due to space limitiations. These waveforms allow the interval to change so that all samples can be collected at a particularly interesting time during the discharge.

This algorithm is usually the location of waveforms that specify the fast data sampling interval for each cpu over the course of the shot. The fast data sampling interval is the interval at which the fast data are stored for archiving after the shot. These waveforms replace the single value of fast data start time found in the data item `operating setup data` (all samples are then saved).

This algorithm is usually the location of the `save_timing` data items. These data items allow for the measurement of how long each function takes and the overall time of each cycle. The code for this is located in the file `common/save_timing.h` and must be included in numerous places in the algorithm. Also, the compile line option `-DSAVE_TIMING` must be included in the build of the PCS processes.

17.0.4 operating setup data

The Data Acquisition algorithm is usually the location of the parameter data item `operating setup data` which is required by the infrastructure to specify the shot number as well as various values for each cpu.

The installation requires two variables be defined usually in the file `installdefs.h`:

```
/*
Variables required by the infrastructure and used
by the baselinedata algorithm.
*/
#define MAX_CPUS 24

struct diagnostic_infra diagnostic_infra_default = {
         0,             /* hardware test mode */
         0,             /* software test mode */
         900001,        /* test shot number */
         7000000        /* time to stop feedback in microseconds */
    };

struct diagnostic_cpu_infra diagnostic_cpu_infra_default[MAX_CPUS] = {
   /* first sample time, number of raw and processed samples to save,
      sample interval, fast data start time, number of fast data samples,
      routine to save fast data, flag indicating whether to save the sfile */
        {-3000000, 40000,   250, -1000000, 12*1024, 0, 0}, /* cpu 1 */
        { -100000,100000,   100, -1000000,  0*1024, 0, 0}, /* cpu 2 */
        {-3000000, 40000,   250, -1000000,  0*1024, 0, 0}, /* cpu 3 */
        {-1000000,100000,   100, -1000000,  0*1024, 0, 0}, /* cpu 4 */
        {  100000, 40000,   250, -1000000,  0*1024, 0, 0}, /* cpu 5 */
        {       0,     0,     0,        0,       0, 0, 0},
    };
```

```
/*
Other variables needed by the baselinedata algorithm.
*/
int cycle_times[MAX_CPUS] = {250, 50, 250, 50, 0, }; /* in usec */

int save_timing_defaults[MAX_CPUS] = {60000, 200000, 60000, 200000, 60000, };
```

Note that the sample intervals for cpus 2 and 4 are 100, but the cycle times for these cpus are 50. Thus, all data are not archived. Also, note that cpu 5 has a cycling time of 0 which means that the cycle time varies.

The amount of space required by the save_timing is given here as the number of samples to save. This specifies the amount of space to set aside for the timing information for each function in the function vector.

17.0.5 pass_data

This real time function follows the function `get_new_data` in the function vector. Its purpose is to send data to other processors in some manner. This is only necessary if all data required by each processor is not available.

For DIII-D, many processors do not have access to required data and digitizers are spread throughout the system. A general scheme was set up that requires each algorithm to specify which channels are required in real time. These are specified in a section of the algorithm delineated by the DATADEFS macro. Here, another macro DATA_NEEDED specifies the name of the channel and the virtual cpu that requires it. A program then collects all this information by requesting only the DATADEFS sections of all algorithms and creates an include file that specifies which processors need which data.

This file is then included in the baselinedata algorithm which uses it to create parameter data blocks giving the indices of the channels in the datavector that need to be passed from the processor that collects the data to processors that need the data.

This function then moves the data from the datavector to a write buffer that is used to send this information to each processor that needs it. The currenttime is always the last value in this block of data so that the receiver can match the time with its own time.

17.0.6 get_remote_data

This real time function follows the function `get_new_data` in the function vector. Its purpose is to wait for data that other processors pass. This is only necessary if some data required by the processor is not collected locally.

For DIII-D, many processors do not have access to required data and digitizers are spread throughout the system. A general scheme was set up that requires each algorithm to specify which channels are required in real time. These are specified in a section of the algorithm delineated by the DATADEFS macro. Here, another macro DATA_NEEDED specifies the name of the channel and the virtual cpu that requires it. A program then collects all this information by requesting only the DATADEFS sections of all algorithms and creates an include file that specifies which processors need which data.

This file is then included in the baselinedata algorithm which uses it to create parameter data blocks giving the indices of the channels in the datavector that are needed by processors which do not collect the channels. Other parameter data blocks are created with the starting locations and lengths of the data that will be written.

The area where this data is written to is a section of the install_buffer. This function waits for this data to arrive. The time value is always the last value in the buffer so this function waits for the time value to agree with the current time. Note that a slower processor can wait for data from another processor whose cycle time is faster than or equal, but a faster processor should never wait for data from a slower processor. The faster processor can simply grab what is in the data buffer and use it.

17.0.7 save_samples

This real time function MUST be the LAST function in the function vector. This function is responsible for updating the pointers to the various data sections, (DMAregion, shapevector, errorvector, etc.) in the real time heap. Most of the code required can be found in two functions located in the file realmain.c. Here is an example of this function which uses sample interval waveforms.

```
/*
*************************************************************************
SUBROUTINE: save_samples

Execute the steps required after all functions have been called
for each time step.  Note that this MUST be the last function
in the function vector.
```

```
Global variables needed here (all are set in the init function):
    minimum_cycle_time     cycle time of digitizer or if no digitizer,
                           cycle time of fastest processor that provides
                           this cpu its data
    total_cycles           counter giving number of cycles
*****************************************************************************
*/
static int previous_interval;
void save_samples(struct rt_heap_misc *rtheap)
{
int sample_interval;
int *istargets;

/*
   The time interval
   between saved samples is given by an integer step target vector
   element in units of fine scale clock ticks. If the sample interval
   is negative, then the data are not saved. If the sample interval
   is 0, every sample is saved.  If the sample interval is positive,
   then the data for this cycle are saved if the time is greater
   than the time to save the next sample.  Whenever the sample
   interval changes, the data for the next cycle are always saved.
   Data after that are saved at times that are as close as
   possible to a multiple of the sample interval after the time
   of the first sample saved for the current sample interval.
*/
   istargets = (int *)rtheap->adtarget;
#define TARGET_TO_USE(CPU) CONCAT(IST_ACQ_SAMPLEINTERVAL_CPU_,CPU)
   sample_interval = istargets[TARGET_TO_USE(CPU_NUM) - 1];
#undef TARGET_TO_USE

   if((sample_interval == 0) ||                 /* save every cycle */
     ((sample_interval != previous_interval) &&
      (total_cycles != 0)))   /* interval changed */
      rtheap->datasample_flag = 0xfffffffb;
   previous_interval = sample_interval;
/*
```

```
Pass sample_interval along in the rtheap.
Note that sample_interval is signed, rtheap->sampleincrement is unsigned.
*/
   if(sample_interval > 0)
      rtheap->sampleincrement = sample_interval;
   else
      rtheap->sampleincrement = 0;
/*
The fast data sample interval is also given by an
integer step target vector element.  Pass its value
along in the rtheap.
*/
#define TARGET_TO_USE(CPU) CONCAT(IST_ACQ_FASTSAMPLEINT_CPU_,CPU)
   rtheap->fastdatainterval = istargets[TARGET_TO_USE(CPU_NUM) - 1];
#undef TARGET_TO_USE
/*
Call the default function to setup whether samples should be saved.
*/
   default_save_samples_setup(rtheap);
/*
Call the default function that actually saves the data
by changing all the data pointers.
*/
   default_save_samples(rtheap);
/*
Do any installation specific stuff here such as
toggling the watchdog timer.
*/
   total_cycles++;
/*
Increment the time to update the continuous target vector.
This allows the calculations to be done at the end of the cycle
rather than when the currenttime changes.
*/
   if(rtheap->software_test_mode != SIMULATION_TEST_MODE)
   {
      rtheap->update_time = rtheap->currenttime + minimum_cycle_time;
   }
```

```
   return;
}
```

18 Access control

The plasma control system allows multiple users to make changes to the current setup of the next shot. This allows flexability, but it also can be dangerous. For example, someone could inadvertently make changes to the next shot while working on a future shot setup.

Access control can be used to better control access to the next shot setup. Each user can be granted write access to one or more categories. The user can then make changes to data in only those categories. In addition, users are notified with messages on the user interface when someone else makes changes to the data item being viewed.

There are two options under the "File" menu in the next shot user interface:

- "change access control" will bring up an editor which allows changes to the access control. The editor allows the current "operator" to grant or deny access to a particular user. See the User's Manual for more details on how to use the editor.

- "show access control" will display a user's access for each category. If the user has been denied write access to the current data item, then the normal "apply" button will be replaced with a "R-only" (read only) button. This allows the user to make changes to the display but does not allow these changes to be applied to the waveform server's setup.

One of the useful features of access control is the notification that another user has made a change to the data that one is viewing or modifying. A message indicating what item was changed and who changed it is written on the user interface at the top of the plotting region. The color of the apply/cancel region changes and the labels in this area change also. In addition, a message is written on the plotting region and the color and labels change whenever someone else BEGINS to make a change to the data item being viewed. This feature makes it easier for two or more users to make changes to the same data items.

In general, each category grants access to individual users or "groups" of users. A user is identified by their computer username. A group can then be granted access to one or more categories. For example, a group called "beam operators" could be

assigned only to the "Neutral Beam" category. A group called "current operators", which is designated as the main group, would be assigned to all categories. Then a user could be assigned to be just a "beam operator" or a "current operator".

The access control editor displays a list of categories and which users and groups have been assigned to each. It has another section which displays the users in each group. The term "leader" is used to designate a single user who is in charge of operations. A user can choose their username from a droplist of possible leaders and apply this change to become the current leader. The menubar at the top of the editor contains a single button "Add user to group..." which contains the usernames of everyone who can be granted access. Choosing a name from this droplist automatically adds that user to the main group. The editor also has a widget to allow a username to be typed in and added to a group or category. And there is a widget to allow the removal of a username from a group or category.

Access control requires a default setup file located in the data directory called "access_control.data". This file, if it exists, is read by the waveform server at startup. This file represents the default settings for access control. This file must be edited to add or remove users. There are two parts to the file: the keyword part followed by one line for each known category. Comment lines begin with an exclamation point. Each line has the following format:

```
keyword=value,value,...
```

```
where "keyword" is one of the following:
```

- LEADER_NAME: the name to designate the PCS "leader" ("physics operator" ar DIII-D but perhaps a different title elsewhere). The "leader" is the person who controls access to the PCS.
- CURRENT_LEADER: username of the default leader; the waveform server will grant this user access by default.
- CURRENT_NAME: a group name used for current users, for example, "current operators"; if a user is in this group, then that user is allowed to become the "current" leader.
- GROUP_NAMES: list of group names; each group with default values must then follow, one group per line. Note that the value of the LEADER_NAME must be one of the group names.

- OTHERS_WITH_ACCESS: users who can grant themselves access without becoming the "current leader"; allows for programmers to do testing.

The values to specify are computer usernames. The username is what is passed to the waveserver when a change command is given.

The keyword section is then followed by the default access for each category in the PCS. The category name to use is the "major identifer" found in the "cat_alg" definition in the category's master file. For each category identifer specify the group(s) that have default write access to the data items in that category.

Example of the access_control.data file:

```
CURRENT_LEADER=hyatt
LEADER_NAME=physics operator
CURRENT_NAME=current operators
OTHERS_WITH_ACCESS=
GROUP_NAMES=current operators,physics operators,beam operators,chief operators,comput
current operators=hyatt
physics operators=ferron,hyatt,jackson
beam operators=gohil,reimerde
chief operators=holtrop,leer,nagy,taylorpl
computer operators=johnsonb,penaflor,piglowsk
others=
!
acq=current operators,ACQUSER,chief operators
alarm=current operators,chief operators
beam=current operators,TIMCON,beam operators,chief operators
bsupply=current operators,chief operators
cer=current operators,chief operators
density=current operators,chief operators
equil=current operators,chief operators
fsupply=current operators,chief operators
gas=current operators,chief operators
gyrotron=current operators,chief operators
ip=current operators,chief operators
lpellet=current operators,chief operators
mode=current operators,chief operators
n1=current operators,chief operators
ntm=current operators,chief operators
```

```
rf=current operators,chief operators
shape=current operators,chief operators
system=current operators,chief operators
thomson=current operators,chief operators
```

For more information see the file access_control_editor.pro located in the infrastructure.

19 Command files

It may be useful to set up files that contain commands that do useful tasks. For example, turning gas on and off or bringing up an idl interface that allows a user to go through a checklist. These commands can be a simple change of vertices for a waveform, or much more complicated such as a multi-step checklist.

These commands can be put into files in the data directory which end in the file extension ".com". When the interface for the next shot is initialized, the idl code looks for these command files and, if any exist, a menu option "Commands" will be added to the main menu bar. The name of the file is used as the command name.

Files with the ".com" extension are used for the operations next shot. Likewise, those files with an extension of ".com_runsa" are used to create the "Commands" menu for a next shot in standalone mode. Future shots combine these files with duplicates removed.

For example, a simple way to turn off gas puffing from the PCS is to add a permanent phase in the GAS category called "nogas" where all gas puffing is off. Then choose only this phase in the Primary phase sequence. A simple command to set the vertex for the Primary phase sequence to the nogas phase could be put into the file "turn_gas_OFF.com":

```
skip message  ;skip popup confirming selection of this command
SET_VERTICES,'Gas','','Primary',[-9.0],['NoGas'],1,-9
```

And another file could have a simple command to turn the gas back on in the file "turn_gas_ON.com":

```
skip message  ;skip popup confirming selection of this command
SET_VERTICES,'Gas','','Primary',[-9.0],['ShotStart'],1,-9
```

With these two files in the data directory the "Command" menu would be created with these two options.

A more complicated example would be the restoring of a shot in standalone mode. In some cases, the restoring of a shot requires restoring certain items that do not automatically get restored, such as the "snap setups" parameter data for the rtEFIT, because they are marked NORESTORE. To duplicate such a shot in standalone mode, these items need to be restored. A command file can be written to handle all the steps necessary. An example follows.

```
skip message  -  skip popup confirming selection
;
;Commands to restore a shot as it was when it was taken.
;
IDL>topbase = widget_base(title='Restore shot',xoffset=20,yoffset=20)
IDL>widget_control,/realize
IDL>xmanager,'restore_shot',topbase
;
IDL>getsetup,dx,3
IDL>get_wavest,wavest
IDL>if(wavest.last_error ne 0) then widget_control,topbase,/destroy
IDL>if(wavest.last_error ne 0) then return
IDL>widget_control,/hourglass
IDL>shot=wavest.last_shot
IDL>setup=wavest.last_setup
;
;Load all categories
IDL>SEND_RESTORE_MESSAGE,dx,error,shot,setup,'load all categories',$
    ' ',' ',' ',' ',' ',' ',' ',' ',' ',' ',' '
;
IDL>IGNORE = 1    ; now ignore errors
;
IDL>widget_control,/hourglass

;Load the BreakDown phases
IDL>SEND_RESTORE_MESSAGE,dx,error,shot,setup,'load selected phase','F Power Supplies'
    ' ','BreakDown',' ',' ','F Power Supplies',' ','BreakDown',' ',' '
IDL>SEND_RESTORE_MESSAGE,dx,error,shot,setup,'load selected phase','Discharge Shape',
    ' ','BreakDown',' ',' ','Discharge Shape',' ','BreakDown',' ',' '
```

```
;
;Load the snap setups from the shot
;IDL>SEND_RESTORE_MESSAGE,dx,error,shot,setup,'load selected data item',$
    'Equilibrium',' ','efit','Switches','Snap setups',$
    'Equilibrium',' ','efit','Switches','Snap setups'
;
;Load the patch data from the shot
IDL>IF(shot gt 0 and shot lt 108040) then $
    patch_editor_get_data,dx,error,shot else $
    SEND_RESTORE_MESSAGE,dx,error,shot,setup,'load selected data item',$
    'System',' ','ShotStart','All Waveforms','patch data',$
    'System',' ','ShotStart','All Waveforms','patch data'
;
;Load the calibration data from the shot
;IDL>IF(shot gt 0 and shot lt 108040) then get_acq_data,dx,error,shot
IDL>IF(shot gt 0 and shot lt 108040) then print,'MUST LOAD CALIBRATION DATA'+string(7
IDL>IF(shot ge 108040) then SEND_RESTORE_MESSAGE,dx,error,shot,setup,$
    'load selected data item','Data Acquisition',' ','ShotStart',$
    'Parameter Data','calibration data',$
    'Data Acquisition',' ','ShotStart','Parameter Data','calibration data'
;
;Correct the vertices for Ecoil_GP
IDL>IF(does_phase_exist(dx,'E coil','plasmaresp')) then $
    execute_command,dx,"SET_VERTICES,'E coil','plasmaresp',$
    'Ecoil_GP',[-8.990],[-20.0],1,-8.99",error
;
;Set to defaults the fastloop input files
;IDL>SET_PARAMDATA_DEFAULTS,dx,error,'Equilibrium','efit','fl input file :used names'
;
IDL>widget_control,topbase,/destroy
```

20 Change history

When changes are made to the PCS, a record is written to the "change history" which is part of the archived setup. This record contains the date and time of the change, the user who made it, and the details of the change, e.g., restore information, waveform name and vertices, or a change to a parameter data block (but with no detail about

what in the block was changed). In addition, for the operations version of the PCS, a record is written to the file found in the data directory (usually, /link/ops/data_v10 which can be a link to the installation-specific directory) called "history.txt". This file contains all changes made and can be consulted for details without restoring individual shot archives.

The user interface has an item under the Information menu called "list changes in current setup" which will show the change history. In addition, the "select data to load" window from the restore interface has an item under the Show menu called "show changes" which will list the change history for a given archived shot or future shot setup.

The change history is recorded in the format of an executable command. The window that displays the change history has an "execute" button which allows the user to click on most any line, press the execute button, and execute the highlighted command. Parameter data history lines are not executable since the detail of the change is not recorded.

The change history is cleared when all categories are set to their defaults or when all categories for a shot archive or future shot setup are restored. In the latter case, the restored change history is added to the current change history. This is seen in the display as lines that are slightly indented. Also, in the display are buttons to view the lines in a "brief" mode (where details are left off) and a filter box to allow filtering of the display.

Note that an installation can turn off all change history recording by specifying the compile line option -DNO_ARCHIVING_OF_CHANGES when building the infrastructure.

21 The current DIII-D plasma control system

Here is brief description of the current plasma control system at DIII-D (as of April, 2007). There are 19 "cpus" in today's system. There are 3 VME based processors which handle digital inputs (one of which is the watchdog), digital outputs, D/A converter channels, and communication with the Thomson and CER VME-based processors. The remaining 16 processors are all INTEL based off-the-shelf processors. All systems run a version of the LINUX operating system.

Data acquisition is mostly accomplished using several different types of D-TACQ digitizers. These digitizers write their data directly to DMA memory set aside by the operating system. The "tlatch" (or time) is read in different ways, either from DMA memory or by accessing the hardware through a PCI interface.

Each real time process cycles at a given cycle rate that matches the cycle rate of any digitizers that are local to the processor. Some processors do not have any digitizers and must receive the time and data from other processors. They can then cycle at the same rate as the slowest one that is passing data, or a multiple of the slowest cycle time. Some processors have a cycle rate of 0 which means that cycles vary depending on how long the real time code takes to execute.

The fastest processor in the system cycles at 11 μs (10 would be more desirable but the current hardware cannot handle it as yet). Three other processors cycle at 44 μs (to be in sync with the fastest one). Still another processor cycles at 2048 μs! Most of the other processors cycle at 250 μs.

The LINUX operating system was customized to allow for "real-time" mode. During real-time mode, all memory is locked in the system and all interrupts (except for the keyboard interrupt) are turned off so that the processor can cycle in real time. The keyboard interrupt allows for a function key to be pressed to "wake" up the system if something should go wrong and a processor needs to be taken out of real-time mode.

One processor is used to send data to a "real-time scope" which displays plots during the discharge. The rtEFIT also provides data to the scope processor which allows the boundary display to be displayed over an image of the DIII-D tokamak.

The processes that run on the real time processors are a combined image containing the host_cpu code and the realtime code. These processes are given the names `pcs_rt_cpu%` where % is the cpu number.

All the real time processors are connected in a Myrinet network. This network can transfer data as fast as 2 Gigabits per second.

The gateway host, another LINUX processor, runs the waveform server, message server, and lockout server. Users sign on this gateway machine to run the PCS user interface.

22 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 to determine where this document has been changed since it was last referenced.

1. 9/24/98:

    - Addition of the message `REALTIME_ROUTINE_STARTING` to the shot cycle (page 21).

- The host real time process can abort the shot if it cannot complete shot setup properly (page 22).
- New section on assembly language code (Sec. 23.1, page 94).

2. 2/14/2001:

   - A new section (Sec. 6) was added with some notes about how a mixture of processor architectures is handled in the PCS. This section needs to be expanded and will eventually be included under the topic of "interprocess communication."

3. 9/19/2001:

   - A section describing the code for archiving data was added (Sec. 15). This section was mostly written by Bob Johnson.

4. 1/21/2002:

   - Section 11 has been filled in. This section discusses the management of the memory on the real time processor.
   - A pdf version of this document is now available. See Sec. .

5. 3/8/2002:

   - Section 10.2 was added. This section describes the installation-specific functions that must be provided to execute tasks on behalf of the `host_cpu` process.
   - Section 13.3 was added. This section describes the installation-specific functions that must be provided to execute tasks on behalf of the infrastructure on the real time processor.
   - Section 13.5 was added. This section discusses how to implement the software testing modes of operation.
   - Section 13.6 was added. This section discusses how to implement the hardware testing mode of operation.

6. 9/18/2003

   - Section 3.1 was updated to remove references to files no longer used and add references to other files.

- Section 3.2 was added. This section discusses the dma_region and how the installation can define its own.
- Section 3.3 was added. This section discusses the install_buffer and how the installation can define its own.
- A very sketchy description of how to make the logo image file was added in Sec. 4.1.1.

7. 12/03/2003

   - Limitations Section 2 was added.

8. 04/03/2007 (RDJ)

   - Data Acquisition Category Section 17 was added.

9. 04/04/2007 (RDJ)

   - Added Section 21 "The current DIII-D plasma control system".
   - Section 23.1 "Overview of assembly language routines" was made obsolete.
   - Section 23.2 "The cycle time in the DIII-D plasma control system" was made obsolete.

10. 04/05/2007 (RDJ)

    - Modified section 13.1 on the real time code.
    - Added Section 13.2.1 about real time function new_shot_phase_tick.
    - Added Section 13.2.2 about real time function new_continuous_target_c.
    - Added Section 13.2.3 about real time function startup_state.
    - Added Section 13.2.4 about real time function update_the_state.
    - Added Section 13.2.5 about real time function time_critical.

11. 04/06/2007 (RDJ)

    - System Category Section 16 was added.
    - Added to the S file access routines Section 14.
    - Updated the sections on the functions found in files `pcshotfile.c` 15.2 and `close_shotfile` 15.3.

- Added files used in the `lockserver` program 9.
- Added paragraph about hardware simulation 13.6.

12. 06/25/2007 (RDJ)

    - Added section outlining use of access control 18.
    - Added section outlining use of command files 19.

13. 08/21/2007 (RDJ)

    - Changed references of host_realtime to host_cpu and host_realtime_cpu to host_cpu.
    - Added section describing the change history for a setup 20.

14. 10/07/2008 (RDJ)

    - Filled in section for the message server 5.1.
    - Filled in section for the user interface server 5.2.
    - Filled in section for interprocess communication 7.
    - Filled in section for executing the real time code 10.5.
    - Filled in section for archiving data 10.6.

23 Obsolete features

23.1 Overview of assembly language routines

Some of the code that runs in real time is written in assembly language. A description of the reasons for this is given in the comments at the top of the file `timecrit.s`. This section gives an overview of the assembly language files.

1. Assembly language files that contain infrastructure code.

   (a) `asm_symbols.s`: Defines symbols for the assembly language routines. Primarily these are offsets into C language structures, but there are a few installation specific definitions having to do with the Supercard and the data acquisition.

   (b) `timecrit.s`: This contains the main routine for executing the real time control cycle.

(c) `phasetick.s`: Contains the routines called each time a period equal to a phase clock tick has elapsed.

(d) `timecrit_routines.s`: Contains a mixture of infrastructure and installation specific routines. The data acquisition routines are here. These are really installation specific. There are also a few infrastructure routines that are used by the main routine in `timecrit.s`.

2. Assembly language files that are presently included in the infrastructure directory but which actually contain code that is really installation specific. These files should be moved eventually to the installation specific directory.

(a) `algutility.s`: Routines that are used by various control algorithms. Particularly, there are routines for sending data to the D/A converters. Also, the PID routines are here. These could be considered infrastructure.

(b) `readstatus.s`: Read the trigger input status board on the digital i/o board.

(c) `write_beamgas_enable.s`: Write to the word on the digital i/o board that drives the enable lines for the neutral beams and the gas valves.

(d) `write_toggle.s`: Handles the watchdog toggle bit on the digital i/o board.

(e) `infra_utils.s`: The first comment line says that this contains infrastructure code, but because much of it really depends on having a Supercard on a VME bus, these routines should really be considered installation specific.

(f) `returndata.s`: Apparently an obsolete routine.

(g) `setdavalue.s`: Set the value of a d/a converter channel.

(h) `write_enable.s`: Write to the watchdog bit enable word on the digital i/o board. It isn't clear whether this file is used anymore.

(i) `beamtoggle.s`: Obsolete routine.

23.2 The cycle time in the DIII-D plasma control system

Here is brief description of the factors determining the cycle time in the plasma control system. The format here is simply an outline of what is done in a single cycle with notes on how long each step takes.

The bottom line is that the minimum cycle times range from 13.2 μs for a small system to 25.4 μs for a large system. During this time, about 7.5 μs of application

code work could be done without extending the cycle time. So, the effective overhead ranges from 5.7 μs to 17.9 μs.

The 25.4 μs value is in agreement with tests performed on the plasma control system.

Examples:

1. For a system with only 16 data channels and some plasma control system-specific features removed (no digital i/o data, no watchdog timer and no communication buffer), the shortest cycle time is about 13.2 μs. During this time there are 7.75 μs available in which application code work could be done without extending the cycle time. Any extra application code work would increase this time.

2. For a system which is the same as the version 10.8 of the plasma control system (current as of 5/24/95) which has 192 data channels, the time required is about 25.4 μs. In this time, there are about 7.42 μs in which application code work could be done without extending the cycle time.

Note that the absolute shortest cycle time is dominated by the A/D conversion time of the digitizer, time to acquire the data and convert it to floating point and the time to execute the application code.

In the case of a specific, well controlled, feedback control application, it is possible to run very close to the minimum possible cycle time. However, as a practical matter, the minimum cycle time is achieved under only the correct conditions. The actual cycle time is most dependent on how the cache memory is used. Some routines are tuned to run at maximum speed assuming that certain data structures are located in the cache memory. When the actual application code executes it will strongly affect the use of the cache memory and access to the memory bus so the cycle time actually achieved will increase somewhat from the minimum possible value.

Here are the steps in a feedback cycle with the approximate required times:

1. Wait for the next set of data to arrive in SuperCard memory: 0.5 μs.

2. Read the digital i/o data value: about 1 μs.

3. Copy the data, subtract the baseline value and convert to floating point: time is 40 data values per μs plus overhead (0.8 μs for 16 data values, 5.75 μs for 192 data values).

4. Copy the communication buffer (not necessary on a single cpu system): 0.6 μs for 16 element buffer.

5. Overhead: 0.5 μs.

6. Execute each application code function: 0.2 μs for each function plus time required for the application code.

7. Start acquisition of the next set of data: optimum is 0.4 μs (if the data from the cycle is not saved).

   Then the A/D conversion is performed, requiring 8.6 μs (determined by the model of digitizer).

8. Call more application code functions if desired. 8.6 μs of application code work can be done between here and step 11 below with no addition to the cycle time while the A/D conversion completes (this includes steps 9 and 10).

   Note that the output to the D/A converter channels is included in the functions of the application code. About 0.9 μs is required for each write to a D/A converter channel.

9. Touch the watchdog timer bits: 0.33 μs.

10. Do some between cycle things that provide the flexibility of the control system to archive data, have time varying target values, switch phases etc.

    optimum is 0.85 μs. (a few μs longer if it is time to save data, a phase clock tick has passed or it is time for target vector calculation)

11. After the A/D conversion completes the data are written to the SuperCard memory in a dma operation. Assuming that the application code requires access to memory, the application code is blocked during this time (if the application code depends only on cache memory then it continues to run during this time). DMA transfer time is $0.4+1.6*$(round up to integer)[(number of channels)/64] μs which is 2.0 μs for a 16 channel system and 5.2 μs for a 192 channel system.

12. Finish any routines that were blocked by the dma transfer in step 11 (i.e. this is any of the work from steps 8, 9, 10 above that didn't finish before the dma transfer started).

13. Loop back to step 1.