Hall C Reference

The CEBAF Test Package: A Symbolic and Dynamic Test, Histogram, and Parameter Package for On- and Off-line Particle Physics Data Analysis

Stephen A. Wood

August 3, 1995

Abstract

The CEBAF Test Package (CTP), a set of analysis tools for nuclear physics data acquisition systems, has been developed. This package allows an analysis code to receive parameter settings, cuts definitions, and histogram definitions through ASCII files loaded at run time, or through a remote procedure call interface. These tools, modeled very loosely on the LAMPF Q test, histogram and dynamic parameter packages, allow one to change the parameters of data analysis without stopping and recompiling the data acquisition system. CTP includes a variable registration system that allows the same variable names to be used in the data acquisition system and in the definition files.

1 Introduction

On-line data analysis software for nuclear physics data acquisition systems often share several common elements.

- A method of setting and modifying parameters that are used by the analysis software. (Particle masses, detector geometries, analysis control flags, etc.)

- A specialized method of placing cuts and other tests upon raw and analyzed data that allows these cuts to be modified without changing the analyzer source code.

- A method of specifying histograms (conditioned by cuts) that again does not require changing the source code to modify.

A good example of a data acquisition system with these elements is the LAMPF Q system [?]. The Q system allows parameters, tests and histograms to be defined in ASCII files that are read when the data acquisition system is started. All of these elements may then be modified while the system is active through the use of shared memory.

An advantage of such analysis packages is the ability to make the definitions of histograms, cuts and parameters more concise. Furthermore, since these definitions are separated from the data acquisition source code, users who are not familiar with the details of the analysis code can make changes without the risk of introducing bugs while an experiment is in progress. A major disadvantage however, is that histogram, test and parameter packages typically access data internal to the analysis code through specific large arrays. In order to make the analysis code more readable, equivalence statements are typically used, making maintenance of a large experiment analyzer cumbersome.

In the CEBAF Test Package (CTP), the same variable names are used in both the source code and in the CTP definition files. This is accomplished by requiring the analyzer developer to "register" each variable and array that is to be accessible to CTP. Once a variable has been registered, its value(s) can be set through a CTP parameter definition file. For registered variables that are calculated each event, the variable may be used in histogram or test definitions. Since the definition files refer to variables by the same names as used in the analysis software, they can be made readable when sensible variable names are chosen by the programmer.

Any registered variable as well as the test and histogram definitions may be dynamically read and modified by tasks that communicate via standard networking calls. As this package is implemented in C and uses SUN RPC networking, it is portable. This package works with CODA [?], but may be used with other data acquisition systems with stand alone offline analysis codes.

2 CTP Definition Files

The main aspect of CTP seen by a user is the definition file(s) that are used to define parameters, tests, histograms and reports. All of these types of definitions may be mixed within a single file, with the various types delineated by `begin` and `end` lines. Each of these control lines has keywords to denote the type of definition and a name for the block of definitions. The use of these blocks is discussed below.

2.1 Parameter Definitions

The parameter component of CTP supplies a means by which the parameters, constants, control flags, string variables, etc., of a data acquisition or analysis system may be initialized with readable ASCII files. It is similar to the NAMELIST package available with many Fortran compilers. The following is a sample of a definition file that sets several parameters used in an analyzer.

```
begin parm constants
  proton = 938.272   ; <- Comment character
  neutron = 939.57
  pion = 139.57
  pizero = 134.97
  ; The following fills an array
  offsets = 0.34, 5.7, 0.1, -.4
            0.0, .1356, -1.3, 8.6
  ; array fills may take multiple lines
  thmin = -50
  thmax = 50
  phimin = -40
  phimax = 40
  ; Lines from begin to end are called a block
  hbook_filename = 'hist.dat' ; String
  report_filename = 'endrun.txt' ; String
end parm constants
```

In order for the variables of these names to be set in the analyzer, these variables must be registered by the analyzer before this block of parameters is "booked". This registering and booking is described in section III.

2.2 Test/Cut Definitions

A test package provides a means of gathering the logic decisions of event analysis in a concise yet readable ASCII file maintained separately from the analyzer code. The results of tests are typically used to condition the incrementing of histograms. Scalers of test results are useful for computing yields and efficiencies.

In the following example of a test block, `fptheta`, `fpphi`, `mass`, and the array `njunk` are event data that are computed each event by the analyzer. The variables `thmin`, `thmax`, `phimin`, and `phimax` are parameters that are set in the previous example.

```
begin test hms
 goodthe = (fptheta > thmin)&&(fptheta < thmax)
 goodphi = (fpphi > phimin)&&(fpphi < phimax)
 goodphase = goodthe && goodphi
       ; Acceptable phase space of particle
 electron = mass < 10
       ; Must be an electron
 clean(1) = njunk(1) < 2
       ; Less than 2 junk hits per plane
 clean(2) = njunk(2) < 2
 clean(3) = njunk(3) < 2
 clean(4) = njunk(4) < 2
 clean(5) = njunk(5) < 2
 clean(6) = njunk(6) < 2
 allclean = clean(1)&&clean(2)&&clean(3)
       &&clean(4)&&clean(5)&&clean(6)
 goodparticle = goodphase&&electron&&allclean
 missingmass = ebeam - e1 - e2
end test
```

The expressions on the right hand side use any combination of a set unary and binary operators similar to the standard C operators. These operators are shown in table 1. A limited set of single argument functions are also

Table 1: Operators allowed in test expressions. In order of precedence.

<table>
<tr><td>()</td><td>Fortran Array element reference</td></tr>
<tr><td>[]</td><td>C Array element reference</td></tr>
<tr><td>-</td><td>Unary Minus</td></tr>
<tr><td>!</td><td>Logical Negation</td></tr>
<tr><td>~</td><td>Ones Complement</td></tr>
<tr><td>*</td><td>Multiplication</td></tr>
<tr><td>/</td><td>Division</td></tr>
<tr><td>//</td><td>Division, result integerized</td></tr>
<tr><td>%</td><td>Modulus</td></tr>
<tr><td>+</td><td>Addition</td></tr>
<tr><td>-</td><td>Subtraction</td></tr>
<tr><td><<</td><td>Left Shift</td></tr>
<tr><td>>></td><td>Right Shift</td></tr>
<tr><td><</td><td>Less than</td></tr>
<tr><td><=</td><td>Less than or equal to</td></tr>
<tr><td>></td><td>Greater than</td></tr>
<tr><td>>=</td><td>Greater than or equal</td></tr>
<tr><td>==</td><td>Equal to</td></tr>
<tr><td>!=</td><td>Not equal to</td></tr>
<tr><td>&</td><td>Bitwise AND</td></tr>
<tr><td>^</td><td>Bitwise XOR</td></tr>
<tr><td>|</td><td>Bitwise OR</td></tr>
<tr><td>&&</td><td>Logical AND</td></tr>
<tr><td>^^</td><td>Logical XOR</td></tr>
<tr><td>||</td><td>Logical OR</td></tr>
<tr><td>=</td><td>Assignment operator</td></tr>
<tr><td>,</td><td>C Comma operator</td></tr>
</table>

Table 2: Functions allowed in test expressions.

`abs()`	Absolute value
`sqrt()`	Square root

available as shown in table 2. The results of the expressions will be logical or arithmetic values depending on the operators. The variables on the left hand side of the expressions are the test results. The variables need not be registered by the analyzer, but must be if the test result is needed directly by the analyzer or is a real value. Test result variables that have not been registered may still be used in subsequent test expressions and as test flags in the histogram package, but these variables will all be established as integers.

2.3 Histogram Definitions

The histogram part of the analysis package is essentially a book keeper for HBOOK [?]. It reads a list of what variables are to be histogrammed over what limits. Using the information in the configuration file, histograms are booked with standard HBOOK calls. For each event, all the histograms (subject to test conditions) are incremented. The programmer is thus saved from managing the source code for booking and filling.

```
begin hist hms
 ;Histname, Data source, nbins, low, high, test_flag $title
 h1,mass,1000,0,2000,goodphase $Mass distribution of good tracks
 h2,fptheta,(thmax-thmin)/2,thmin,thmax,allclean $Angular distr at focal plane

 ;Histname, X Data source, Y Data source, nxbins, xlow, xhigh
 ; , nybins, ylow, yhigh, test_flag $title
 twod,mass,fptheta,100,0,2000,25,-50,50,allclean $A 2-D histogram
end hist
```

The format for defining histograms is at present a simple comma separated list of arguments. The first item on each line is the histogram “name”. This name may be used to display the histogram in the histogram display application. The second item is the variable name of the quantity to be histogrammed. The analyzer must set this variable for each event. The next three items are the number of bins followed by the lower and upper limits of the histogram. (The binning arguments have the same meaning as in the HBOOK1 call.) These three arguments may be expressions that include registered variables. These expressions are evaluated at the time that the histograms are booked. The last item is a test flag. The histogram will only be incremented for events where the test flag is true. Both the variable to histogram and the test flag may be an array named followed by an index in parentheses. This index may also be an expression that is evaluated at the time the histogram is booked. If the line defining a histogram contains a `$`, the remainder of the line will be used as the histogram title. A similar format exists for 2 dimensional histograms.

2.4 Report Templates

A typical analyzer writes a text file at the end of a run that contains various summary information about the run. This "print out" might contain information such as test scaler values, hardware scalers, parameters, and calculation of physics results based on these quantities. CTP includes a "report generator" that can be adequete for many such print outs. This allows the content and format of summary sheets to be changed without modifying the analyzer code.

```
begin report endrun
The test block "hms" was called {block.test.hms} times.
{goodphase.scaler} events passed the phase space cuts for
an efficiency of {goodphase.scaler/block.test.hms}
e(good theta) = {x=goodthe.scaler/block.test.hms}
e(good phi)   = {y=goodphi.scaler/block.test.hms}
e(electrons)  = {electron.scaler/block.test.hms}

Time of run ~ {scalers(159)*timnorm} sec.

S1X+                        Counts              Rate
scalers(17) = S1X1+ = {scalers(17)} = {scalers(17)/(scalers(159)*timnorm)}
scalers(25) = S1X2+ = {scalers(25)} = {scalers(25)/(scalers(159)*timnorm)}
scalers(18) = S1X3+ = {scalers(18)} = {scalers(18)/(scalers(159)*timnorm)}
scalers(26) = S1X4+ = {scalers(26)} = {scalers(26)/(scalers(159)*timnorm)}
end report endrun
```

Note that reports can access test scalers.

Describe the vlist hack used to list out all registered variables.

2.5 Accessing data in variable length “Hit” lists

2.6 Include files

A CTP definition file may contain one or more definition blocks of any of the above types. To aid with modularity or readability of the file, include statements may be placed in the definition file. The include statement has the form:

`#include "filename"`

These include statements may be used to include whole definition blocks, or one or more may be placed within blocks. The include files can be nested to arbitrary depths.

2.7 Registering Local Variables

The variables used in CTP input files generally need to be registered in the analyzer. However, as some of the examples here show, there are many uses for CTP variables that are not known to the analyzer code. The uses of these variables, known as local variables, might be used as test results variables, locally calculated quantities to histogram, constants used to setting other parameters or array indices, etc.

When a CTP parameter, test, or gethit block is booked, any unregistered result variables will be created automatically. However, CTP may not always create the variables with the desired data type. Furthermore, CTP will print out a warnings when variables has been registered automatically. As these warnings can be disconcerting to the user and they can guide in identifying variables that the analyzer programmer has failed to properly register, a means has been provided to register variables within CTP blocks.

Some examples:

```
#integer.test allclean
#integer goodparticle, electron
#double.event missingmass
#real positions(10), emass
```

The allowed types are `integer`, `real`, `double`, and `string`. The types may be optionally qualified with a class, usually `parm`, `test`, or `event`.

3 Fortran Application Interface

The CTP package may be used with analysis software written in either C or Fortran. We describe here only the Fortran interface which is presently available HP-UX, Ultrix and Linux.

3.1 Registration Commands

As the variables used in parameter, test and histogram definitions are intended to refer to variables of the same name used by an analysis code, they must be linked in some manner. This is accomplished by having the analyzer "register" any variable that is required by the desciption files. This registration must be performed before the parameters, tests and histograms are booked. Four byte integers and four byte real numbers may be registered. A variable is registered by calling the appropriate registration function with a string giving its name and a pointer to the variable's location in memory. At the time a variable is registered, it must be given a class through the use of the appropriate subroutine call.

```
character*length name
character*LEN title
integer*4 ierr


integer*4 ilen, rlen
parameter (ilen=LEN)
parameter (rlen=LEN)
integer*4 ival
integer*4 iarray(ilen)
real*4 rval
real*4 rarray(rlen)
character*LEN string

ierr=regeventint(name,ival,title)              ! I*4 event variable
ierr=regeventreal(name,rval,title)             ! R*4 event
ierr=regeventdouble(name,rval,title)           ! R*8 event
ierr=regparmint(name,ival,title)               ! I*4 parameter
ierr=regparmreal(name,rval,title)              ! R*4 parameter
```

```
ierr=regparmdouble(name,rval,title)              ! R*8 parameter
ierr=regtestint(name,ival,title)                 ! I*4 test result
ierr=regtestreal(name,rval,title)                ! R*4 test result
ierr=regtestdouble(name,rval,title)              ! R*8 test result
ierr=regeventintarray(name,iarray,ilen,title)    ! I*4 event array
ierr=regeventrealarray(name,rarray,rlen,title)   ! R*4 event array
ierr=regeventdoublearray(name,rarray,rlen,title) ! R*8 event array
ierr=regparmintarray(name,iarray,ilen,title)     ! I*4 parameter array
ierr=regparmrealarray(name,rarray,rlen,title)    ! R*4 parameter array
ierr=regparmdoublearray(name,rarray,rlen,title)  ! R*8 parameter array
ierr=regtestmintarray(name,iarray,ilen,title)    ! I*4 test result array
ierr=regtestrealarray(name,rarray,rlen,title)    ! R*4 test result array
ierr=regtestdoublearray(name,rarray,rlen,title)  ! R*8 test result array

ierr=regparmstring(name,string,title)            ! Character string
```

The title argument is an arbitrary string that may be used to label a variable. It may be omitted by using a zero instead of a string. The above functions return an error code, zero for success, and not zero for failure. No specific meanings have been assigned yet for the failure error codes.

The memory locations of the variables that are registered must be static. In Fortran this can be guaranteed by placing all variables that are registered in common blocks. (Or by listing the variables in SAVE statements.)

A test result variable does need not to be registered unless one wants access to the test result from within the source code. An example would be where one wants to condition analysis (peformed after executing a test block) on the result of one of the tests.

If a test result is to be a real value, then the variable or array must be registered before the tests are booked.

3.2 Booking Commands

The following calls are used to load and book the parameters, test definitions, histogram definitions, and report templates. The application must establish the histogramming area with HLIMIT or HLIMAP before the histograms are booked.

```
character *(*) filename
```

```
character *(*) histname
integer*4 id
integer*4 ierr

ierr=thload(filename)
ierr=thbook()
ierr=thwhalias(filename)
id = thgetid(histname)
```

The `thload` routine loads all of the blocks of parameters, tests and histograms contained in a single file. This file may have any number of blocks of separate types. Several calls to `thload` may be made to load definitions contained in several files. The loading does not actually set parameter values or book tests or histograms, but merely makes a copy of the files in memory. A single call to thbook will then book all of the blocks that have been loaded and not yet booked. After booking, the routine `thwhalias` may be used to write a file of PAW [?] alias commands that the user may load into PAW so that histograms can be refered to by name rather than by id number.

3.3 Test Execution and Test Scalers

In a typical analyzer, as each event is analyzed, all the defined tests will be evaluated. The results of these tests are then available to be used as conditions on histogram incrementing. CTP calls are also available to accumulate scalers of the number of time each test was evaluated as true (not zero.)

All CTP calls return an `integer*4` error code. Zero is returned for successful execute. No specific failure codes are defined except that a failure will be indicated by a non-zero return code.

The following calls are available for test evaluation and for incrementing test result scalers:

```
character *(*) blockname
integer*4 ierr

ierr=thtstexe()            ! Execute all tests
ierr=thtstexeb(blockname)  ! Execute a single block of tests

ierr=thtstins()            ! Increment all test scalers
ierr=thtstinsb(blockname)  ! Increment scalers for on block
```

The calls without arguments will cause all the blocks (in alphabetical order) to be processed. For the routines that end with `b`, only a single block is executed.

The test scalers must be cleared initially before any calls to increment the scalers are made or before a new run is analyzed. As with test execution and scaler incrementing, all test scalers or just those associated with a specific test block may be cleared.

```
character *(*) blockname
integer*4 ierr

ierr=thtstcls()            ! Clear all test scalers
ierr=thtstclsb(blockname)  ! Clear test scalers for
                           ! a single block of tests
```

Test test results may be cleared before tests are executed with the following routines.

```
character *(*) blockname
integer*4 ierr

ierr=thtstclr()            ! Clear all test flags
ierr=thtstclrb(blockname)  ! Clear test flags a single block
```

Clearing the test results is not required as the execution of each test will result in a definite value. However, there can be situations where clearing the test results is important. Two of the situations are:

- If a particular block of tests is not executed for every event, but if its test results are used in other test blocks or as conditions on histogram execution, the test results should be cleared so that they do not reflect test results from previous events.

- If some of the test results in a block are arrays, the test results flags for that block should be cleared. Otherwise, array elements that were not accessed for a particular event may have values from previous events.

In writing the tests that go into CTP test blocks, a given test result name should generally only be used in one block. In particular particular care must be taken if arrays are used for test results. If mutually exclusive array elements (of a given array) are used in more than one test block, the use of test scalers and the clearing of test flags can have undesired results. This is because, the `ins`, `cls`, and `clr` routines will operate on an entire array even if only one element of that array is used as a test result in a given block. Since the indices of arrays can be variable as well as fixed, CTP has no easy way of determining which elements of these arrays to operate on (increment or clear.)

3.4 Histogram Filling

In a typical analyzer, as each event is analyzed, all the defined tests will be evaluated. Then the histograms will be filled with the results of the event analysis conditioned by the test flags. The following calls are available for test evaluation and histogram filling.

```
character *(*) blockname
integer*4 ierr

ierr=thhstexe()
ierr=thhstexeb(blockname)
```

The CTP histogram filling calls should be called after tests have been executed.

CTP has no routines for saving histograms to a file. This should be accomplished with

3.5 Gethit Routines

To fill the gethit value and test variables, two routines are available to execute a single gethit block or all gethit blocks.

```
character *(*) blockname
integer*4 ierr

ierr=thgethit()
ierr=thgethitb(blockname)
```

Calls to execute gethit blocks should generally be performed before test execution and histogram filling calls.

3.6 Report Generation

```
character *(*) blockname
integer*4 ierr

ierr=threp(blockname,filename)
ierr=threpa(blockname,filename)
```

The `threp` will copy the text contained in the specified block into the specified output file, replacing strings insides of braces ({}) with the expressions contained within them. The routine `threpa` is identical except that the report is appended to the specified file, rather than overwriting it.

3.7 Sample Code

As a coding example, we give fragments of code that would work with then example CTP input files used above. (The parameter block "constants", the test block "hms" and , the histogram block "hms".) We assume that all three of the CTP blocks are contained in the single file `ctp.input`.

3.7.1 Sample Variable registration

Before the CTP input file can be read in, the variables used must be registered by the physics analysis code. We do that by a call to the following routine:

```
  subroutine regallvars()

  implicit none

  include 'parcommon.inc'
  include 'testcommon.inc'
  include 'eventcommon.inc'

  integer*4 regparmint, regparmreal
  integer*4 ierr
c
c Register the parameters used in the ``constants block''
c
  ierr = regparmreal('proton',proton,'Mass of the proton')
```

```
  ierr = regparmreal('neutron',neutron,'Mass of the neutron')
  ierr = regparmreal('pion',pion,'Mass of the charged pion')
  ierr = regparmreal('pizero',pizero,'Mass of the neutral pion')
  ierr = regparmrealarray('offsets',offsets,8,'An array of offsets')
  ierr = regparmreal('thmin',thmin,'Low end of theta cut')
  ierr = regparmreal('thmax',thmax,'High end of theta cut')
  ierr = regparmreal('phimin',phimin,'Low end of phi cut')
  ierr = regparmreal('phimax',phimax,'High end of phi cut')
  ierr = regparmstring('hbook_filename',hbook_filename,
'HBOOK Filename')
  ierr = regparmstring('report_filename',report_filename,
'Report Generator Filename')
c
c Register some test result variables.  Only the array clean is
c registered as CTP will automatically register test result variables
c as the test block is booked.  If you want to easily access
c
  ierr = regtestintarray('clean',clean,6,'Array of test results')
c
c Register variables that are recalculated each event which contain
c quantities that may be histogrammed.
c
  ierr = regeventreal('mass',mass,'Calculated mass')
  ierr = regeventreal('fptheta',fptheta,'Theta angle at focal plane')
c
  return
  end
```

3.7.2 Sample common blocks

The following are sample include files that put the registered variables into common blocks.

```
c
c parcommon.inc  -  Common block for registered parameters
c
  real*4 proton, neutron, pion, pizero
  real*4 offsets(8)
```

```
      real*4 thmin, thmax, phimin, phimax
      character*80 hbook_filename,report_filename
c
      common /parms/ proton, neutron, pion, pizero
     &      ,offsets, thmin, thmax, phimin, phimax
      common /strings/ hbook_filename,report_filename

c
c testcommon.inc - Common block for test results
c
      integer*4 clean(6)
c
      common /tests/ clean


c
c eventcommon.inc - Common for event variables
c
      real*4 mass, fptheta
c
      common /event/ mass,fptheta
```

3.7.3 Sample initialization code

The following is an example of a routine that calls the above routine to register all of the variables, and then loads and books the CTP files.

```
      subroutine initialize()

      implicit none
      integer*4 hmemor(20000)
      common /pawc/ hmemor
      integer*4 ierr

      call regallvars

      call hlimit(20000)
      ierr = thload('ctp.input')
```

```
      ierr = thbook()
      ierr = thwalias('paw.aliases')

      ierr = thtstcls()  ! Clear all test scalers

      return
      end
```

3.7.4 Example Event code

The following sample routine is called for each event.

```
      subroutine analyze_event

      implicit none

      include 'parcommon.inc'
      include 'testcommon.inc'
      include 'eventcommon.inc'

      integer*4 ierr
c OTHER DECLARATIONS

c GET THE EVENT

c DO EVENT ANALYSIS

      ierr = thtstexeb('hms') ! Exec tests

      ierr = thtstinsb('hms') ! Incr scalers

      ierr = thhstexeb('hms') ! Fill histograms

      return
      end
```

3.7.5 Example End of Run Code

After analyzing a set of data, an analyzer program will typically save the histograms a write out various calculations and statistics. The following example saves the histograms into a file and uses the CTP report generator to print some statistics.

```
subroutine end_analysis

implicit none

include 'parcommon.inc'
include 'testcommon.inc'
include 'eventcommon.inc'

integer*4 ierr

hbook calls to save file
threpout call.
```

4 Variable Registration System

As shown in section IIIA, there are separate calls for registering variables that are parameter constants, or event specific data. The reason for this is to allow variables to be grouped by function or "class". This is important because applications that are developed to communicate with the analyzer need the ability to build separate tables of variables for event data, parameters, test results, etc.

Internally, variable names are stored as a series of strings separated by periods. The first part of the name is the class, followed by the name followed by an optional attribute (`class.name[.attribute]`). (Attributes are usually not used in the definition files described in section 2) The class names for what has been discussed so far are `event`, `parm`, and `test`. For example, the variable `proton` is stored internally as `parm.proton`. It is important to note that in the source code, one continues to use the variable name proton. Similarly, in the input files for parameters, tests and histograms, the variable name without the class may be generally used. When these files are interpreted, assumptions are made about the variable class by context rules. The assumptions may be overridden by writing out the variable name with the class identifier included.

When a variable is registered, a number of attributes of this variable are saved along with the pointer to the actual value of the variable. These attributes are

- Name: The name of the variable including its class.
- Type: A flag indicating whether the variable is an integer, real or string.
- Size: For arrays or strings, the length. One for scalers.
- Flag: Flags indicating permissions for remote procedures to change this variable.
- Title: A title/description for the variable.
- Hooks: See below.
- Opaque: A pointer to extra data about the variable (such as scalers for test results).

5 RPC service

An application interface is currently being developed that allows access to the registered variables. Through these calls, one may get a list of registered variables or a list of a subset of the variables (e.g. by class). One may also read or set any of the registered variables. Through special hooks placed on a special class of variables, test and histogram definitions may also be changed through remote procedure calls.

5.1 Fortran RPC access routines

5.2 C RPC access routines

At present, the C interface to RPC is undocumented and may be subject to change.

As an example the call to retrieve a list of variables is

```
long daVarList(char *pattern, char ***listp, int *count);
```

This function will return a pointer to a list of all registered variables that begin with the string contained in `pattern`. The number of variables in the list will be returned in `count`. For example

```
status = daVarList("parm.", &list, &count);
```

will return a list of all variables in the parameter class.

Routines are also available to read or write any parameter. Since there is a networking overhead that limits RPC to several hundred calls a second, routines will also be provided that can set or read an arbitrary list of variables in one subroutine call. Since RPC calls are served in between the processing of events, the use of a single RPC call to read or write a list of variables insures that all variables returned or set correspond to a single event.

5.3 Class Parsing

When a string identifying a variable is passsed through an RPC call, the server follows a specific rule to interpret the variable name. If the string contains several periods in it, the server first searches for a registered variable of the name only up to the first period. If this is not found, the string up to the second period is searched for and so on. When a match is found, the server looks at the hook attributes for the registered variable. If the hook for the present function (read or write) is found, that function is called to handle the read or write request. If there are no hooks, then a default handler is called.

5.4 Attributes

From remote applications, one may add an attribute specifier to the end of a registered variable name. This is done by adding a period followed by the name of the attribute. The default attributes are `value`, `title`, `type`, `size`, `flags`, and `name`. If the variable or it's handler allows it, additional attributes may be specified. If no attribute is specified, then the value attribute is used.

5.5 Rebooking histograms and tests through RPC

When a block of tests or histograms is booked, a registered variable of the name `block`.class.blockname is created. (E.g., for the test block in section 2.2, the variable `block.test.hms` is created.) The value of this variable is the number of times that the block has been executed and the title is the entire string of characters from the input file between and including the begin and end lines.

In order to change the test definitions for this example test block, the remote application requests the value of `block.test.hms.title`. The remote application can then put this string in an editor widget where the user can edit the test definitions. The application then writes the new string to the variable `block.test.hms.title`. On the server, the variable `block.test` will have been defined with a special write hook. As well as resetting the title attribute of `block.test.hms`, the special handler will recompile the test code so that on the next event processed, the new tests will be in effect.

6 Hints for linking to CTP

[For HPUX, need to compile fortran programs with the +ppu flag. Hopefully, the cernlib libraries being used were compiled with this flag too.]

7 Portability

The CEBAF Test Package has been written using the GNU C compiler under Linux. It has been ported to HP-UX, Ultrix, and OSF-Alpha. It should easily port to other architectures, particularly if the GNU C compiler is used. The Fortran entry points into CTP may require some attention on new ports. The histogramming part of CTP requires the CERN HBOOK package. However, the use of HBOOK calls is limited to a few places in the source code, so use with a different histogramming package would not be difficult.

8 Availability

CTP is available for download with anonymous ftp from `ftp.cebaf.gov` as `/pub/hallc/ctp-*.tar` where * is the version number. A set of test programs is also available in the file `ctp_test.tar`.

9 Acknowledgements

The CTP package would not have been possible without many long discussions with Chip Watson. The variable name structure with classes and attributes is largely his design and is what makes full control of CTP through an RPC interface possible. Jie Chen is also acknowledged for writing a demonstration CTP RPC client that demonstrates the power of using RPC to interact with a CTP endowed analyzer. Dave Abbott must be acknowledged as being the first user to use CTP in a working data acquisition system. John Arrington has

The comments and suggestions of many others have also been valuable in designing and implimenting CTP as well as in improving the meager documentation.

References