Unified **P**arallel **S**oftware
User's Guide and Reference Manual[1]

`UPS_VERSION=v-02-07-05`

Date of this manual's printing: October 3, 2007

http://public.lanl.gov/ups

`ups-team@lanl.gov`

Abstract

UPS (Unified Parallel Software) is a collection of software tools (libraries, scripts, executables) that assist in parallel programming.

This consists of:

- libups.a
  C/Fortran callable routines for message passing (utilities written on top of MPI) and file IO (utilities written on top of HDF).
- libuserd-HDF.so
  EnSight user-defined reader for vizualizing data files written with UPS File IO.
- ups_libuserd_query, ups_libuserd_prep.pl, ups_libuserd_script.pl
  Executables/scripts to get information from data files and to simplify the use of EnSight on those data files.
- ups_io_rm/ups_io_cp
  Manipulate data files written with UPS File IO

These tools are portable to a wide variety of Unix platforms.

It is assumed that the reader has a general knowledge of the issues involved in writing applications on distributed memory, parallel processing computers. A good introduction to parallel programming can be found in "Designing and Building Parallel Programs" [3] by Ian Foster of Argonne National Laboratory. It is helpful, though not necessary, for the reader to be familiar with MPI[2] or PVM[4]. Good sources of information for these are "Using MPI" [6] and "PVM: Parallel Virtual Machine; A Users' Guide and Tutorial for Networked Parallel Computing" [5]. The UPS web page lists a variety of resources and references for parallel processing.

[1]LA-CC 03-041

Contents

1 COPYRIGHT **4**

2 Introduction **5**

3 Getting Started **7**
3.1 Writing a UPS Program . . . 7
3.1.1 Writing a UPS Program in C/C++ . . . 8
3.1.2 Writing a UPS Program in Fortran . . . 9
3.1.3 Writing a UPS Program in Fortran-77 . . . 10
3.2 Compiling/Running a UPS Program . . . 11
3.2.1 General Notes on Libraries . . . 11
3.2.2 Compiling/Running a UPS Program: Basic Example . . . 12

4 Administration Details **13**
4.1 Install Directory Structure . . . 13
4.2 Email Information . . . 14
4.3 Future Plans . . . 15

5 Use Details **16**
5.1 Name Space Conventions . . . 16
5.2 Initialization/Termination . . . 17
5.3 Routine argument list ordering . . . 17
5.4 Opaque Object Handles . . . 17
5.5 Programming Language Issues . . . 17
5.5.1 Fortran Interface . . . 17
5.5.2 UPS_DT_INT8 vs UPS_DT_LONG . . . 18
5.5.3 Passing Identical Arguments from Fortran . . . 18
5.6 Communication Contexts . . . 18
5.7 Conflicts With Other Packages . . . 18
5.8 Error Reporting . . . 19
5.9 UPS Version Consistency . . . 20

6 Packages **22**
6.1 General Package (AA) . . . 22
6.2 Communication Package (CM) . . . 23
6.2.1 Shared Memory Example . . . 23
6.3 Data Parallel Package (DP) . . . 27
6.4 Datatype Package (DT) . . . 28
6.5 Error Package (ER) . . . 29
6.6 Gather Scatter Package (GS) . . . 30
6.7 File IO Package (IO) . . . 32
6.8 Utility Package (UT) . . . 35

A Acknowledgements **37**

B UPS Constants **38**

C Reference Manual **89**
C.1 Organization of reference pages 89
C.2 General 91
C.3 Communication 102
C.4 Data parallel 126
C.5 Datatypes 139
C.6 Error handling 140
C.7 Gather/scatter 146
C.8 File IO 175
C.9 Utilities 230
C.10 Reference Pages Index 254

List of Figures

1 UPS Program in C/C++ 8
2 UPS Program in Fortran 9
3 UPS Program in Fortran-77 10
4 Compiling a UPS Program (Fortran: Basic Example) 12
5 Running a UPS Program: Basic Example) 12
6 Gather/scatter view of a domain and its decomposition into 3 processes. . . 30
7 Gather/scatter view of a single cell 31

List of Tables

1 UPS Packages 6
2 Brief Description of a Compile Line: Basic Example 12
3 Example Directory Structure 13
4 Filename Descriptions 14
5 Email Lists 15
6 Name Space Conventions 16

1 COPYRIGHT

2 Introduction

UPS, an acronym for "Unified Parallel Software", is a library of routines designed to help the application developer create efficient, extensible, and robust large scale parallel programs for physics simulations.

Some parallel programming models attempt to hide the parallelism from the application writer, while others require that the application writer work at the lowest levels. UPS falls in between: it is designed to expose the parallel environment in a natural way while abstracting away some of the complexities.

Currently, UPS is built on MPI[2] (which provides process startup and a default mechanism for transferring data). Althought there are no plancs to do so, UPS could be ported to a PVM[4] version as it also provides process control and data movement.

Although most of UPS is written in C, UPS has interfaces that allow it to be called from C, C++, Fortran, and Fortran-77.

UPS is designed to be readily portable to an environment that has a message passing interface (providing process control and data movement) and a C compiler.

The UPS project relies on the inclusion of software developed by team members as well as other groups throughout the laboratory and from outside sources. For information regarding how to contribute to the UPS project, see the UPS Developer's Guide [1].

As will be explained in detail (see section 6 page 22), UPS functionality is divided into packages:

UPS Packages	
Package	*Description*
General (AA)	Includes init/terminate routines. *description*: section 6.1, page 22 *reference*: section C.2, page 91
Communication (CM)	Process information, Collective operations. *description*: section 6.2, page 23 *reference*: section C.3, page 102
Data Parallel (DP)	Operations on a distributed vector. *description*: section 6.3, page 27 *reference*: section C.4, page 126
Data Type (DT)	Datatype information. *description*: section 6.4, page 28 *reference*: section C.5, page 139
Error (ER)	Error handling. *description*: section 6.5, page 29 *reference*: section C.6, page 140
Gather/Scatter (GS)	Access to globally distributed data. *description*: section 6.6, page 30 *reference*: section C.7, page 146
File IO (IO)	File IO. *description*: section 6.7, page 32 *reference*: section C.8, page 175
Utilities (UT)	Other useful functions. *description*: section 6.8, page 35 *reference*: section C.9, page 230

Table 1: UPS Packages

3 Getting Started

In order to use UPS, three things must be accomplished:

1. Write a program that uses UPS.
2. Compile that program.
3. Run that program.

This section shows an example of how to use UPS. Covered will be specific programming languages, architectures, and message passing interfaces.

3.1 Writing a UPS Program

The main differences when programming to UPS in different languages:

- Access to UPS constants ("include" versus "use")
- Function names (eg "UPSF_AA_INIT" versus "UPS_AA_Init")
- Function Arguments (eg "ierr" is included in Fortran argument list)

The following code examples illustrate these points.

3.1.1 Writing a UPS Program in C/C++

The following example is written in C (the UPS C and C++ interfaces are the same). C++ Name mangling issues are taken care of in the prototype include file "ups.h". Future plans are to provide a specific C++ style interface.

Access to UPS constants and strong type checking for function calls is provided by "include ups.h". Function calls are of the form:

```
UPS_<PACKAGE>_<CAPITAL LETTER><lower case letters>
     ierr = UPS_AA_Init(...);
```

The return value is the error code.

```
/* program my_ups_program */

#include <stdio.h>
#include <stdlib.h>

/* include the ups C/C++ include file */
#include "ups.h"

void main( int argc, char **argv )
{
   int ierr[3], penum;

   /* init UPS - note error flag is return value */
   ierr[0] = UPS_AA_Init( argc, argv );

   /* get the pe number - note error flag is return value */
   ierr[1] = UPS_CM_Get_penum( &penum );

   /* print hello */
   printf( "Hello from %d\n", penum );

   /* terminate UPS - note error flag is return value */
   ierr[2] = UPS_AA_Terminate();

   /* check errors */
   if ( ierr[0] + ierr[1] + ierr[2] != UPS_OK ) exit(-1);
   exit(0);
}
```

Figure 1: UPS Program in C/C++

3.1.2 Writing a UPS Program in Fortran

The following example is written in Fortran. Access to UPS constants and strong type checking for function calls is provided by "use UPS". Function calls are of the form:

```
UPSF_<PACKAGE>_<CAPITAL LETTERS>
     call UPSF_AA_INIT(...,ierr,...);
```

The error code is a parameter.

```
! program my_ups_program
program my_ups_program

  ! use the fortran UPS module
  use UPS

  implicit none

  integer(KIND=UPS_KIND_INT4) :: ierr(3), penum

  ! init UPS - note error flag is a parameter
  call UPSF_AA_INIT( ierr(1) )

  ! get the pe number - note error flag is a parameter
  call UPSF_CM_GET_PENUM( penum, ierr(2) )

  ! print hello
  print*, 'Hello from ',penum

  ! terminate UPS - note error flag is a parameter
  call UPSF_AA_TERMINATE( ierr(3) )

  ! check errors
  if ( ierr(1) + ierr(2) + ierr(3) /= UPS_OK ) stop

  end program my_ups_program
```

Figure 2: UPS Program in Fortran

3.1.3 Writing a UPS Program in Fortran-77

The following example is written in Fortran-77. Access to UPS constants is provided by “include upsf77.h”. The Fortran-77 version does not have strong type checking of function calls. So, at compile time, you will not be warned for incorrect/missing arguments. Function calls are of the form:

```
UPS_<PACKAGE>_<CAPITAL LETTERS>
     call UPS_AA_INIT(...,ierr,...);
```

The error code is a parameter.

```
!234567  program my_ups_program
      program my_ups_program

      implicit none

!     include the ups Fortran-77 include file
      include 'upsf77.h'

      integer ierr(3), penum

!     init UPS - note error flag is a parameter
      call UPS_AA_INIT( ierr(1) )

!     get the pe number - note error flag is a parameter
      call UPS_CM_GET_PENUM( penum, ierr(2) )

!     print hello
      print*, 'Hello from ',penum

!     terminate UPS - note error flag is a parameter
      call UPS_AA_TERMINATE( ierr(3) )

!     check errors
      if ( ierr(1) + ierr(2) + ierr(3) .ne. UPS_OK ) stop

      end
```

Figure 3: UPS Program in Fortran-77

3.2 Compiling/Running a UPS Program

The specifics of compiling and running on different systems will vary and depend on the requirements of the user. The following is a brief overview on how to do simple compilations/runs. Additional sources of information (eg system administrators) may need to be consulted.

3.2.1 General Notes on Libraries

UPS typically layers upon several other libraries. Examples include a message passing library (e.g. MPI), some system libraries, and some math libraries. On some systems these libraries are automatically included, on others the library name is required, and on others the path to the library is required. We discuss these situations and issues in this section.

1. **Communication component**

   Procedures in the communication component are used to move data between parallel processes. The default mechanism for this is a library adhering to the Message Passing Interface definition[2]. This library is installed on most machines of intertest, and is linked as `-lmpi`. Future versions of UPS may use other data movement capabilities on certain machines, e.g. shmem, which may require linking other libraries.

   Some architectures require additional libraries:

   - mpi: -lmpi
   - mpich: -lmpich
   - Compaq with mpi: -lmpi -lelan

   Note that several other UPS components use the same data movement library, e.g. data parallel and gather/scatter.

2. **IO component**

   The IO component layers on top of HDF. So, you will need to add in `-lhdf5`. HDF comes with UPS and its library is in the same location as `libups.a`.

   On the sun, I found you probably need `-laio` as well.

3. **Miscellaneous Libraries**

   UPS tries to use the POSIX standard where possible. On some architectures, you might need to include:

   - POSIX libraries: -lrt, -lposix4, -lnsl, -lsocket, ...
   - Math libraries: -lm
   - Fortran libraries: -lfortran, -lftn, -lU77 ...

3.2.2 Compiling/Running a UPS Program: Basic Example

The following example is a basic compile line. For specific architectures/languages, additional include paths/libraries will have to be added.

If, for example, the user wishes to use the C example, it might be possible to simply replace the "f90" compiler and ".F" suffix with the "C" compiler and ".c" suffix.

```
f90 my_ups_program.F -o my_ups_program
    -I /usr/projects/ups/latest/include/SGI64_mpi
    -I /usr/projects/ups/latest/include/SGI64_mpi/Fortran_mods
    -L /usr/projects/ups/latest/lib/SGI64_mpi
    -lups -lhdf5 -lmpi
```

Figure 4: Compiling a UPS Program (Fortran: Basic Example)

The following table gives some explanation to the compile line. See Administration Details (section 4 page 13) for more information.

Brief Description of Compile Line: Basic Example	
Item	*Description*
f90 and .F	Fortran example (change for appropriate language)
/usr/projects/ups	Where ups is locally installed.
latest/	Points to latest version.
SGI64_mpi	specifies build options directory.
-l[libraries]	The libraries needed. The order in which libraries are specified is important.

Table 2: Brief Description of a Compile Line: Basic Example

Running a UPS program will be the same as running a MPI program. Again, this will vary from system to system. In general, to do a simple run on 4 processes, users will type in something like:

```
mpirun -np 4 my_ups_program
```

Figure 5: Running a UPS Program: Basic Example)

4 Administration Details

This section contains information that deals with the administrative side of UPS.

To see basic information about using UPS, see section 3 page 7. To see detailed information about using UPS, see section 5 page 16.

4.1 Install Directory Structure

The following table shows a typical directory structure. The version numbers have changed to protect the innocent.

Example Directory Structure			
latest v-01-01-01 v-01-02-00			
v-01-02-01	bin	SGI64_mpi –> ../SGI64_mpi/bin *other type dirs*	
	doc	doc_UPS	DeveloperGuide.ps UserGuide.ps
		Documentation for other products *(eg EnSight Reader: doc_libuserd-HDF)*	
	include	SGI64_mpi –> ../SGI64_mpi/include *other type dirs*	
	lib	SGI64_mpi –> ../SGI64_mpi/lib *other type dirs*	
	script	ups_aa_statistics_plot.pl *other scripts*	
	SGI64_mpi	include	UPS.mod UPS_CONST_MOD.mod ups.h upsf.h upsf77.h *other include-like files*
		lib	libups.a *other library files*
		bin	h5ls *other bin files*
	SGIn32_mpi *other type dirs*		
other version directories			

Table 3: Example Directory Structure

Filename Descriptions	
Filename	**Description**
`all_changes.txt`	All changes made to all files since last release.
`DeveloperGuide.ps`	Describes the infrastructure of the UPS code.
`doc`	Directory containing the guides.
`include`	Directory containing platform dependant include dirs.
`latest`	Symbolic link pointing to latest version directory.
`lib`	Directory containing platform dependant library dirs.
`libups.a`	Contains the UPS routines to which users link.
`script`	Directory containing useful scripts.
`SGI64_mpi`	Directory containing files dependent upon platform, message passing interface, bit addressing, ... This names signifies it was compiled on an SGI with 64 bit addressing using MPI.
`SGIn32_mpi`	Directory containing files dependent upon platform, message passing interface, bit addressing, ... This names signifies it was compiled on an SGI with 32 new bit addressing using MPI.
`ups.h`	C/C++ include file. Contains constants and prototypes.
`UPS.mod`	Fortran use file. Contains constants and interfaces.
`ups_aa_statistics_plot.pl`	Perl plotting script that take the output file ups_log.ps and uses gnuplot to create ups_log.ps
`UPS_CONST_MOD.mod`	Fortran use file (contained inside UPS.mod). Contains constants only.
`upsf.h`	Fortran include file. Contains Fortran style constants.
`upsf77.h`	Fortran-77 include file. Contains Fortran-77 style constants.
`UserGuide.ps`	This guide.
`v-01-02-01`	UPS version. (one can also access the UPS_VERSION variable). rightmost numbers - minor changes. center numbers - moderate changes. leftmost numbers - major changes.

Table 4: Filename Descriptions

4.2 Email Information

Below is a list of important email address through which you can contact the UPS team (and through which the UPS team can contact users as a whole).

Email Lists	
Item	*Description*
`ups-team@lanl.gov`	Sends mail to the UPS team.
	Purpose: ask questions to team.
	Purpose: submit bug reports.
`ups-users@lanl.gov`	Sends mail to other users.
	Purpose: UPS team announcements.
	Purpose: User forum.
	Also forwarded to `ups-team@lan.gov`.

Table 5: Email Lists

4.3 Future Plans

UPS is an ongoing project - reflecting the changing needs of the user community. If users wish to have a certain functionality included in UPS, they can contact the UPS team at `ups-team@lanl.gov` (see section 4.2 page 14).

Listed below are capabilities that could be added to UPS:

- **Linear/Non-Linear Solvers**
- **Load Balancing Algorithms**
- **PVM Implementation**

To see a brief overview of current capabilities, see section 2 page 5. A more detailed overview can be seen in section 6 page 22.

5 Use Details

This section contains information that deals with some additional specifics of using UPS in applications.

To see basic information about using UPS, see section 3 page 7. To see detailed information about administrative issues, see section 4 page 13.

5.1 Name Space Conventions

UPS symbols (eg function names, constants, ...) follow specific naming rules in order for users to avoid name space collisions.

Name Space Conventions	
Item	*Format/Example*
Constant[2]	UPS_[W] `UPS_ERROR_GS_SCATTER`
C Routine	UPS_(2L)[3]_(L)[w] `UPS_CM_Get_penum`
Fortran Routine	UPSF_(2L)[3]_[W] `UPSF_CM_GET_PENUM`
Fortran-77 Routine	UPS_(2L)[3]_[W] `UPS_CM_GET_PENUM`
Internal UPS Routines[4]	upsp_[w], upsi_[w], or upspi_[w] `upspi_cm_wait_for_flag`
Internal UPS Variables[4]	upsp_[w], upsi_[w], upspi_[w], UPSP_[W], UPSI_[W], or UPSPI_[W] `upsp_cm`
key	
expression	*meaning*
[]	at least one occurrence
()	single occurrence
(l)	lower case letter
(L)	upper case letter
(2l)	2 lower case letters
(2L)	2 upper case letters
[w]	lower case letters, underscores, and/or numbers
[W]	upper case letters, underscores, and/or numbers
[wW]	lower/upper case letters, underscores, and/or numbers

Table 6: Name Space Conventions

[2]Constants include datatypes, error codes, operations,... (see section B page 38 for a listing)

[3]Often, the "_(2L)_" will denote a package (eg communication - `_CM_`)

[4]Routines and variables only used by UPS

5.2 Initialization/Termination

See Getting Started (section 3 page 7) for an example using the init/terminate functions (`UPS_AA_Init` [page 92] and `UPS_AA_Terminate` [page 100]).

Many UPS functions require that the underlying communication protocol (currently MPI) be initialized. Upon initialization, UPS detects if this protocol has already been initialized by the user. If it has not, then UPS initializes the protocol. `UPS_AA_Terminate` will only terminate the underlying communication protocol if UPS initialized it.

UPS does not interfere with the users ability to issue commands to the underlying communication protocol themselves. See Communication Contexts (section 5.6 page 18) for an additional information regarding this issue.

5.3 Routine argument list ordering

UPS functions have their arguments in the following order:

1. Input data followed by its qualifiers.
2. Output data followed by its qualifiers.
3. Qualifiers that apply to both input and output data.
4. Error code (for Fortran - return value for C)

5.4 Opaque Object Handles

Many functions in UPS include an argument that the user should think of as a tag, or handle, to an object that further defines what the argument is. These handles, often integers, identify “opaque” objects, objects whose size, shape, and content are not visible or meaningful to the user. The user then uses this handle for further operations.

Examples of opaque objects in UPS are the routines UPS_GS_Setup (section C.7 page 165) and UPS_IO_Info_create (section C.8 page 214).

5.5 Programming Language Issues

5.5.1 Fortran Interface

The goal of the Fortran interface is to provide strong type checking.

Due to the function interface capability of Fortran, the argument lists of some UPS Fortran routines are different than the Fortran77 routines. Most of these changes come from the optional mask argument (since UPS can detect these in Fortran).

In fact, due to “optional arguments”, some routines have been combined into one interface (see UPS_DP_Combiner and UPS_DP_Combinerm - section C.4 pages 126 and 127).

It was decided that since many codes do not use up the entire allocated array, specification of count is required. Also, since it is impossible to distinguish between a two dimensional array and a set of elements of the datatype UPS_DT_2<datatype>, we made the specification of the datatype required.

5.5.2 UPS_DT_INT8 vs UPS_DT_LONG

Often an 8 byte integer is referred to as a long. In fact, when given something of the type UPS_DT_INT8, we often process as a C long variable. When compiling under 64 bit addressing, we equate the constants UPS_DT_INT8 and UPS_DT_LONG. Unfortunately, when compiling under n32 bit addressing, the C long is the same size as an int (UPS_DT_INT).

We (correctly) no longer equate the constants UPS_DT_LONG and UPS_DT_INT8. So, if you are using Fortran, use the Fortran constants (C users must use the C constants). If you are using standard integers or 64 bit addressing, this will not affect you.

5.5.3 Passing Identical Arguments from Fortran

Some routines (eg UPSF_CM_REDUCE) behave differently when the same argument is used for both input and output. However, in Fortran just passing the same argument for the input and output buffers does not guarantee that the arguments passed down to UPS point to the same memory location. Some Fortran compilers (eg Fujitsu Fortran compiler on linux) create a copy of the arguments and send those to UPS (thus making UPS see them as different buffers). Passing in the first element of an argument (eg x(1)) when going through the UPS Fortran interface might help "fool" the compiler.

5.6 Communication Contexts

Having different communicator contexts is important for the following reasons:

- **compartmentalize tasks**

  It is often desirable to have certain subsets of processes be in charge of different tasks (I/O, different physics packages, ...)

- **Message Insulation**

  This allows for 2 messages with the same process destination and tag to be distinguished.

UPS uses the message insulation feature to allow it to coexist with the underlying message passing protocol. For example, users do not have to worry about MPI messages sent by UPS conflicting with their own MPI messages.

UPS_CM_Set_context (section C.3 page 118) allows the user to change the process view of UPS. So, unlike MPI functions, the UPS process context is not an explicit argument to function calls.

Users are responsible for the formation of the communication context (MPI users might use MPI_COMM_SPLIT). The handle for that context is then passed on to UPS_CM_Set_context.

5.7 Conflicts With Other Packages

UPS has taken not to interfere with the user's ability to use any product outside of UPS. For example, UPS insulates itself from outside use of MPI and HDF.

If a conflict does arise, send email to ups-team@lanl.gov and we will resolve it.

5.8 Error Reporting

The way in which UPS returns internal errors can be modified by setting options and/or environment variables. For more information, see UPS_AA_Opt_set() (section C.2 page 98), the ER section of the options one can set UPS_AA_OPT_TYPE_enum (section B page 51), and constants dealing with file output UPS_ER_OUTPUT_enum (section B page 65).

- Output
  - Location

    By default, UPS error output is sent to stderr. This can be overridden by a function call or environment variable (see previous paragraph) to send error output to the file ups_err.txt or to not print any error output. The environment variable overrides the function call. One can take an exec with error printing turned off and turn it back on without having to recompile.
  - Buffer

    Users can turn error message printing off and, via a function call, still get the error message buffer. That way, one can decide what to do with any UPS error messages.
- Format
  - Grouping UPS Error Messages

    Error messages start with "UPS Error" and have the process id / error level. This way, output can be grep'ed to get UPS error messages and then sort'ed to get them organized by process number.
  - Location of Error

    The error message has __LINE__ and __FILE__ and cascade upwards to that a stack trace is produced.
  - Cause of Error

    The condition causing the error is stringized and placed in the error message. A text message is also added to the error message. The hope is that if one sees something like "message_size ¡ 0" and "invalid message_size", one can infer that the error has to do with the message_size argument passed to UPS.
  - Error Interlacing

    The error message is printed on one line and the output buffer is fflush'ed before and after the fprintf. This is done to reduce the risk of error messages being interlaced with one another.
- Return Value

  All UPS functions return an error code. So, users can turn off error message printing but still get an error code. To see a list of error codes, see UPS_AA_Error_enum (section B page 45).

5.9 UPS Version Consistency

It is important to maintain UPS version consistency at all stages of the build. The same version of UPS that is used to build user libraries should then be used to build user execs.

There are various ways to get the UPS version number (which is tied to UPS source via a CVS tag):

- UPS Installation Directory

  UPS will be installed in a directory with the name:

  ```
  v-##-##-##
  ```

  The version number will be the digits in that name.

- Constant `UPS_VERSION` in Header Files

  The UPS header file (`ups.h, upsf.h, upsf77.h`) have the constant `UPS_VERSION` defined as the version number.

- Function Call Return Value

  One can make the following call to get the version number:

  ```
  int version_ups;
  UPS_AA_Opt_get( UPS_AA_OPT_VERSION_CHECK, &version_ups );
  ```

  By calling the set function with the current `UPS_VERSION`, UPS tests the version consistency and returns an error if there is a mismatch. If users are concerned about UPS version consistency, they should make one of the following calls in their code:

  C:

  ```
  int ierr, version_ups=UPS_VERSION;
  ierr = UPS_AA_Opt_set( UPS_AA_OPT_VERSION_CHECK, &version_ups );
  assert( ierr == UPS_OK );
  ```

  Fortran:

  ```
  use UPS
  integer(KIND=UPS_KIND_INT4) :: ierr
  call UPSF_AA_OPT_SET( UPS_AA_OPT_VERSION_CHECK, UPS_VERSION, ierr )
  if( ierr .ne. UPS_OK ) stop
  ```

  F77:

```
include 'upsf77.h'
integer*4 ierr
call UPS_AA_OPT_SET( UPS_AA_OPT_VERSION_CHECK, UPS_VERSION, ierr )
if( ierr .ne. UPS_OK ) stop
```

- Library Function

  A no-op function is compiled in to `libups.a`. It takes no arguments and returns no values. Its name is of the form:

  ```
  ups_version_#####
  ```

  This allows users to `'nm libups.a | grep -i ups_version'` to get the version number of the library.

  Inside a C routine that includes `ups.h`, one can make the following call:

  ```
  UPS_VERSION_CHECK();
  ```

  An error will occur at link time if there is a version mismatch.

6 Packages

The following is a more in depth explanation of the different packages in UPS. See the reference pages (section C page 89) for routine descriptions.

6.1 General Package (AA)

The general component of UPS contains a small number of routines that are applicable to all other components. These include:

- **Initialization and Termination**

  `UPS_AA_Init` (section C.2 page 92) and `UPS_AA_Terminate` (section C.2 page 100) initialize and terminate UPS respectively.

  `UPS_AA_Init` initializes everything UPS functions need in order to operate (eg `UPS_AA_Init` will call `MPI_Init` if needed and not already done so by the user).

  `UPS_AA_Terminate` frees resources used by UPS. This includes buffer spaces and perhaps other products (eg `MPI_Finalize` will be called if `UPS_AA_Init` initialized MPI)

- **Statistics Information**

  Statistics about how UPS is used will be sent to an outpu file upon UPS termination. See `UPS_AA_Statistics` (section C.2 page 99) for more information.

The AA reference section (section C.2, page 91) describes each function.

6.2 Communication Package (CM)

Applications written for parallel computing environments typically require the exchange of data among the parallel processes. Users will recognize many similarities between the functions in the UPS package and the functions in MPI [2] and PVM [4].

The functions in the CM package can be split into the following categories.

- **Collective**

  All processes participate in these operations. These include barriers, broadcasts, reductions, ...

- **Process Information**

  Information like communicator context, number of processes on the host, pe number with respect to the host, number of hosts, total number of processes, global process number, ... Setting the communicator context is also possible.

- **Shared Memory**

  Routines for allocating chunks of memory shared among processes.

The CM reference section (section C.3, page 102) describes each function.

6.2.1 Shared Memory Example

The following code is found in the examples directory: `shared_memory.c`

```
// This program is an example of using shared memory.
// Each set of processes from a shared memory area compute
// a local sum.  Then, the shared memory area masters compute
// a global sum and then send that global answer back to
// the local processes.
// - Get shared memory info (eg penum_sm = penum wrt sm)
// - Allocate shared memory area (sm_area)
// - All: sm_area[penum_sm] = penum_all
// - SM Masters: sm_area[numpes_sm]   = Sum( sm_area[1..numpes_sm] )
// - SM Masters: sm_area[numpes_sm+1] = Total sum (using MPI)
#include <stdio.h>
#include "ups.h"
#include "mpi.h"
#define GIGB ((double)1024.0*(double)1024.0*(double)1024.0)
#if defined( _UPS_USE_PROCMON )
#include "procmon_info.h"
#endif
int main( int argc, char **argv )
{
  // IDs
  int
```

```
    idnum,       // rank of the shared memory area the pe belongs to
    numids,      // number of shared memory areas
    numpes_all,  // number of pes total
    numpes_sm,   // number of pes in this pes shared memory area
    penum_all,   // pe number (rank) wrt the global context
    penum_sm;    // pe number (rank) wrt this pes shared memory area
  // Other
  int
    i,           // loop variable
    ierr=0,      // error return value
    sum_all,     // total sum of all shared memory areas
    sum_correct, // the correct answer
    sum_sm;      // sum of this pes shared memory area
  long long
    sm_size;     // the desired/actual size in bytes of the sm area
  volatile int
    *sm_area;    // shared memory area - volatile
                 // sm_area[penum_sm]    = penum_all
                 // sm_area[numpes_sm]   = sum_sm
                 // sm_area[numpes_sm+1] = sum_all
  UPS_CM_P_group_enum
    p_group;     // process group connected by shared memory
                 //   UPS_CM_P_GROUP_BOX  - usually
                 //   UPS_CM_P_GROUP_SELF - if no shared memory
  MPI_Comm
    context_mstr_sm, // master pes of each sm area (penum_sm == 0)
    context_sm;      // pes within this pe's sm area
  // process info
  double
    m_size,      // Gbytes of      machine memory (-1 if unknown)
    m_free;      // Gbytes of free machine memory (-1 if unknown)
#if defined( _UPS_USE_PROCMON )
  PROCMON_INFO_struct
    pinfo; // process info (especially machine memory)
#endif
  ierr += UPS_AA_Init( argc, argv );
  // get p_group connected by sm (usually UPS_CM_P_GROUP_BOX)
  ierr += UPS_CM_Sm_get_item( UPS_AA_MEM_ITEM_P_GROUP, NULL, &p_group );
  // get contexts: pes within shared memory area, sm masters
  ierr += UPS_CM_P_group_item( NULL, p_group,
                               UPS_CM_P_GROUP_ITEM_ALL_CNTXT,
                               &context_sm );
  ierr += UPS_CM_P_group_item( NULL, p_group,
                               UPS_CM_P_GROUP_ITEM_MSTR_CNTXT,
                               &context_mstr_sm );
  // rank and size with respect to current context
  ierr += UPS_CM_Get_penum( &penum_all );
  ierr += UPS_CM_Get_numpes( &numpes_all );
  // rank and size with respect to this pe's shared memory area
  ierr += UPS_CM_P_group_item( NULL, p_group,
                               UPS_CM_P_GROUP_ITEM_PENUM,
                               &penum_sm );
  ierr += UPS_CM_P_group_item( NULL, p_group,
                               UPS_CM_P_GROUP_ITEM_NUMPES,
                               &numpes_sm );
  // shared memory area id and number of shared memory areas
  ierr += UPS_CM_P_group_item( NULL, p_group,
                               UPS_CM_P_GROUP_ITEM_IDNUM,
                               &idnum );
  ierr += UPS_CM_P_group_item( NULL, p_group,
                               UPS_CM_P_GROUP_ITEM_NUMIDS,
                               &numids );
  // get machine memory size if using PROCMON
  m_size = -1.0;
  m_free = -1.0;
#if defined( _UPS_USE_PROCMON )
  procmon_info_init( &pinfo );
  while( procmon_info_get( 0, &pinfo ) == PROCMON_WARNING ){};
  m_size = pinfo.m_size/GIGB;
  m_free = pinfo.m_free/GIGB;
#endif
  // PE's within shared memory area do work
  // get shared memory area (enough for 1 int per pe)
  // master pe's 0 must know requested size
  // all pes connected to a sm area must call UPS_CM_Sm_malloc
  sm_size = (long)((numpes_sm+2)*sizeof( int ) );
  ierr += UPS_CM_Sm_malloc( &sm_size, (volatile void **)(&sm_area) );
  // modify sm area and wait for p_group processes to finish
  sm_area[penum_sm] = penum_all;
  ierr += UPS_CM_Set_context( &context_sm );
  ierr += UPS_CM_Barrier( );
  // master pes work to get sum_all
  if ( penum_sm == 0 )
    {
      ierr += UPS_CM_Set_context( &context_mstr_sm );
      // master pe reduce sum to sum_sm
      sum_sm = 0;
      for( i = 0; i < numpes_sm; i++ )
        sum_sm += sm_area[i];
      sm_area[numpes_sm] = sum_sm;
      // master pes reduce sum_sm to sum_total and place in sm area
      ierr += UPS_CM_Allreduce(&sum_sm,&sum_all,1,UPS_DT_INT,UPS_AA_SUM);
      sm_area[numpes_sm+1] = sum_all;
      ierr += UPS_CM_Set_context( &context_sm );
    }
  // sm area waits for answer
  ierr += UPS_CM_Barrier( );
  // print out answer
  sum_correct = (numpes_all*(numpes_all-1))/2;
  if( penum_all == 0 )
    {
      printf( "%6d: %s\n", numpes_all, "Numpes" );
      printf( "%6d: %s\n", numids, "Number of Shared Memory Areas" );
      printf( "(%5s/%5s) (%5s/%5s) (%8s/%5s) (%9s/%9s): %10s %10s %10s\n",
              "penum", "total", "idnum", "total", "penum_sm", "total",
              "m_free:GB", "m_size:GB",
              "sm_sum", "total_sum", "correct" );
    }
  printf( "(%5d/%5d) (%5d/%5d) (%8d/%5d) (%9.2e/%9.2e): %10d %10d %10d\n",
          penum_all, numpes_all, idnum, numids, penum_sm, numpes_sm,
          m_free, m_size,
          sm_area[numpes_sm], sm_area[numpes_sm+1], sum_correct );
  // free sm_area
  ierr += UPS_CM_Sm_free( (volatile void **)(&sm_area) );
  // check for any errors
  if ( ierr != UPS_OK )
    UPS_AA_Abort();
  UPS_AA_Terminate( );
  return( ierr );
}
```

6.3 Data Parallel Package (DP)

Data parallel functions provide each process a global view of vectors which are distributed across multiple independent processes. Hence data parallel calls may be viewed as serial processes.

Some of the operations that can be done on vectors are:

- **Vector Reductions**

  Like max, min, sum, ...

- **Dot Products, Vector Norms**

- **Miscellaneous**

  Like sorting and global numbering.

The DP reference section (section C.4, page 126) describes each function.

6.4 Datatype Package (DT)

Functions that deal with datatype information are kept here. An example is `UPS_DT_Sizeof` (which returns the size of a UPS datatype).

To see a list of valid datatypes, see section B page 63.

- **I Don't see my datatype???**

  Many times all that is important is to match up the size of the datatype. For example, when using the IO package to write an array of C++ booleans, choose the UPS_DT_CHAR datatype (one can verify sizes via `sizeof()` and `UPS_DT_Sizeof()` functions). Likewise one can use UPS_DT_INT4 when dealing with Fortran logicals.

The DT reference section (section C.5, page 139) describes each function.

6.5 Error Package (ER)

This package contains routines that allow users to better deal with error conditions. For example, users may set an alarm to go off (and trigger the program to terminate) after a certain time has passed unless the alarm is turned off before then.

For information about how UPS deals with internal errors, see Error Reporting (section 5.8, page 19)

The ER reference section (section C.6, page 140) describes each function.

6.6 Gather Scatter Package (GS)

Gather/Scatter functions (while similar to Data Parallel functions - see section 6.3, page 27), are so widely used to warrant their own package area. These routines deal with the accessing (gather) and storing (scatter) of data distributed across processes. The GS packages has routines in the following categories:

- **Gather Scatter Specific**

  gather/scatter calls have an init routine (`UPS_GS_Setup` - see section C.7 page 165). This routine sets up communication patterns and buffers for future calls to gather/scatter.

- **Collate and Distribute**

  Similar to gather/scatter, these functions operate on a dataset where the io_pe sends and receives data from all other processes.

The following is an example of how users might apply the gather/scatter component.

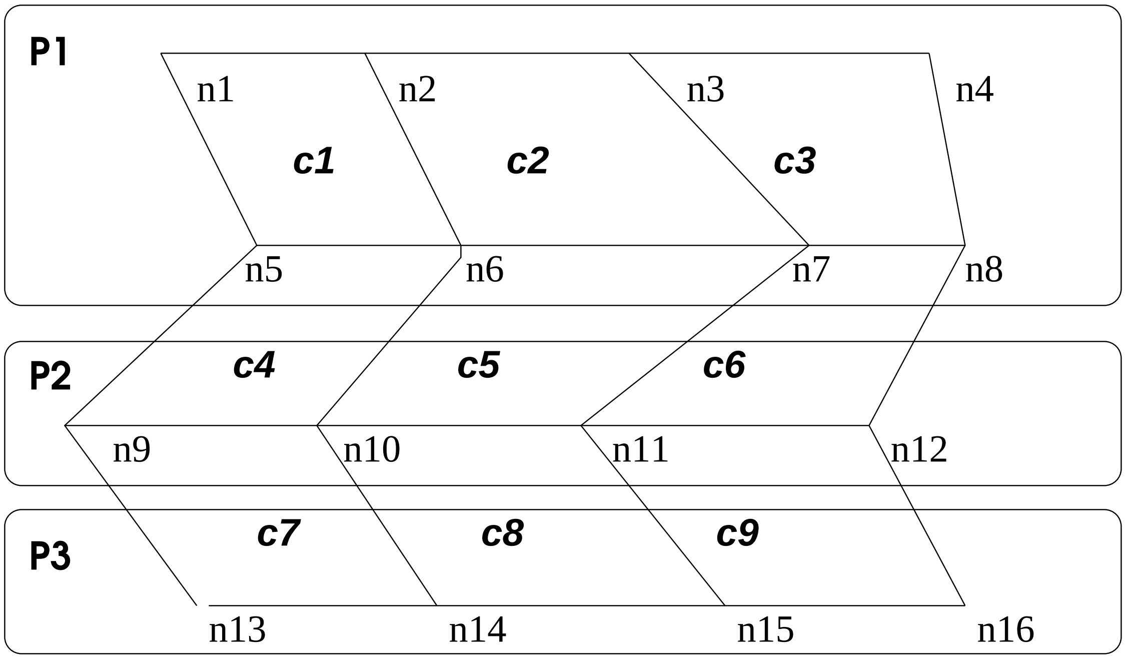

Figure 6: Gather/scatter view of a domain and its decomposition into 3 processes.

This figure illustrates a domain decomposition and cell connectivity scheme common to applications that could benefit from using the UPS gather/scatter component. The domain, consisting of nine cells with sixteen node vertices, has been divided into three partitions.

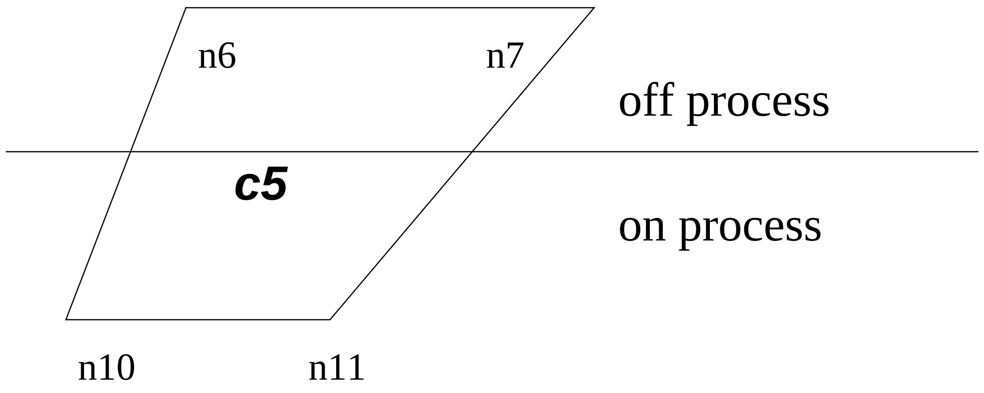

Figure 7: Gather/scatter view of a single cell

This figure, with its closeup view of cell `c5`*, illustrates the communication requirements of the gather/scatter model.*

The domain consists of nine cells with sixteen associated vertices, or nodes. The nodes are decomposed into three partitions as follows:

Partition 1	owns nodes `n1-n8`	and cell centers `c1-c3`
Partition 2	owns nodes `n9-n12`	and cell centers `c4-c6`
Partition 3	owns nodes `n13-n16`	and cell centers `c7-c9`

So, if process **P2** wishes to get the values for all the cell centers (both on process and off process) it can make a single gather call. Later on, the communication reverses and processes scatter information back to the cell centers.

The GS reference section (section C.7, page 146) describes each function.

6.7 File IO Package (IO)

UPS offers IO functionality based on Hierarchical Data Format
(HDF http://hdf.ncsa.uiuc.edu/HDF5). A file written with HDF has a structure similar to a Unix directory structure (eg HDF datasets/groups are similar to Unix files/directories respectively). UPS provides a simpler (but less robust) interface to file IO.

The basic file objects are:

- groups - like Unix directories
- datasets - like Unix files (arrays)

  UPS has simplified the calls one would have to make to HDF in order to write datasets distributed across processes.

  Please see UPS_IO_Dataset_read (section C.8 page 183) for example/discussion.
- attributes - simple datasets attached to groups or datasets

  These can be used to describe the object to which they are attached.

See the following for examples on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.
- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.
- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

Some other topics:

- Self Description

  Using the file objects mentioned above, one can use the IO package functions to create self-descriptive/structured data files. This is the real strength of using HDF.

  One can examine HDF files files with the command line tools h5ls and h5dump which are installed in the UPS bin directory.
- Sequential File Access

  UPS can provide sequential-like file access. Essentially we choose the names of datasets so you do not have to. When finished writing a dataset, a name is created with an internal count as part of the name. This count is then incremented in preparation of the next write. The same is done for reading.

Note, we encourage users to instead provide their own names/structure to the file so that the file is more self descriptive.

Please see UPS_IO_Dataset_read (section C.8 page 183) for example/discussion of writing/reading sequentially.

- UDM

  In the near(?) future, the UPS IO package will layer on top of other packages (UDM - Unified Data Model) which will give users access to greater functionality.

- Libraries

  Note: The user must now link in libhdf5.a (which is located in the UPS lib directory).

- Metadata

  Some metadata is also written to the file. This metadata is in the form of additional groups, datasets, or attributes. However, the naming convention of this metadata is standardized so that users can automate detection of certain object names.

  The format for the names of this metadata is as follows:

  – `UPSP_IO_`<rest of name> - UPS metadata
  – `UDM, udm` - UDM metadata (if using UDM protocol)

  It is important that the user does not create/modify file objects with names within the above naming conventions or else you could make UPS confused.

- Character Strings as input and output

  Null terminators are used by C users to signify the "end" of the string.

  Now, the tricky null-termination question:

  "When is there an additional null-terminator at the end of the string and when is there not one ?"

  The question is further complicated by Fortran users. UPS is designed so the the difference between the functionality of the C interface and the Fortran interface is minimal. In fact, in all but a few cases, the only difference between them is that the error return value for Fortran in an argument whereas in C it is a function return value.

  So, I came up with the following rules:

  – Return of "String Length"
    Any time the length of a string is requested or set by the user (ie length of object name or length of a string value) that length is without any additional null terminator and will be the number of chars that will be copied into/from a buffer.

- Strings as Output

  The user can always obtain the length of data to be obtained before obtaining the data (ie get the length of an object name, make sure sufficient space is allocated, then get the name itself). So, upon output, no artificial null-terminator will be added. During the "allocation" phase, the user can allocate length+1 to create their own null-terminated string.

- Strings as Input

  There are 2 different cases here:

  1. A "length" of the string has been supplied
     An example would be a writing a string dataset. The user has specified the size of the dataset and thus the size of the string. Exactly that length will be used.
  2. A "length" of the string has not been supplied
     An example would be the name of a group for group open. Such strings must be null-terminated in order for UPS to know the length. Fortran users can send in a string concatenated with the null terminator:

     ```
     'my_group'//ACHAR(0)
     ```

The IO reference section (section C.8, page 175) describes each function and has exquisite examples.

6.8 Utility Package (UT)

This component contains some utility functions that some may find useful. These routines do not necessarily deal with parallel programming.

The UT reference section (section C.9, page 230) describes each function.

References

[1] R. Barrett and M. McKay Jr. *UPS Developer's Guide*, 1999.
See `http://ccn-8.lanl.gov (Projects to Tools to UPS)`.

[2] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8, 1994.

[3] Ian Foster. *Designing and Building Parallel Programs.* Addison Wesley; also available online at `www.mcs.anl.gov/dbpp/`, 1996.

[4] G. A. Geist, A. L. Beguelin, J. J. Dongarra, R. J. Manchek, and V. S. Sunderam. PVM 3.0 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee, 1993.

[5] G. A. Geist, A. L. Beguelin, J. J. Dongarra, R. J. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine; A Users' Guide and Tutorial for Networked Parallel Computing.* MIT, 1994.

[6] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI.* MIT Press, 1996.

A Acknowledgements

The UPS effort relies on far more than the work of just the core team.

Ken Koch, an ASCI project leader at LANL and the godfather of UPS, continues to provide invaluable advice and guidance.

Dave Shirley of Abba Technologies contracted with Sandia National Laboratories, supplies advice regarding shared memory usage on Blue Mountain as well as a library for performing many of the basic communication functions in shared memory. We are expecting further assistance and guidance from him as UPS is ported to other platforms, specifically ASCI Red and ASCI Blue Pacific.

John Thorp of LANL participated in the initial design of UPS, a design that has held up well as new functionality has been added.

Hal Meyer of Cray/SGI Albuquerque has provided advice regarding I/O, and we intend to incorporate his I/O library into a subsequent version of UPS.

Sunlung Suen of LANL is involved in our crossbox communication algorithms on Blue Mountain, and we intend to include his library in a subsequent version of UPS.

Robert Ferrell of Cambridge Power Computing Associates has provided advice regarding the UPS gather/scatter capabilities.

Joe McGrath of SGI has patiently worked with UPS, and has tirelessly provided feedback and bug reports.

Bob Robey of LANL has worked closely with us in the development of the gather/scatter component.

And finally, our management, specifically Don Shirk and Pat Soran, have provided us the resources and encouragement necessary for this undertaking.

B UPS Constants

UPS makes use of predefined integer variables in various circumstances. The actual values of the variables will change from release to release, so it is advisable that users use the name of the variable instead of the value.

The values are defined in the include files ups.h (C), UPS.mod (fortran-90), and upsf77.h (fortran77),

Below is a listing of upsf77.h:

```
! ------------------------------------------------
!     This file is automatically generated from:
!        master_ups.h
!     by:
!        ./sync_f_include.pl
!
!     If you wish to make permanent changes, edit
!     that perl script.
!
! ------------------------------------------------
!
! --------------------------------
! LaTeX readable table of contents
! --------------------------------
!
!

! page 39 :  UPS_AA_Code_location_enum
! page 43 :  UPS_AA_ENVIRONMENT_VARIABLES_enum
! page 45 :  UPS_AA_Error_enum
! page 49 :  UPS_AA_ID_START_enum
! page 49 :  UPS_AA_MAIN_LANG_enum
! page 49 :  UPS_AA_Mem_item_enum
! page 51 :  UPS_AA_Mem_type_enum
! page 51 :  UPS_AA_OPT_TYPE_enum
! page 56 :  UPS_AA_Operation_enum
! page 57 :  UPS_AA_RUN_MODE_enum
! page 58 :  UPS_AA_Statistics_enum
! page 58 :  UPS_AA_Tags_enum
! page 59 :  UPS_AA_Version_enum
! page 59 :  UPS_CM_More_stuff_enum
! page 60 :  UPS_CM_P_group_enum
! page 61 :  UPS_CM_P_group_item_enum
! page 63 :  UPS_CM_Protocol_context_compare_enum
! page 63 :  UPS_DT_Boolean_Constants_enum
! page 63 :  UPS_DT_Datatype_enum
! page 65 :  UPS_ER_OUTPUT_enum
! page 65 :  UPS_GS_Index_type_enum
  page 65 :  UPS_GS_Item_enum
```

```
!
!
  page 69 :  UPS_GS_Setup_study_type_enum
!
  page 69 :  UPS_GS_Setup_type_enum
!
  page 70 :  UPS_GS_Special_index_enum
!
  page 70 :  UPS_IO_ACCESS_PES_enum
!
  page 72 :  UPS_IO_FILE_OBJECT_TYPE_enum
!
  page 73 :  UPS_IO_FILTER_TYPE_enum
!
  page 74 :  UPS_IO_INFO_ITEM_enum
!
  page 79 :  UPS_IO_INFO_TYPE_enum
!
  page 79 :  UPS_IO_LOC_ITEM_enum
!
  page 82 :  UPS_IO_OPEN_METHOD_enum
!
  page 82 :  UPS_IO_PROTOCOL_enum
!
  page 83 :  UPS_MS_INFO_ITEM_enum
!
  page 85 :  UPS_MS_NODEID_SCHEME_enum
!
  page 85 :  UPS_UT_Alloc_enum
!
  page 85 :  UPS_UT_CHECKSUM_TYPE_enum
!
  page 86 :  UPS_UT_Convert_enum
!
  page 87 :  UPS_UT_Loc_structure_op_enum
!
  page 87 :  UPS_UT_Loc_type_enum
!
  page 87 :  UPS_UT_Name_or_value_enum
!


!
!------------------------------------------------------------
!     Block from c enum: [UPS_AA_Code_location_enum]
!------------------------------------------------------------
!       ---------------------------------------------------------------
!       Code location enumerations:
!          UPS_<PACKAGE>_LOC_<ROUTINE_NAME>:   C interface
!          UPS_<PACKAGE>_LOCF_<ROUTINE_NAME>:  F77 interface
!          UPS_<PACKAGE>_LOCI_<ROUTINE_NAME>:  Internal routine
!          UPS_<PACKAGE>_LOCP_<ROUTINE_NAME>:  Private routine
!          UPS_<PACKAGE>_LOCPI_<ROUTINE_NAME>: Private Internal routine
!
!       NOTE: Fortran interface calls
!             Fortran-77 interface which calls
!             C interface which calls
!             Internal routine (if it exists)
!
!       In general, our Fortran and Fortran-77 are flat interfaces
!       (do nothing but call the next interface).  However, some
!       routines (eg fortran77 UPS_CM_ALLREDUCE) must do substantial
!       work and are therefore listed below (eg UPS_CM_LOCF_ALLREDUCE)
!       ---------------------------------------------------------------
!
!
```

```
!       -----------------------------------------------------------
!       Other slots for tool/user - will not be used by UPS library
!       -----------------------------------------------------------
!       must start with 0 so NUM_TIMINGS ok
      parameter ( UPS_AA_LOC_OTHER_0              = 0 )
      parameter ( UPS_AA_LOC_OTHER_1              = 1 )
      parameter ( UPS_AA_LOC_OTHER_2              = 2 )
      parameter ( UPS_AA_LOC_OTHER_3              = 3 )
      parameter ( UPS_AA_LOC_OTHER_4              = 4 )
      parameter ( UPS_AA_LOC_OTHER_5              = 5 )
      parameter ( UPS_AA_LOC_OTHER_6              = 6 )
      parameter ( UPS_AA_LOC_OTHER_7              = 7 )
      parameter ( UPS_AA_LOC_OTHER_8              = 8 )
      parameter ( UPS_AA_LOC_OTHER_9              = 9 )
!       --
!       AA
!       --
      parameter ( UPS_AA_LOC_ABORT                = 10 )
      parameter ( UPS_AA_LOC_INIT                 = 11 )
      parameter ( UPS_AA_LOC_IO_PE_GET            = 12 )
      parameter ( UPS_AA_LOC_IO_PE_SET            = 13 )
      parameter ( UPS_AA_LOC_OPT_GET              = 14 )
      parameter ( UPS_AA_LOC_OPT_SET              = 15 )
      parameter ( UPS_AA_LOC_STATISTICS           = 16 )
      parameter ( UPS_AA_LOC_TERMINATE            = 17 )
!       --
!       CM
!       --
      parameter ( UPS_CM_LOCF_ALLREDUCE           = 18 )
      parameter ( UPS_CM_LOCP_INIT                = 19 )
      parameter ( UPS_CM_LOCP_TERMINATE           = 20 )
!
      parameter ( UPS_CM_LOC_ALLGATHER            = 21 )
      parameter ( UPS_CM_LOC_ALLREDUCE            = 22 )
      parameter ( UPS_CM_LOC_BARRIER              = 23 )
      parameter ( UPS_CM_LOC_BARRIER_IDLE         = 24 )
      parameter ( UPS_CM_LOC_BCAST                = 25 )
      parameter ( UPS_CM_LOC_CONTEXT_FREE         = 26 )
      parameter ( UPS_CM_LOC_GET_CONTEXT          = 27 )
      parameter ( UPS_CM_LOC_GET_NUMPES           = 28 )
      parameter ( UPS_CM_LOC_GET_PENUM            = 29 )
      parameter ( UPS_CM_LOC_P_GROUP_ITEM         = 30 )
      parameter ( UPS_CM_LOC_REDUCE               = 31 )
      parameter ( UPS_CM_LOC_SALLTOALL            = 32 )
      parameter ( UPS_CM_LOC_SET_CONTEXT          = 33 )
      parameter ( UPS_CM_LOC_SM_FREE              = 34 )
      parameter ( UPS_CM_LOC_SM_GET_ITEM          = 35 )
      parameter ( UPS_CM_LOC_SM_MALLOC            = 36 )
      parameter ( UPS_CM_LOC_SM_SET_ITEM          = 37 )
!       --
!       DP
!       --
      parameter ( UPS_DP_LOCP_INIT                = 38 )
      parameter ( UPS_DP_LOCP_TERMINATE           = 39 )
!
      parameter ( UPS_DP_LOC_COMBINER             = 40 )
      parameter ( UPS_DP_LOC_COMBINERM            = 41 )
      parameter ( UPS_DP_LOC_COUNT_MASK           = 42 )
      parameter ( UPS_DP_LOC_DOT_PRODUCT          = 43 )
      parameter ( UPS_DP_LOC_DOT_PRODUCTM         = 44 )
      parameter ( UPS_DP_LOC_NUMBER_MASK          = 45 )
      parameter ( UPS_DP_LOC_SORT                 = 46 )
      parameter ( UPS_DP_LOC_VECTOR_NORM          = 47 )
      parameter ( UPS_DP_LOC_VECTOR_NORMM         = 48 )
!       --
!       DT
!       --
      parameter ( UPS_DT_LOCP_INIT                = 49 )
      parameter ( UPS_DT_LOCP_TERMINATE           = 50 )
!
      parameter ( UPS_DT_LOC_SIZEOF               = 51 )
!       --
!       ER
!       --
      parameter ( UPS_ER_LOCP_INIT                = 52 )
      parameter ( UPS_ER_LOCP_TERMINATE           = 53 )
!
      parameter ( UPS_ER_LOC_GET_WAIT_TIME        = 54 )
      parameter ( UPS_ER_LOC_PERROR               = 55 )
      parameter ( UPS_ER_LOC_SET_ALARM            = 56 )
      parameter ( UPS_ER_LOC_SET_WAIT_TIME        = 57 )
      parameter ( UPS_ER_LOC_UNSET_ALARM          = 58 )
!       --
!       GS
!       --
      parameter ( UPS_GS_LOCP_INIT                = 59 )
      parameter ( UPS_GS_LOCP_TERMINATE           = 60 )
!
      parameter ( UPS_GS_LOC_COLLATE              = 61 )
      parameter ( UPS_GS_LOC_DISTRIBUTE           = 62 )
      parameter ( UPS_GS_LOC_FREE                 = 63 )
      parameter ( UPS_GS_LOC_GATHER               = 64 )
      parameter ( UPS_GS_LOC_GATHER_LIST          = 65 )
      parameter ( UPS_GS_LOC_GATHER_MULTI         = 66 )
      parameter ( UPS_GS_LOCP_GATHER_SCATTER      = 67 )
      parameter ( UPS_GS_LOC_GET_ITEM             = 68 )
      parameter ( UPS_GS_LOC_SCATTER              = 69 )
      parameter ( UPS_GS_LOC_SCATTER_LIST         = 70 )
      parameter ( UPS_GS_LOC_SCATTER_MULTI        = 71 )
      parameter ( UPS_GS_LOCP_SETUP_GENERIC       = 72 )
      parameter ( UPS_GS_LOCP_SETUP_COMPRESSION   = 73 )
      parameter ( UPS_GS_LOC_SETUP                = 74 )
      parameter ( UPS_GS_LOC_SETUP_S_GLOBAL       = 75 )
      parameter ( UPS_GS_LOC_SETUP_S_LOCAL        = 76 )
      parameter ( UPS_GS_LOC_SETUP_STUDY          = 77 )
!       --
!       IO
!       --
!
      parameter ( UPS_IO_LOCP_INIT                = 78 )
      parameter ( UPS_IO_LOCP_TERMINATE           = 79 )
!
      parameter ( UPS_IO_LOC_ATTR_READ            = 80 )
      parameter ( UPS_IO_LOC_ATTR_WRITE           = 81 )
      parameter ( UPS_IO_LOC_ATTR_WRITE_S         = 82 )
      parameter ( UPS_IO_LOC_DATASET_READ         = 83 )
      parameter ( UPS_IO_LOC_DATASET_WRITE        = 84 )
      parameter ( UPS_IO_LOC_DS_R_S               = 85 )
      parameter ( UPS_IO_LOC_DS_W_S               = 86 )
      parameter ( UPS_IO_LOC_FILE_CLOSE           = 87 )
      parameter ( UPS_IO_LOC_FILE_OPEN            = 88 )
      parameter ( UPS_IO_LOC_FILE_TYPE            = 89 )
      parameter ( UPS_IO_LOC_FILTER_GET           = 90 )
      parameter ( UPS_IO_LOC_FILTER_SET           = 91 )
      parameter ( UPS_IO_LOC_GROUP_CLOSE          = 92 )
      parameter ( UPS_IO_LOC_GROUP_OPEN           = 93 )
      parameter ( UPS_IO_LOC_INFO_COUNT           = 94 )
      parameter ( UPS_IO_LOC_INFO_CREATE          = 95 )
      parameter ( UPS_IO_LOC_INFO_CREATE_SELF     = 96 )
      parameter ( UPS_IO_LOC_INFO_FREE            = 97 )
      parameter ( UPS_IO_LOC_INFO_ITEM_GET        = 98 )
      parameter ( UPS_IO_LOC_INFO_ITEM_SET        = 99 )
      parameter ( UPS_IO_LOC_LOC_ITEM_GET         = 100 )
      parameter ( UPS_IO_LOC_LOC_ITEM_SET         = 101 )
      parameter ( UPS_IO_LOC_RM                   = 102 )
!       --
!       UT
!       --
      parameter ( UPS_UT_LOCP_INIT                = 103 )
      parameter ( UPS_UT_LOCP_MEM_ALLOC           = 104 )
      parameter ( UPS_UT_LOCP_MEM_FREE            = 105 )
      parameter ( UPS_UT_LOCP_TERMINATE           = 106 )
!
      parameter ( UPS_UT_LOC_BINARY_OP            = 107 )
      parameter ( UPS_UT_LOC_BINARY_OPM           = 108 )
      parameter ( UPS_UT_LOC_CHECKSUM_GET         = 109 )
      parameter ( UPS_UT_LOC_CONVERT              = 110 )
      parameter ( UPS_UT_LOC_DT_CHANGE            = 111 )
      parameter ( UPS_UT_LOC_GET_NAME_OR_VALUE    = 112 )
      parameter ( UPS_UT_LOC_LOC_STRUCT_ALLOC     = 113 )
      parameter ( UPS_UT_LOC_LOC_STRUCTURE        = 114 )
      parameter ( UPS_UT_LOC_MEM_GET_ITEM         = 115 )
      parameter ( UPS_UT_LOC_REDUCE_OP            = 116 )
      parameter ( UPS_UT_LOC_REDUCE_OPM          = 117 )
      parameter ( UPS_UT_LOC_SLEEP               = 118 )
      parameter ( UPS_UT_LOC_SORT_COMPRESS       = 119 )
      parameter ( UPS_UT_LOC_SQUARE_ROOT         = 120 )
      parameter ( UPS_UT_LOC_TIME_WALL_GET       = 121 )
      parameter ( UPS_UT_LOC_TIME_WALL_INTERVAL  = 122 )
!       obsolete - will go away
      parameter ( UPS_CM_LOC_FREE_BUFFER         = 123 )
      parameter ( UPS_CM_LOC_GET_NUMHOSTS        = 124 )
      parameter ( UPS_CM_LOC_INITSEND            = 125 )
      parameter ( UPS_CM_LOC_MSGINFO             = 126 )
      parameter ( UPS_CM_LOC_PACK                = 127 )
      parameter ( UPS_CM_LOC_PBCAST              = 128 )
      parameter ( UPS_CM_LOC_PRECV               = 129 )
      parameter ( UPS_CM_LOC_PROBE               = 130 )
      parameter ( UPS_CM_LOC_PSEND               = 131 )
      parameter ( UPS_CM_LOC_RECV                = 132 )
      parameter ( UPS_CM_LOC_SEND                = 133 )
      parameter ( UPS_CM_LOC_UNPACK              = 134 )
!       -------------------
!       number of locations
!       -------------------
      parameter ( UPS_NUM_CODE_LOCATIONS         = 135 )
!
!


!
!------------------------------------------------------------
!     Block from c enum: [UPS_AA_ENVIRONMENT_VARIABLES_enum]
!------------------------------------------------------------
!       ----------------------------------------------------------------------
!       This is a partial list of environmental variables that can be set
!       to affect how UPS runs.  It is "partial" because other environmental
!       variables are also listed in other enums here.  Search on "environment"
!       To find them.
!       Remember to propogate environment variables to all processes.
!
!       Unless otherwise noted:
!          - Environment variables supersede function calls made by the user.
!            This allows one to change behavior without needing to recompile
!            code.
!          - These variables are read during UPS_AA_Init() and are not reread.
!            So, modifying them during a run has no affect.
!          - The enum value for the variable has no meaning.
!       ----------------------------------------------------------------------
!
!
!       ------------------
!       ER Alarm Variables
!       ------------------
!       value: time (secs) to wait when
!       *    UPS_ER_Set_alarm() has been called before
!       *    triggering a dump of type
!       *    UPS_ER_ALARM_DUMPCORE.
      parameter ( UPS_ER_MAX_WAIT_TIME           = 0 )
!       value: what flavor of core to dump when
!       *    UPS_ER_Set_alarm() has been called and
!       *    UPS_ER_MAX_WAIT_TIME time has passed.
!       * purpose: The following are the possible
!       *    signals sent to kill the processes:
!       *       0: SIGKILL (default)
!       *          core file will not be created
!       *       1: SIGABRT
!       *          core file will be created
      parameter ( UPS_ER_ALARM_DUMPCORE          = 1 )
!       -----------------------------------------
!       UT (and CM_sm) Memory Management Variables
!       -----------------------------------------
!       value: nothing or !0: memory management off
!       *        otherwise:     memory management on
!       * purpose:
!       *    Default is that memory management is on
!       *    when UPS built in debug mode, and off when
!       *    UPS build in optimized mode.
!       *    Memory management does things like memory
!       *    overwrite and memory lead checking for UPS
!       *    internal variables.
      parameter ( UPS_MEM_NO_MANAGEMENT          = 2 )
!       value: number of "blocks" of memory
!       *    allocated for guard bytes before and
!       *    after the requested space for UPS
!       *    internal variables.
!       * purpose: When using memory management,
!       *    additional memory is allocated before
!       *    and after the memory block which is
!       *    checked for memory overwrites.
!       *    By default, this value is 1 and the
!       *    size of a block is the size needed to
!       *    align memory.
      parameter ( UPS_MEM_NUM_GUARD_BLOCKS       = 3 )
!       ----------------
!       TMPDIR variables
!       ----------------
!       value: directory name to place temporary shared
!       *    memory files (if needed).
!       * purpose: When using UPS shared memory calls
!       *    (UPS_CM_Sm_<blah>), some architectures create
!       *    temporary files associated with the shared memory
!       *    area.  By default, these files are placed in
!       *    a directory prefered/required by the underlying
!       *    shared memory protocol (eg in /tmp or in the
!       *    current working directory).  You may change
!       *    this directory with this environment variable.
!       *    Normally, no temp files should remain after
!       *    execution is finished.  However, if errors
!       *    preclude cleanup, users might see ".nfs" files or
!       *    files beginning with "ups_cm_sm_" in the tmp
!       *    directory.
!       *    The order of preference for the location of
!       *    these files is:
!       *       1: UPS_TMPDIR_SM
!       *       2: UPS_TMPDIR
!       *       3: TMPDIR
      parameter ( UPS_TMPDIR_SM                  = 4 )
!       value: directory name to place temp files (if needed)
      parameter ( UPS_TMPDIR                     = 5 )
!
!



!
!-------------------------------------------------------------
!     Block from c enum: [UPS_AA_Error_enum]
!-------------------------------------------------------------
!       -----------
!       error codes
!       -----------
!
!
!       ---------------------------
!                    OK
!       ---------------------------
      parameter ( UPS_OK                         = 0 )
!       ---------------------------
!                    aa
!       ---------------------------
!
      parameter ( UPS_ERROR_AA_GENERAL           = -999 )
!       --------------
!       Generic Errors
!       --------------
      parameter ( UPS_ERROR_AA_BAD_PARAMETER     = -998 )
      parameter ( UPS_ERROR_AA_HANG              = -997 )
      parameter ( UPS_ERROR_AA_MEMORY_NOT_FREED  = -996 )
      parameter ( UPS_ERROR_AA_NO_MEMORY         = -995 )
      parameter ( UPS_ERROR_AA_NOT_IMPLEMENTED   = -994 )
      parameter ( UPS_ERROR_AA_INVALID_DATATYPE  = -993 )
      parameter ( UPS_ERROR_AA_INVALID_OPERATION = -992 )
      parameter ( UPS_ERROR_AA_INVALID_TAG       = -991 )
!       ------------------
!       aa: routine errors
!       ------------------
```

```
      parameter ( UPS_ERROR_AA_ABORT              = -990 )
      parameter ( UPS_ERROR_AA_INIT               = -989 )
      parameter ( UPS_ERROR_AA_IO_PE_GET          = -988 )
      parameter ( UPS_ERROR_AA_IO_PE_SET          = -987 )
      parameter ( UPS_ERROR_AA_OPT_GET            = -986 )
      parameter ( UPS_ERROR_AA_OPT_SET            = -985 )
      parameter ( UPS_ERROR_AA_STATISTICS         = -984 )
      parameter ( UPS_ERROR_AA_TERMINATE          = -983 )
!       ---------------------------
!                    cm
!       ---------------------------
      parameter ( UPS_ERROR_CM_GENERAL            = -1999 )
!       ---------------------
!       cm: Basic environment
!       ---------------------
      parameter ( UPS_ERROR_CM_ABORT              = -1998 )
      parameter ( UPS_ERROR_CM_CORE               = -1997 )
      parameter ( UPS_ERROR_CM_INIT               = -1996 )
      parameter ( UPS_ERROR_CM_TERMINATE          = -1995 )
!       ----------------------
!       cm: Process management
!       ----------------------
      parameter ( UPS_ERROR_CM_CONTEXT_FREE       = -1994 )
      parameter ( UPS_ERROR_CM_GET_CONTEXT        = -1993 )
      parameter ( UPS_ERROR_CM_GET_NUMPES         = -1992 )
      parameter ( UPS_ERROR_CM_GET_PENUM          = -1991 )
!       Used by upspi_cm_global_penum.
      parameter ( UPS_ERROR_CM_GLOBAL_PENUM       = -1990 )
      parameter ( UPS_ERROR_CM_MSGINFO_FILL       = -1989 )
      parameter ( UPS_ERROR_CM_P_GROUP_ITEM       = -1988 )
      parameter ( UPS_ERROR_CM_SET_CONTEXT        = -1987 )
!       --------------
!       cm: Collective
!       --------------
      parameter ( UPS_ERROR_CM_ALLGATHER          = -1986 )
      parameter ( UPS_ERROR_CM_ALLREDUCE          = -1985 )
      parameter ( UPS_ERROR_CM_BARRIER            = -1984 )
      parameter ( UPS_ERROR_CM_BARRIER_IDLE       = -1983 )
      parameter ( UPS_ERROR_CM_BCAST              = -1982 )
      parameter ( UPS_ERROR_CM_REDUCE             = -1981 )
      parameter ( UPS_ERROR_CM_SALLTOALL          = -1980 )
      parameter ( UPS_ERROR_CM_SM_FREE            = -1979 )
      parameter ( UPS_ERROR_CM_SM_GET_ITEM        = -1978 )
      parameter ( UPS_ERROR_CM_SM_MALLOC          = -1977 )
!
      parameter ( UPS_ERROR_CM_SET_HOST_INFO      = -1976 )
!       ---------------------------
!                    dp
!       ---------------------------
      parameter ( UPS_ERROR_DP_GENERAL            = -2999 )
      parameter ( UPS_ERROR_DP_INIT               = -2998 )
      parameter ( UPS_ERROR_DP_COMBINER           = -2997 )
      parameter ( UPS_ERROR_DP_COMBINERM          = -2996 )
      parameter ( UPS_ERROR_DP_COUNT_MASK         = -2995 )
      parameter ( UPS_ERROR_DP_DOT_PRODUCT        = -2994 )
      parameter ( UPS_ERROR_DP_DOT_PRODUCTM       = -2993 )
      parameter ( UPS_ERROR_DP_NUMBER_MASK        = -2992 )
      parameter ( UPS_ERROR_DP_SORT               = -2991 )
      parameter ( UPS_ERROR_DP_TERMINATE          = -2990 )
      parameter ( UPS_ERROR_DP_VECTOR_NORM        = -2989 )
      parameter ( UPS_ERROR_DP_VECTOR_NORMM       = -2988 )
!       ---------------------------
!                    dt
!       ---------------------------
      parameter ( UPS_ERROR_DT_GENERAL            = -3999 )
      parameter ( UPS_ERROR_DT_INIT               = -3998 )
      parameter ( UPS_ERROR_DT_SIZEOF             = -3997 )
      parameter ( UPS_ERROR_DT_TERMINATE          = -3996 )
!       ---------------------------
!                    er
!       ---------------------------
      parameter ( UPS_ERROR_ER_GENERAL            = -4999 )
      parameter ( UPS_ERROR_ER_GET_WAIT_TIME      = -4998 )
      parameter ( UPS_ERROR_ER_INIT               = -4997 )
      parameter ( UPS_ERROR_ER_PERROR             = -4996 )
      parameter ( UPS_ERROR_ER_SET_ALARM          = -4995 )
      parameter ( UPS_ERROR_ER_SET_WAIT_TIME      = -4994 )
      parameter ( UPS_ERROR_ER_TERMINATE          = -4993 )
      parameter ( UPS_ERROR_ER_UNSET_ALARM        = -4992 )
!       ---------------------------
!                    gs
!       ---------------------------
      parameter ( UPS_ERROR_GS_GENERAL            = -5999 )
      parameter ( UPS_ERROR_GS_SETUP              = -5998 )
      parameter ( UPS_ERROR_GS_INIT               = -5997 )
      parameter ( UPS_ERROR_GS_GATHER             = -5996 )
      parameter ( UPS_ERROR_GS_GATHER_LIST        = -5995 )
      parameter ( UPS_ERROR_GS_GATHER_MULTI       = -5994 )
      parameter ( UPS_ERROR_GS_UNKNOWN_ID         = -5993 )
      parameter ( UPS_ERROR_GS_SCATTER            = -5992 )
      parameter ( UPS_ERROR_GS_SCATTER_LIST       = -5991 )
      parameter ( UPS_ERROR_GS_SCATTER_MULTI      = -5990 )
      parameter ( UPS_ERROR_GS_COLLATE            = -5989 )
      parameter ( UPS_ERROR_GS_DISTRIBUTE         = -5988 )
      parameter ( UPS_ERROR_GS_FREE               = -5987 )
      parameter ( UPS_ERROR_GS_GET_ITEM           = -5986 )
      parameter ( UPS_ERROR_GS_SETUP_STUDY        = -5985 )
      parameter ( UPS_ERROR_GS_SETUP_S_GLOBAL     = -5984 )
      parameter ( UPS_ERROR_GS_SETUP_S_LOCAL      = -5983 )
      parameter ( UPS_ERROR_GS_TERMINATE          = -5982 )
!       ---------------------------
!                    io
!       --------------------------
      parameter ( UPS_ERROR_IO_GENERAL            = -6999 )
      parameter ( UPS_ERROR_IO_CORE               = -6998 )
      parameter ( UPS_ERROR_IO_ATTR_READ          = -6997 )
      parameter ( UPS_ERROR_IO_ATTR_WRITE         = -6996 )
      parameter ( UPS_ERROR_IO_ATTR_WRITE_S       = -6995 )
      parameter ( UPS_ERROR_IO_DATASET_READ       = -6994 )
      parameter ( UPS_ERROR_IO_DATASET_WRITE      = -6993 )
      parameter ( UPS_ERROR_IO_DS_R_S             = -6992 )
      parameter ( UPS_ERROR_IO_DS_W_S             = -6991 )
      parameter ( UPS_ERROR_IO_FILTER_GET         = -6990 )
      parameter ( UPS_ERROR_IO_FILTER_SET         = -6989 )
      parameter ( UPS_ERROR_IO_FILE_CLOSE         = -6988 )
      parameter ( UPS_ERROR_IO_FILE_OPEN          = -6987 )
      parameter ( UPS_ERROR_IO_FILE_TYPE          = -6986 )
      parameter ( UPS_ERROR_IO_GROUP_CLOSE        = -6985 )
      parameter ( UPS_ERROR_IO_GROUP_OPEN         = -6984 )
      parameter ( UPS_ERROR_IO_INFO_COUNT         = -6983 )
      parameter ( UPS_ERROR_IO_INFO_CREATE        = -6982 )
      parameter ( UPS_ERROR_IO_INFO_CREATE_SELF   = -6981 )
      parameter ( UPS_ERROR_IO_INFO_FREE          = -6980 )
      parameter ( UPS_ERROR_IO_INFO_ITEM_GET      = -6979 )
      parameter ( UPS_ERROR_IO_INFO_ITEM_SET      = -6978 )
      parameter ( UPS_ERROR_IO_INIT               = -6977 )
      parameter ( UPS_ERROR_IO_LOC_ITEM_GET       = -6976 )
      parameter ( UPS_ERROR_IO_LOC_ITEM_SET       = -6975 )
      parameter ( UPS_ERROR_IO_RM                 = -6974 )
      parameter ( UPS_ERROR_IO_TERMINATE          = -6973 )
!       --------------------------
!                    ut
!       --------------------------
      parameter ( UPS_ERROR_UT_GENERAL            = -8999 )
      parameter ( UPS_ERROR_UT_INIT               = -8998 )
      parameter ( UPS_ERROR_UT_CHECKSUM_GET       = -8997 )
      parameter ( UPS_ERROR_UT_CONVERT            = -8996 )
      parameter ( UPS_ERROR_UT_BINARY_OP          = -8995 )
      parameter ( UPS_ERROR_UT_BINARY_OPM         = -8994 )
      parameter ( UPS_ERROR_UT_DT_CHANGE          = -8993 )
      parameter ( UPS_ERROR_UT_GET_NAME_OR_VALUE  = -8992 )
      parameter ( UPS_ERROR_UT_LOC_STRUCT_ALLOC   = -8991 )
      parameter ( UPS_ERROR_UT_LOC_STRUCTURE      = -8990 )
      parameter ( UPS_ERROR_UT_MEM_ALLOC          = -8989 )
      parameter ( UPS_ERROR_UT_MEM_GET_ITEM       = -8988 )
      parameter ( UPS_ERROR_UT_MEM_FREE           = -8987 )
      parameter ( UPS_ERROR_UT_REDUCE_OP          = -8986 )
      parameter ( UPS_ERROR_UT_REDUCE_OPM         = -8985 )
      parameter ( UPS_ERROR_UT_SORT_COMPRESS      = -8984 )
      parameter ( UPS_ERROR_UT_SLEEP              = -8983 )
      parameter ( UPS_ERROR_UT_SQUARE_ROOT        = -8982 )
      parameter ( UPS_ERROR_UT_TERMINATE          = -8981 )
      parameter ( UPS_ERROR_UT_TIME_WALL_GET      = -8980 )
      parameter ( UPS_ERROR_UT_TIME_WALL_INTERVAL = -8979 )
!
!


!
!-------------------------------------------------------------
!     Block from c enum: [UPS_AA_ID_START_enum]
!-------------------------------------------------------------
!       ------------------------------------------------------------------
!       Where ids start for different things.
!       Start at different locations in hopes of helping users detect errors.
!       ------------------------------------------------------------------
!
!
!       0 = special id, use 1 (user creates indexed array)
      parameter ( UPS_GS_INFO_ID_START            = 1 )
      parameter ( UPS_IO_LOC_ID_START             = 100000 )
      parameter ( UPS_IO_INFO_ID_START            = 200000 )
      parameter ( UPS_MS_INFO_ID_START            = 300000 )
!
!


!
!-------------------------------------------------------------
!     Block from c enum: [UPS_AA_MAIN_LANG_enum]
!-------------------------------------------------------------
!       The language of the main program
!
!
!       Use the default inferred from the
!       * interface called (eg using the C interface
!       * initialization routine UPS_AA_Init implies
!       * having a C main program).
      parameter ( UPS_AA_MAIN_LANG_UNSET          = 0 )
!       Main program is C
      parameter ( UPS_AA_MAIN_LANG_C              = 1 )
!       Main program is Fortran
      parameter ( UPS_AA_MAIN_LANG_F              = 2 )
!
!


!
!-------------------------------------------------------------
!     Block from c enum: [UPS_AA_Mem_item_enum]
!-------------------------------------------------------------
!       ------------------------------------------------------------------
!       Define elements of info about the UPS memory routines.
!       The datatype/value of the "address" argument will be the
!       the address used in the UPS memory routines.
!       The datatype of the "item" argument is given in the item description.
!       The get/set routines of UPS_CM_Sm_get_item, UPS_CM_Sm_set_item, and
!       UPS_UT_Mem_get_item can be used with some of these arguments.
!       The UT routines deal with normal memory and the Sm routines deal with
!       shared memory.
!       --------------------------------------------------------------------
!
!
!       Purpose: check memory (eg overwrite guard bytes)
!       * Datatype: item not used
!       * Use: UT get Sm get
!       * If the address supplied is NULL, all memory
!       * areas allocated by UPS are checked.
      parameter ( UPS_AA_MEM_ITEM_CHECK           = 0 )
!       Purpose: The number of memory info structs in use.
!       * Datatype: UPS_DT_INT8
!       * Use: UT get Sm get
      parameter ( UPS_AA_MEM_ITEM_NUM             = 1 )
!       Purpose: The number of bytes used
!       *          per guard byte area.
!       * Datatype: UPS_DT_INT8
!       * Use: UT get Sm get
!       * There will be a guard byte area
!       * before and a guard byte area after
!       * memory blocks.
      parameter ( UPS_AA_MEM_ITEM_NUM_GUARD_BYTES = 2 )
!       Purpose: The process group connected to a
!       * sm area.
!       * Datatype: int (UPS_CM_P_group_enum)
!       * Use: Sm get/set.
!       * If the address given is NULL, this will
!       * be the default p_group.
!       * If shared memory is allowed, the value will
!       * usually be UPS_CM_P_GROUP_BOX (it might be
!       * larger like UPS_CM_P_GROUP_ALL).
!       * If shared memory is not allowed, the value will
!       * be UPS_CM_P_GROUP_SELF.
!       * If the address is given, you can get (not set)
!       * the p_group for the sm area in question.
!       * This can be set as an environment variable.
      parameter ( UPS_AA_MEM_ITEM_P_GROUP         = 3 )
!       Purpose: The the size of a page.
!       * Datatype: UPS_DT_INT8
!       * Use: UT get Sm get
      parameter ( UPS_AA_MEM_ITEM_PAGESIZE        = 4 )
!       Purpose: The size in bytes of the memory area
!       * Datatype: UPS_DT_INT8
!       * Use: UT get Sm get
!       * This size does not include guard bytes
      parameter ( UPS_AA_MEM_ITEM_SIZE            = 5 )
```

```
!       Purpose: The total size in bytes of the memory
!       *          areas.
!       * Datatype: UPS_DT_INT8
!       * Use: UT get Sm get
!       * For UPS_CM_Sm_get_item, only master pe's
!       * will have non-0 values.  If no shared memory
!       * is actually used (there is no ability for
!       * shared memory or the proup is
!       * UPS_CM_P_GROUP_SELF) this value will be 0
!       * for all processes.
      parameter ( UPS_AA_MEM_ITEM_TOTAL           = 6 )
!
!



!
!------------------------------------------------------------
!     Block from c enum: [UPS_AA_Mem_type_enum]
!------------------------------------------------------------
!       ------------------
!       The type of memory
!       ------------------
!
!
!       -----
!       types
!       -----
!       normal memory - UPSP_UT_MEM_MALLOC
      parameter ( UPS_AA_MEM_NORMAL               = 0 )
!       shared memory - UPS_CM_Sm_malloc
      parameter ( UPS_AA_MEM_SHARED               = 1 )
!       ---
!       all
!       ---
!       all types of memory (above)
      parameter ( UPS_AA_MEM_ALL                  = 2 )
!
!



!
!------------------------------------------------------------
!     Block from c enum: [UPS_AA_OPT_TYPE_enum]
!------------------------------------------------------------
!       ------------------------------------------------------------
!       option parameters (optimizations, settings, ...whatnot...)
!       Get/Set with UPS_AA_Opt_get/UPS_AA_Opt_set
!       opt_start/opt_stop are used internally for determining package
!       ------------------------------------------------------------
!
!
```

```
!       internal use only
      parameter ( UPS_AA_OPT_START                = 1000 )
!       Purpose: Set error message output file.
!       * Datatype: int4(UPS_ER_OUTPUT_enum)
!       * Get: yes
!       * Set: yes
!       * By default, UPS error messages are sent
!       * to stderr.  One can change this by setting
!       * the type to something else.
!       * Calling UPS_AA_Opt_set with this option
!       * must be called by all pes with the same
!       * value.
!       * See UPS_ER_OUTPUT_enum for values.
!
      parameter ( UPS_AA_OPT_ERROUTPUT_TYPE       = 1001 )
!       Purpose: set type of run
!       * Datatype: int4(UPS_AA_RUN_MODE_enum)
!       * Get: yes
!       * Set: yes
!       * This is used for setting UPS to run
!       * in sequential or parallel mode.
!       * See UPS_AA_RUN_MODE_enum for more information.
!       * This can only be set before calling
!       * UPS_AA_Init() as it dictates what happens
!       * during UPS initialization.
      parameter ( UPS_AA_OPT_RUN_MODE             = 1002 )
!       Purpose: determine if UPS had been initialized
!       * Datatype: int4
!       * Get: yes
!       * Set: no
!       * If there is a chance other packages are using
!       * UPS, each package should check to see if UPS
!       * has already been initialized.  This way, they
!       * terminate UPS only if they initialized UPS.
      parameter ( UPS_AA_OPT_INITIALIZED          = 1003 )
!       Purpose: set language type main program
!       * Datatype: int4(UPS_AA_MAIN_LANG_enum)
!       * Get: yes
!       * Set: yes
!       * The language of the main program is normally
!       * inferred from interface UPS_AA_Init is called
!       * from (eg calling the C interface infers a C
!       * main).
!       * However, it is possible to initialize UPS with
!       * the UPS Fortran interface using a C main.
!       * You must set additional options (setting argv
!       * and argc) if you do this (see the options
!       * below).
      parameter ( UPS_AA_OPT_MAIN_LANG            = 1004 )
!       Purpose: set argc for C main + F interface
!       * Datatype: int4
!       * Get: yes
!       * Set: yes
!       * When calling the Fortran UPS interface to
!       * UPS_AA_INIT and using a C main, you must
!       * set argc.
!       * When getting this value, you will get
!       * whatever it was set to which is NOT
!       * necessarily the correct value.
      parameter ( UPS_AA_OPT_MAIN_LANG_ARGC       = 1005 )
!       Purpose: set argc for C main + F interface
!       * Datatype: array of char strings (char**)
!       * Get: yes
!       * Set: yes
!       * When calling the Fortran UPS interface to
!       * UPS_AA_INIT and using a C main, you must
!       * set argv.
!       * When getting this value, you will get
!       * whatever it was set to which is NOT
!       * necessarily the correct value.
      parameter ( UPS_AA_OPT_MAIN_LANG_ARGV       = 1006 )
!       Purpose: To check for UPS version mismatches
!       * Datatype: int4
!       * Get: yes
!       * Set: yes
!       * When getting the value, it will return an
!       * integer value UPS version number.
!       * When setting the value, UPS will print an
!       * error message and return an error value if
!       * there is a version mismatch.
      parameter ( UPS_AA_OPT_VERSION_CHECK        = 1007 )
!       internal use only
      parameter ( UPS_AA_OPT_STOP                 = 1008 )
!       internal use only
      parameter ( UPS_CM_OPT_START                = 2000 )
!       Purpose: determine if underlying protocol
!       * for this package has been initialized.
!       * Datatype: int4
!       * Get: yes
!       * Set: no
!       * Note, this returns if the underlying
!       * protocol for the particular
!       * UPS_AA_OPT_RUN_MODE has been initialized
!       * at all (not just if UPS has
!       * initialized it).
      parameter ( UPS_CM_OPT_PROTOCOL_INIT        = 2001 )
!       internal use only
      parameter ( UPS_CM_OPT_STOP                 = 2002 )
!       internal use only
      parameter ( UPS_DP_OPT_START                = 5000 )
!       Purpose: set the number of samples for sorting
!       * Datatype: int8
!       * Get: yes
!       * Set: yes
!       * You may also set this with an environment
!       * variable.
      parameter ( UPS_DP_OPT_NUM_SAMPLES          = 5001 )
!       internal use only
      parameter ( UPS_DP_OPT_STOP                 = 5002 )
!       internal use only
      parameter ( UPS_ER_OPT_START                = 3000 )
!       Purpose: Get the current UPS error message buffer
!       * Datatype: char array
!       * Get: yes
!       * Set: no
!       * The number of characters is obtained by
!       * UPS_ER_OPT_ERMESS_LEN and does not contain any
!       * null termination character.
      parameter ( UPS_ER_OPT_ERRMESS              = 3001 )
!       Purpose: number of chars in UPS_ERR_OPT_ERMESS
!       * Datatype: int8
!       * Get: yes
!       * Set: yes
!       * This length is without a null terminator.
!       * Setting this value to 0 will empty the error
!       * message buffer.
      parameter ( UPS_ER_OPT_ERRMESS_LEN          = 3002 )
!       internal use only
      parameter ( UPS_ER_OPT_STOP                 = 3003 )
!       internal use only
      parameter ( UPS_GS_OPT_START                = 4000 )
!       Purpose: sets whether or not to try compression
!       * Datatype: int4 (UPS_DT_TRUE=1/UPS_DT_FALSE=0)
!       * Get: yes
!       * Set: yes
!       * If true, will try to use compression.
!       * (a duplicated index is sent once and coppied
!       * on the receiving side)
!       * All processes must have the same value for this.
!       * You may also set this as an enviroment
!       * variable.
      parameter ( UPS_GS_OPT_COMPRESSION          = 4001 )
!       Purpose: num of indices for compression scan
!       * Datatype: int8
!       * Get: yes
!       * Set: yes
!       * This sets the number of indices to look for
!       * when doing compression.  The different
!       * values are:
!       *   <0: scan whole array (costly)
!       *    0: do not do compression
!       *   >0: look ahead this number of indices
!       * All processes must have the same value for this.
!       * You may also set this as an enviroment
!       * variable.
      parameter ( UPS_GS_OPT_COMP_LOOKAHEAD       = 4002 )
!       Purpose: sets minimum compression ratio
!       * Datatype: real8
!       * Get: yes
!       * Set: yes
!       * Ratio = 100*Comp_Size/UnComp_Size (rounded to int)
!       * Values will then ranbe from 0 - 100.
!       * The lower the value, the more compressed.
!       * The ratio for a particular GS setup must be less
!       * than this value in order to use compression.
!       *   <0: Use compression for any ratio
!       *    0: do not do compression
!       *   >0: use this ratio value
!       * All processes must have the same value for this.
!       * You may also set this as an enviroment
!       * variable.
      parameter ( UPS_GS_OPT_COMP_RATIO           = 4003 )
!       Purpose: num of indices for compression scan
!       * Datatype: int4 (UPS_DT_TRUE=1/UPS_DT_FALSE=0)
!       * Get: yes
!       * Set: yes
!       * If true, will try to use ordering of send/recvs
!       * (eg start off-box communication before on-box)
!       * All processes must have the same value for this.
!       * You may also set this as an enviroment
!       * variable.
      parameter ( UPS_GS_OPT_COMM_ORDERING        = 4004 )
!       internal use only
      parameter ( UPS_GS_OPT_STOP                 = 4005 )
!       internal use only
      parameter ( UPS_IO_OPT_START                = 0 )
!       Purpose: get/set the info for IO
!       * Datatype: protocol info (eg with mpi, MPI_Info)
!       * Get: yes
!       * Set: yes
!       * The user may pass in, for example, an MPI_Info
!       * variable for fine-tuning performance.
      parameter ( UPS_IO_OPT_INFO                 = 1 )
!       Purpose: underlying IO protocol
!       * Datatype: int4(UPS_IO_PROTOCOL_enum)
!       * Get: yes
!       * Set: yes
!       * Datafiles written with one protocol are
!       * incompatible with other protocols.
!       * Be careful when changing this value to ensure
!       * a consistent protocol for the file.
!       * You may also set this as an enviroment
!       * variable.
!       * See UPS_IO_PROTOCOL_enum below.
```

```
      parameter ( UPS_IO_OPT_PROTOCOL            = 2 )
!       Purpose: dataset name prefix for sequential
!       *         reads/writes
!       * Datatype: char array
!       * Get: yes
!       * Set: yes
!       * The dataset name consists of:
!       *   <prefix><0 based dataset number>
!       * and is used if no dataset name is given.
!       * The number of characters is obtained by
!       * UPS_IO_OPT_PREFIX_LENGTH.
      parameter ( UPS_IO_OPT_DS_PREFIX           = 3 )
!       Purpose: number of chars in
!       *         UPS_IO_OPT_DS_PREFIX
!       * Datatype: int8
!       * Get: yes
!       * Set: no
!       * The length of the prefix of the default
!       * dataset name.
!       * This length is without a null
!       * terminator.
      parameter ( UPS_IO_OPT_DS_PREFIX_LENGTH    = 4 )
!       Purpose: specifies pes that write to file
!       * Datatype: int(UPS_IO_ACCESS_PES_enum)
!       * Get: yes
!       * Set: yes
!       * You may also set this as an enviroment
!       * variable.
!       * See UPS_IO_ACCESS_PES_enum below.
      parameter ( UPS_IO_OPT_ACCESS_WRITE        = 5 )
!       Purpose: use loc id given by the user
!       * Datatype: int(UPS_DT_Boolean_Constants_enum)
!       * Get: yes
!       * Set: yes
!       * UPS_DT_FALSE(default):
!       *    let UPS set the value obtained from
!       *    IO file/group open calls.
!       * UPS_DT_TRUE:
!       *    UPS will use the value supplied to
!       *    IO file/group open calls.
!       *    This value must be non-negative.
!       * One must be careful not to mix ids set
!       * by the user and ones set by UPS.
      parameter ( UPS_IO_OPT_LOC_ID_USERS        = 6 )
!       internal use only
      parameter ( UPS_IO_OPT_STOP                = 7 )
!
!


!
```

```
!------------------------------------------------------------
!     Block from c enum: [UPS_AA_Operation_enum]
!------------------------------------------------------------
!       ------------------------------------------
!       define types of operations.
!       Can be used in packages like cm, gs, and dp
!       Start from 1000 so that they do not overlap
!       the datatypes.  Might find possible user
!       errors where they swap operation and
!       datatype arguments in a function call
!       ------------------------------------------
!
!
      parameter ( UPS_AA_BAND                     = 1000 )
      parameter ( UPS_AA_BOR                      = 1001 )
      parameter ( UPS_AA_LAND                     = 1002 )
      parameter ( UPS_AA_LOR                      = 1003 )
      parameter ( UPS_AA_MAX                      = 1004 )
      parameter ( UPS_AA_MAXLOC                   = 1005 )
      parameter ( UPS_AA_MIN                      = 1006 )
      parameter ( UPS_AA_MINLOC                   = 1007 )
      parameter ( UPS_AA_PROD                     = 1008 )
      parameter ( UPS_AA_DOT_PROD                 = 1009 )
      parameter ( UPS_AA_PUT                      = 1010 )
      parameter ( UPS_AA_SUM                      = 1011 )
      parameter ( UPS_AA_SUB                      = 1012 )
!       -----------------------------------------------
!       internal operations not intended for general use
!       -----------------------------------------------
!       internal op used for gs package
      parameter ( UPS_AA_GS_LIST                  = 1013 )
!
!



!
!------------------------------------------------------------
!     Block from c enum: [UPS_AA_RUN_MODE_enum]
!------------------------------------------------------------
!       ------------------------------------------------------------
!       Different types of run modes
!       Use UPS_AA_Opt_get/UPS_AA_Opt_set with UPS_AA_OPT_RUN_MODE.
!       These may only be set before calling UPS_AA_Init()
!       You may set the following as environmental variables as well:
!         (in mutually exclusive groups)
!            - setenv UPS_AA_RUN_MODE_PARALLEL
!              setenv UPS_AA_RUN_MODE_SERIAL
!       Remember that all environment variables must be propagated to
!       all the processes.
!       ------------------------------------------------------------
!
```

```
!
!       Default (must equal 0 to match init)
!       * This option causes the underlying
!       * communication protocol (eg MPI) to
!       * be called regardless of the number of
!       * processors being used.
      parameter ( UPS_AA_RUN_MODE_PARALLEL        = 0 )
!       This option causes serial code (single
!       * process) to be called instead of the
!       * underlying communication protocol (eg MPI).
!       * UPS has the same functionality when running
!       * serial as it does running parallel with
!       * one process.
!       * This option was added mainly for running
!       * with the HDF-EnSight reader which uses
!       * UPS IO routines but cannot run under mpirun.
!       * There are other tools that also have this
!       * restriction.
      parameter ( UPS_AA_RUN_MODE_SERIAL          = 1 )
!
!


!
!------------------------------------------------------------
!     Block from c enum: [UPS_AA_Statistics_enum]
!------------------------------------------------------------
!       ---------------------------------------------------------------
!       Statistics flags
!       Use UPS_AA_Statistics()
!       You may set the following as environmental variables as well:
!         (in mutually exclusive groups)
!            - setenv UPS_STATISTICS_OFF
!              setenv UPS_STATISTICS_ON
!       Remember that all environment variables must be propagated to
!       all the processes.
!       ---------------------------------------------------------------
!
!
!       (must equal 0 to match init)
!       * Disable statistics.
      parameter ( UPS_STATISTICS_OFF              = 0 )
!       Default
!       * Enable statistics.
!       * The file ups_log.txt will be produced on the
!       * io_pe (default pe 0)
      parameter ( UPS_STATISTICS_ON               = 1 )
!
!


!
```

```
!------------------------------------------------------------
!     Block from c enum: [UPS_AA_Tags_enum]
!------------------------------------------------------------
!       ---------------------------------------------------------
!       Tags used for communication (for example, mpi message tags)
!       ---------------------------------------------------------
!
!
!       Generic message tags
!       must be negative or might get confused with valid values
      parameter ( UPS_CM_ANY_TAG                    = -300 )
      parameter ( UPS_CM_ANY_SOURCE                 = -200 )
      parameter ( UPS_CM_PROC_NULL                  = -100 )
!       UPS Reserves message tags in this range
      parameter ( UPS_AA_TAG_RESERVE_START          = 100000 )
!
      parameter ( UPS_CM_SALLTOALL_TAG              = 100001 )
      parameter ( UPS_CM_TAG                        = 100002 )
      parameter ( UPS_CM_BCAST_TAG                  = 100003 )
      parameter ( UPS_CM_BCAST_FC_TAG               = 100004 )
      parameter ( UPS_CM_REDUCE_TAG                 = 100005 )
      parameter ( UPS_CM_REDUCE_FC_TAG              = 100006 )
      parameter ( UPS_CM_BARRIER_IDLE_TAG           = 100007 )
!       GS message tags
      parameter ( UPS_GS_TAG                        = 100008 )
!       IO message tags
      parameter ( UPS_IO_TAG                        = 100009 )
!       UPS Reserves message tags in this range
      parameter ( UPS_AA_TAG_RESERVE_STOP           = 100010 )
!
!


!
!------------------------------------------------------------
!     Block from c enum: [UPS_AA_Version_enum]
!------------------------------------------------------------
!       -----------------------------------------------
!       Version (do not include leading 0 if it exists)
!       -----------------------------------------------
!
!
      parameter ( UPS_VERSION                       = 20705 )
!
!


!
!------------------------------------------------------------
!     Block from c enum: [UPS_CM_More_stuff_enum]
!------------------------------------------------------------
```

```
!       obsolete - will go away
!       ---------------------------------------
!       Some more basic communication parameters.
!       ---------------------------------------
!
!
      parameter ( UPS_CM_MSG_FOUND                = 1 )
      parameter ( UPS_CM_RECV_BUFFER              = 2 )
      parameter ( UPS_CM_SEND_BUFFER              = 3 )
!
!



!
!------------------------------------------------------------
!     Block from c enum: [UPS_CM_P_group_enum]
!------------------------------------------------------------
!       ------------------------------------------------------------
!
!       Process groups categories.
!
!       Different architectures/installation settings will
!       have different values for the process groups.
!       For portability, every architecture should have all
!       the same possibilities with invalid process groups
!       defaulted to UPS_CM_P_GROUP_ALL (0).
!
!       Currently, processes grouped in terms of NUMA layers
!       in order from least specific to most specific.
!
!       See UPS_CM_P_group_item_enum below for a list of
!       items associated with process groups.
!
!       Terms used in variable definitions:
!          size: total number of processes
!          rank: id {0 through (size - 1)}
!
!
!                      Example of Layers of Process Groups
!                    2 machine groups, 4 boxes, 16 processes
!       +-----------------------------------------------------------+
!       |                         ALL                               |
!       | +-------------------------+ +--------------------------+ |
!       | |    MACHINE_GROUP 0      | |    MACHINE_GROUP 1       | |
!       | | +---------+ +---------+ | | +---------+ +---------+  | |
!       | | |  BOX 0  | |  BOX 1  | | | |  BOX 2  | |  BOX 3  |  | |
!       | | |.........| |.........| | | |.........| |.........|  | |
!       | | |SELF SELF| |SELF SELF| | | |SELF SELF| |SELF SELF|  | |
!       | | |SELF SELF| |SELF SELF| | | |SELF SELF| |SELF SELF|  | |
!       | | +---------+ +---------+ | | +---------+ +---------+  | |
!       | +-------------------------+ +--------------------------+ |
!       +----------------------------------------------------------+
!
!       ------------------------------------------------------------
!
!
!       Must start with 0 so the total count (below) is correct
!       All processes
!       idnum = 0    numids = 1
!       penum = rank numpes = size
      parameter ( UPS_CM_P_GROUP_ALL              = 0 )
!
!       set to UPS_CM_P_GROUP_ALL (0)
      parameter ( UPS_CM_P_GROUP_MACHINE_GROUP    = 0 )
!
!       usually gethostname()
      parameter ( UPS_CM_P_GROUP_BOX              = 1 )
!
!       each pe is grouped by itself.
!       idnum = rank numids = size
!       penum = 0    numpes = 1
      parameter ( UPS_CM_P_GROUP_SELF             = 2 )
!       not an process group
!       Number of different process groups
      parameter ( UPS_CM_P_GROUP_COUNT            = 3 )
!
!



!
!------------------------------------------------------------
!     Block from c enum: [UPS_CM_P_group_item_enum]
!------------------------------------------------------------
!       ------------------------------------------------------------
!       Process Group items returned from UPS_CM_P_group_item()
!       All items are with respect to UPS_CM_P_group_enum
!
!                Example of idnum, numids, numpes, penum,
!                       MSTR_CNTXT, and ALL_CNTXT
!                 wrt process group UPS_CM_P_GROUP_BOX
!                        3 boxes, 7 processes
!       +----------------------------------------------------------+
!       | numids = 3 (3 boxes - total of 7 processes)              |
!       | MSTR_CNTXT = 1 context with 3 processes                  |
!       |              (1 master [penum=0] per group)              |
!       |                                                          |
!       | +----------------+ +----------------+ +----------------+ |
!       | | idnum  = 0     | | idnum  = 1     | | idnum  = 2     | |
!       | | numpes = 3     | | numpes = 2     | | numpes = 1     | |
!       | | ALL_CNTXT A    | | ALL_CNTXT B    | | ALL_CNTXT C    | |
!       | |................| |................| |................| |
!       | | penum  = 0     | | penum  = 0     | | penum = 0      | |
!       | | penum  = 1     | | penum  = 1     | |                | |
!       | | penum  = 2     | |                | |                | |
!       | +----------------+ +----------------+ +----------------+ |
!       +----------------------------------------------------------+
!
!       ------------------------------------------------------------
!
!
!       ----------------------------------------------------
!       Start with 0 so UPS_CM_P_GROUP_ITEM_COUNT is correct
!       ----------------------------------------------------
!       ---------------------------------------------
!       Integer values containing this process's info
!       ---------------------------------------------
!       process's group id
      parameter ( UPS_CM_P_GROUP_ITEM_IDNUM       = 0 )
!       number of groups in current p group
      parameter ( UPS_CM_P_GROUP_ITEM_NUMIDS      = 1 )
!       numpes wrt process group
      parameter ( UPS_CM_P_GROUP_ITEM_NUMPES      = 2 )
!       penum wrt process group
      parameter ( UPS_CM_P_GROUP_ITEM_PENUM       = 3 )
!       -----------------------------------------------------------
!       Integer array containing info for each pe in the context.
!       The size is the number of processes in the context.
!         info[i] = information about the i-th process
!                   of the current context
!       -----------------------------------------------------------
!       process group idnums for pes in context
      parameter ( UPS_CM_P_GROUP_ITEM_IDNUMS      = 4 )
!       process group penums for pes in context
      parameter ( UPS_CM_P_GROUP_ITEM_PENUMS      = 5 )
!       global penums for pes in context
!       ( ie penums wrt global context
!       {like MPI_COMM_WORLD} )
      parameter ( UPS_CM_P_GROUP_ITEM_G_PENUMS    = 6 )
!       -----------------------------------------------------------------
!       UPS_CM_Context contexts
!       Contexts returned are of the same type as those of the underlying
!       communication protocol.
!       For example, if MPI is the communication protocol, the following
!       are true:
!         1) UPS will have used MPI_Comm_split() in order to make the
!            contexts being passed back via UPS_CM_P_group_item.
!         2) These contexts can be passed into UPS or MPI calls with the
!            special reminder that UPS_CM_Set_context cannot take
!            MPI_COMM_NULL as an argument.  This is most common when
!            a non-master pe requests UPS_CM_P_GROUP_ITEM_MSTR_CNTXT.
!            In this case, the context returned is MPI_COMM_NULL
!       -----------------------------------------------------------------
!       pe's assembled by process group
```

```
      parameter ( UPS_CM_P_GROUP_ITEM_ALL_CNTXT   = 7 )
!       penum=0 pe's in the process group
      parameter ( UPS_CM_P_GROUP_ITEM_MSTR_CNTXT  = 8 )
!
!       number of items in UPS_CM_P_group_item_enum
      parameter ( UPS_CM_P_GROUP_ITEM_COUNT       = 9 )
!
!



!
!-------------------------------------------------------------
!     Block from c enum: [UPS_CM_Protocol_context_compare_enum]
!-------------------------------------------------------------
!       ------------------------------
!       Comparison of protocol context
!       ------------------------------
!
!
!       identical
      parameter ( UPS_PROTOCOL_CONTEXT_IDENT      = 0 )
!       ranks preserved
      parameter ( UPS_PROTOCOL_CONTEXT_CONGRUENT  = 1 )
!       ranks not preserved
      parameter ( UPS_PROTOCOL_CONTEXT_SIMILAR    = 2 )
!       otherwise
      parameter ( UPS_PROTOCOL_CONTEXT_UNEQUAL    = 3 )
!
!



!
!-------------------------------------------------------------
!     Block from c enum: [UPS_DT_Boolean_Constants_enum]
!-------------------------------------------------------------
!       ---------------------
!       define true and false
!       ---------------------
!
!
      parameter ( UPS_DT_TRUE                     = 1 )
      parameter ( UPS_DT_FALSE                    = 0 )
!
!



!
!-------------------------------------------------------------
!     Block from c enum: [UPS_DT_Datatype_enum]
!-------------------------------------------------------------
!       ---------------------------------
!       all the datatypes ups understands
!       ---------------------------------
!
!
      parameter ( UPS_DT_DOUBLE                   = 1 )
      parameter ( UPS_DT_REAL8                    = 1 )
!
      parameter ( UPS_DT_FLOAT                    = 2 )
      parameter ( UPS_DT_REAL4                    = 2 )
!
!       Fortran logicals are often of type UPS_DT_INT
      parameter ( UPS_DT_INT                      = 3 )
      parameter ( UPS_DT_INT4                     = 3 )
!
!       C++ bools are often of type UPS_DT_CHAR
      parameter ( UPS_DT_CHAR                     = 4 )
      parameter ( UPS_DT_CHARACTER                = 4 )
      parameter ( UPS_DT_BYTE                     = 4 )
!
      parameter ( UPS_DT_LONG                     = 5 )
!       C long long == Fortran int8 == C long
      parameter ( UPS_DT_LONG_LONG                = 5 )
      parameter ( UPS_DT_INT8                     = 5 )
!
      parameter ( UPS_DT_VOID                     = 6 )
!
      parameter ( UPS_DT_SHORT                    = 7 )
!
      parameter ( UPS_DT_DOUBLE_INT               = 101 )
      parameter ( UPS_DT_2REAL8                   = 101 )
!
      parameter ( UPS_DT_FLOAT_INT                = 102 )
      parameter ( UPS_DT_2REAL4                   = 102 )
!
      parameter ( UPS_DT_2INT                     = 103 )
      parameter ( UPS_DT_2INT4                    = 103 )
!
      parameter ( UPS_DT_LONG_INT                 = 104 )
!       C long long == Fortran int8 == C long
      parameter ( UPS_DT_2INT8                    = 104 )
!
      parameter ( UPS_DT_SHORT_INT                = 105 )
!       -------------------------------------------------------------
!       special datatypes that can only be passed to certain functions
!       -------------------------------------------------------------
!       IO package routines accept null-terminated strings.
!       A UPS_DT_STRING of length 1 will be treated as a UPS_DT_CHAR
      parameter ( UPS_DT_STRING                   = 106 )
!       pass to UPS_DT_Sizeof() for size of UPS_DT_TIME_TYPE
      parameter ( UPS_DT_TIME_TYPE_DT             = 107 )
!       pass to UPS_DT_Sizeof() for size of protocol context variable
!       (eg for the MPI protocol, the size of MPI_Comm)
      parameter ( UPS_DT_PROTOCOL_COMM            = 108 )
!       pass to UPS_DT_Sizeof() for size of protocol request variable
!       (eg for the MPI protocol, the size of MPI_Request)
      parameter ( UPS_DT_PROTOCOL_REQUEST         = 109 )
!
!


!
!-------------------------------------------------------------
!     Block from c enum: [UPS_ER_OUTPUT_enum]
!-------------------------------------------------------------
!       -------------------------------------------------------------
!       Specifies where UPS error output will be sent.
!       Use UPS_AA_Opt_get/UPS_AA_Opt_set with UPS_AA_OPT_ERROUTPUT_TYPE.
!       You may set the following as environmental variables as well:
!         (in mutually exclusive groups)
!            - setenv UPS_ER_OUTPUT_DEFAULT
!              setenv UPS_ER_OUTPUT_NONE
!              setenv UPS_ER_OUTPUT_FILE
!       Remember that all environment variables must be propagated to
!       all the processes.
!       -------------------------------------------------------------
!
!
!       By default, UPS error output is sent to stderr
      parameter ( UPS_ER_OUTPUT_DEFAULT           = 0 )
!       no UPS error output is done
      parameter ( UPS_ER_OUTPUT_NONE              = 1 )
!       UPS error output is appended to ups_err.txt
      parameter ( UPS_ER_OUTPUT_FILE              = 2 )
!
!


!
!-------------------------------------------------------------
!     Block from c enum: [UPS_GS_Index_type_enum]
!-------------------------------------------------------------
!       -------------------------------------------
!       gather-scatter first index referring to the
!       local pe array or the global array
!       -------------------------------------------
!
!
      parameter ( UPS_GS_LOCAL_INDEX              = 0 )
      parameter ( UPS_GS_GLOBAL_INDEX             = 1 )
!
!
```

```
!
!------------------------------------------------------------
!     Block from c enum: [UPS_GS_Item_enum]
!------------------------------------------------------------
!       ------------------------------------------------------
!       Listing of items that can be obtained about a GS id via
!       UPS_GS_Get_item()
!       (eg items useful for the scatter_list and gather_list
!       functions are found here)
!       ------------------------------------------------------
!
!
!       -------------------------------------------------------------
!       (int4 array) A listing of the global indices that are accessed
!       during a gather or scatter operation.
!       Indices not accessed (eg skipped) will not be listed in this
!       array.
!       The array returned will have indices that are 0 based with 0
!       corresponding to the first index owned by the calling process.
!       The array length is given by UPS_GS_TOTAL_INDICES_ACCESSED.
!       -------------------------------------------------------------
      parameter ( UPS_GS_INDICES_ACCESSED         = 0 )
!       -------------------------------------------------------------
!       (int4 array) The number of times an accessed index is actually
!       accessed (eg there might be a communication pattern where 3
!       values get scatter-sum'd to the same index.
!       Only the indices owned by the process that are accessed will
!       be in the array.  This can be used in conjunction with
!       UPS_GS_INDICES_ACCESSED to get values if the index that is
!       accessed.
!       The array length is given by UPS_GS_TOTAL_INDICES_ACCESSED.
!       -------------------------------------------------------------
      parameter ( UPS_GS_NUM_INDICES_ACCESSED     = 1 )
!       -------------------------------------------------------------
!       (int4 array) Like UPS_GS_NUM_INDICES_ACCESSED, but indices not
!       accessed (eg skips) will have a num of 0 instead of not being
!       listed.
!       The number of elements in this array go from index 0 through
!       the last index accessed.  This is NOT necessarily the same
!       size as the original my_space_size supplied in UPS_GS_Setup.
!       It will be smaller than my_space_size if the last accessed
!       index comes before the last index owned.
!       One might wish to pass in an array of size my_space_size that
!       has been initialized to 0's.  That way, the indices at the end
!       that are not overwritten will have the correct value of 0.
!       -------------------------------------------------------------
      parameter ( UPS_GS_NUM_INDICES_ALL          = 2 )
!       -------------------------------------------------------------
!       (int4) The total number of indices the calling process owns
!       that are accessed during a gather or scatter operation.
!       This is used with items like UPS_GS_INDICES_ACCESSED and
!       UPS_GS_NUM_INDICES_ACCESSED
!       ------------------------------------------------------------
      parameter ( UPS_GS_TOTAL_INDICES_ACCESSED   = 3 )
!       ------------------------------------------------------------
!       (int4) The number of elements in the list buffer output
!       for scatter_list or the list buffer input for gather_list.
!       This is a sum of UPS_GS_NUM_INDICES_ACCESSED.
!       ------------------------------------------------------------
      parameter ( UPS_GS_SUM_NUM_INDICES_ACCESSED = 4 )
!       ------------------------------------------------------------
!       (int4) The value of the calling process's first index with
!       respect to the global array.  Process 0's first index will
!       have a value of 0.  This can be used in conjunction with
!       UPS_GS_INDICES_ACCESSED to get values that are globally based
!       instead of locally based.
!       Depending upon which GS setup call was used, this value might
!       not be known.  In this case, the value is -1.
!       ------------------------------------------------------------
      parameter ( UPS_GS_START_INDEX_WRT_GLOBAL   = 5 )
!       ------------------------------------------------------------
!       (int4) The number of indices owned.
!       This corresponds to the my_space_size argument for
!       UPS_GS_Setup and UPS_GS_Setup_s_global calls.
!       Depending upon which GS setup call was used, this value might
!       not be known.  In this case, the value is -1.
!       ------------------------------------------------------------
      parameter ( UPS_GS_NUM_INDICES_OWNED        = 6 )
!       ------------------------------------------------------------
!       (int4) The number of elements in the global indices array
!       passed into the GS Setup calls (the count argument).
!       ------------------------------------------------------------
      parameter ( UPS_GS_NUM_GLOBAL_INDICES       = 7 )
!       ------------------------------------------------------------
!       (int4 array) The size of this array can be obtained by
!       UPS_GS_NUM_GLOBAL_INDICES.
!       If given the global indices in the GS Setup call, this returns
!       a 0 based element offset into the global array for the global
!       indices supplied.
!       If the global indices were not given (UPS_GS_Setup_s_local),
!       then a value of -1 will be assigned to each element in the
!       array.
!       Note that negative indices might have been supplied to the
!       GS setup call.  So, values of -1 in the array returned does
!       necessarily mean that global indices were not given in the
!       GS setup call.
!       ------------------------------------------------------------
      parameter ( UPS_GS_GLOBAL_INDICES           = 8 )
!       ------------------------------------------------------------
!       (int4) The original start index for process 0.
!       This value can be obtained if given the my_start_index
!       argument (eg calling UPS_GS_Setup or UPS_GS_Setup_s_global).
!       As with UPS_GS_GLOBAL_INDICES, an artificial value of -1 will
!       be set if my_start_index was not given (UPS_GS_Setup_s_local).
!       Note that the items UPS_GS_GLOBAL_INDICES and
!       UPS_GS_INDICES_ACCESSED still return 0 based arrays.
!       ---------------------------------------------------------------
      parameter ( UPS_GS_START_INDEX             = 9 )
!       ---------------------------------------------------------------
!       (int4 array) The size of this array can be obtained by
!       UPS_GS_NUM_GLOBAL_INDICES.
!       Regardless of which GS setup call made, UPS_GS_INDEX_PE and
!       UPS_GS_INDEX_VALUE can be obtained.  The values of these
!       correspond to the index_pe and index_value arguments to
!       UPS_GS_Setup_s_local() call.  See that function for more
!       information.
!       Each value of this array contains the process number that owns
!       the corresponding index.
!       Possible values:
!          UPS_GS_INDEX_ZERO:  scatter skips, gather zeros
!          UPS_GS_INDEX_SKIP:  scatter skips, gather skips
!          0 through numpes-1: process the index resides on
!       ---------------------------------------------------------------
      parameter ( UPS_GS_INDEX_PE                = 10 )
!       ---------------------------------------------------------------
!       (int array) The size of this array can be obtained by
!       UPS_GS_NUM_GLOBAL_INDICES.
!       Regardless of which GS setup call made, UPS_GS_INDEX_PE and
!       UPS_GS_INDEX_VALUE can be obtained.  The values of these
!       correspond to the index_pe and index_value arguments to
!       UPS_GS_Setup_s_local() call.  See that function for more
!       information.
!       Each value of this array contains the 0 based index into the
!       local array of the process that owns the corresponding index.
!       If the corresponding index_pe is a special index (less than 0),
!       the corresponding index_value is not used.
!       Possible values:
!          UPS_GS_INDEX_ZERO:  scatter skips, gather zeros
!          UPS_GS_INDEX_SKIP:  scatter skips, gather skips
!          >=0:                0 based index into local array
!       ---------------------------------------------------------------
      parameter ( UPS_GS_INDEX_VALUE             = 11 )
!       ---------------------------------------------------------------
!       (real8) The effectiveness of trying compression
!       Values will range from 0 (no compression) to 1 (best
!       compression
!       ---------------------------------------------------------------
      parameter ( UPS_GS_COMP_EFFECT             = 12 )
!       ---------------------------------------------------------------
!       The following are not implemented...but could be if requested.
!       ---------------------------------------------------------------
!       not implemented
      parameter ( UPS_GS_MEANING_OF_LIFE         = 13 )
```

```
!
!


!
!------------------------------------------------------------
!     Block from c enum: [UPS_GS_Setup_study_type_enum]
!------------------------------------------------------------
!       -------------------------------------------------------------
!       Some communication patterns should be handled differently than
!       others.
!       This routine allows the user to tell UPS to look for certain
!       things when doing the setup routine.
!       -------------------------------------------------------------
!
!
!       Index compression:
!       It wastes time to send repeated indices during
!       a gather operation.  The repeated index should be sent once and
!       copied on the receiving end.  If the communication pattern has many
!       repeated indices, spending extra time in the GS-setup routine to
!       detect repeated indices will pay off if enough GS-gather or
!       GS-scatter calls are performed.
!       Default: UPS_GS_INDEX_COMPRESSION_ON
      parameter ( UPS_GS_INDEX_COMPRESSION_OFF    = 0 )
      parameter ( UPS_GS_INDEX_COMPRESSION_ON     = 1 )
!       send/recv ordering:
!       Order send/recvs so that all off-box sends are started first and
!       all off-box recvs are completed and processed last.
!       If off, sends and recvs are with penum ordering
!       Default: UPS_GS_COMM_ORDERING_ON
      parameter ( UPS_GS_COMM_ORDERING_OFF        = 2 )
      parameter ( UPS_GS_COMM_ORDERING_ON         = 3 )
!
!


!
!------------------------------------------------------------
!     Block from c enum: [UPS_GS_Setup_type_enum]
!------------------------------------------------------------
!       -----------------------------------------
!       Used internally to define which method to
!       use for setting up a gs trace (depending
!       on what data is supplied).
!       -----------------------------------------
!
!
!       indices positioned linearly
      parameter ( UPS_GS_SETUP_DENSE_METHOD       = 0 )
!       indices positioned from input array and defined by global value
```

```
      parameter ( UPS_GS_SETUP_S_GLOBAL_METHOD    = 1 )
!       indices positioned from input array and defined by pe/local value
      parameter ( UPS_GS_SETUP_S_LOCAL_METHOD     = 2 )
!
!



!
!------------------------------------------------------------
!     Block from c enum: [UPS_GS_Special_index_enum]
!------------------------------------------------------------
!       ----------------------------------------------------------------------
!       negative consecutive constants used for "mask like" behavior in the
!       gather/scatter calls.  For example, if an index has the value of:
!           first index of process 0 + UPS_GS_INDEX_ZERO
!       then the gather call will zero the output for that index and the scatter
!       call will not contribute that element
!       ----------------------------------------------------------------------
!
!
!       --------------
!       skip the index
!       --------------
      parameter ( UPS_GS_INDEX_SKIP               = -1 )
!       --------------------------
!       scatter skips, gather zeros
!       --------------------------
      parameter ( UPS_GS_INDEX_ZERO               = -2 )
!       ---------------------------------------------
!       any index at or below this marker will be skipped
!       ---------------------------------------------
      parameter ( UPS_GS_INDEX_FINAL              = -3 )
!
!



!
!------------------------------------------------------------
!     Block from c enum: [UPS_IO_ACCESS_PES_enum]
!------------------------------------------------------------
!       --------------------------------------------------------------
!       How the file is being accessed.
!
!         NOTE: currently, for reading UPS_IO_ACCESS_ALL_PE is assumed.
!               The discussion below deals with writing datasets.
!               File access changes for reading can be added if
!               requested.
!
!       Use UPS_AA_Opt_get/UPS_AA_Opt_set with UPS_IO_OPT_ACCESS_WRITE
!       The value cannot be changed for a file while it is opened.
!       In other words, make the call to UPS_AA_Opt_set before the call
!       to UPS_IO_File_open and do not change it until after
!       the call to UPS_IO_File_close.
!       You may set the following as environmental variables as well:
!         (in mutually exclusive groups)
!            - setenv UPS_IO_ACCESS_IO_PE
!              setenv UPS_IO_ACCESS_ALL_PE
!              setenv UPS_IO_ACCESS_AGGREGATION
!              setenv UPS_IO_ACCESS_LINEAR
!       Remember that all environment variables must be propagated to
!       all the processes.
!
!       Performance:
!          In general, processes will either "own" a contiguous chunk
!          of the global dataset (eg each process owns a block of rows
!          of a 2d dataset) or a non-contiguous chunk of the global
!          dataset (eg a process might own a 50X50X50 box of a global
!          1000X1000X1000 dataset).
!
!          Contiguous Case Performance:
!              Since there will likely be little contention between the
!              processes, the speed of writing will be (from fastest to
!              slowest):
!                  UPS_IO_ACCESS_ALL_PE
!                  UPS_IO_ACCESS_LINEAR (not implemented)
!                  UPS_IO_ACCESS_AGGREGATION
!                  UPS_IO_ACCESS_IO_PE
!
!          Non-Contiguous Case Performance:
!              There is a lot of contention between the processes since
!              their data is distributed cyclically in the dataset.
!              The speed of writing will be (from fastest to slowest):
!                  UPS_IO_ACCESS_LINEAR (not implemented)
!                  UPS_IO_ACCESS_AGGREGATION
!                  UPS_IO_ACCESS_IO_PE
!                  UPS_IO_ACCESS_ALL_PE
!
!          Current UPS_IO_PROTOCOL_HDF (default protocol) Performance:
!              When all processes have data to write to a dataset,
!              HDF can use "collective IO".  In this case, regardless
!              of contiguous or non-contiguous data, the speed of
!              writing will be (from fastest to slowest).
!                  UPS_IO_ACCESS_LINEAR (not implemented)
!                  UPS_IO_ACCESS_ALL_PE
!                  UPS_IO_ACCESS_AGGREGATION
!                  UPS_IO_ACCESS_IO_PE
!
!          See below for a description of the different types of file
!          access.
!       ---------------------------------------------------------------
!
!
```

```
!       (default) write access to file limited to
!       * the io_pe.
!       * See UPS_AA_Io_pe_set to set the io_pe.
!       * All pes must still call collective IO
!       * routines, but only the io pe accesses the
!       * file.
!       * Communication is done to collect the data
!       * to the io pe for writing.
      parameter ( UPS_IO_ACCESS_IO_PE             = 0 )
!       all pes access the file.
!       * Every process writes their own chunk of the
!       * global dataset.
      parameter ( UPS_IO_ACCESS_ALL_PE            = 1 )
!       all pes may access the file.
!       * Data written to file is first aggregated
!       * to "master processes" then written.
!       * This allows datasets to maintain their
!       * expected form (dimensions/where data is)
!       * while increasing the performance by only
!       * writing blocks of contiguous data.
      parameter ( UPS_IO_ACCESS_AGGREGATION       = 2 )
!       all pes access the file.
!       * NOT IMPLEMENTED
!       * All processes write their portion of the
!       * global dataset in linear contiguous chunks.
!       * The advantage to this is the writes are
!       * fast.  However, the datasets must then be
!       * read with UPS since their shape and the
!       * location of values has been altered.
      parameter ( UPS_IO_ACCESS_LINEAR            = 3 )
!
!



!
!-------------------------------------------------------------
!     Block from c enum: [UPS_IO_FILE_OBJECT_TYPE_enum]
!-------------------------------------------------------------
!       ------------------------------
!       types of io objects in the file
!       ------------------------------
!
!
!       an attribute attached to an object
      parameter ( UPS_IO_FILE_OBJECT_ATTRIBUTE    = 0 )
!       a dataset (ie unix file)
      parameter ( UPS_IO_FILE_OBJECT_DATASET      = 1 )
!       a group (ie unix directory)
      parameter ( UPS_IO_FILE_OBJECT_GROUP        = 2 )
!
!
```

```
!
!------------------------------------------------------------
!     Block from c enum: [UPS_IO_FILTER_TYPE_enum]
!------------------------------------------------------------
!       --------------------------------------------------------------
!       types of values for UPS_IO_Filter_get and UPS_IO_Filter_set
!       Currently, filters are only applicable to the following calls:
!          UPS_IO_Info_count
!          UPS_IO_Info_create
!       --------------------------------------------------------------
!
!
!       ------------
!       INFO filters
!       ------------
!       Purpose: get or set the filter applied to
!       *   UPS_IO_Info_count/UPS_IO_Info_create calls.
!       * Datatype: char array
!       * Get: yes
!       * Set: yes
!       * The filter to apply that will modify the
!       * number of matches to UPS_IO_Info_count and
!       * UPS_IO_Info_create calls when listing members
!       * or attributes.
!       *
!       * A filter of 0 length signifies that no filter
!       * will be applied.
!       *
!       * Currently, the only filters available are the
!       * following:
!       *   - object_name
!       *     This will do a match for object_name in the
!       *     members in the list.
!       *     This effectively limits the number of
!       *     matches to 1 (if found) or 0 (if not found).
!       *   - * /object_name
!       *     This will also match object_name
!       *     recursively through any subgroups.
!       *     There are no space between * and /.
!       *     Not valid for UPS_IO_INFO_LIST_ATTRIBUTES.
!       *   - * /
!       *     This will do a recursive listing of all
!       *     members.
!       *     There are no space between * and /.
!       *     Not valid for UPS_IO_INFO_LIST_ATTRIBUTES.
      parameter ( UPS_IO_FILTER_INFO              = 0 )
!       Purpose: get the number of chars (not
!       *   null-terminated) of the info filter.
!       * Datatype: int8
!       * Get: yes
!       * Set: no
```

```
      parameter ( UPS_IO_FILTER_INFO_LENGTH       = 1 )
!
!


!
!------------------------------------------------------------
!     Block from c enum: [UPS_IO_INFO_ITEM_enum]
!------------------------------------------------------------
!       ------------------------------------------------------------------
!       Values for UPS_IO_Info_item_(get|set) for an info_id
!       * info_ids are obtained from UPS_IO_Info_create
!       * Form:
!       *   Purpose: simple description
!       *   Datatype: datatype of item
!       *   Get: UPS_IO_Info_create options you can UPS_IO_Info_item_get
!       *   Set: UPS_IO_Info_create options you can UPS_IO_Info_item_get
!       *   Reading: The effect when reading (eg dataset read)
!       *   Writing: The effect when writing (eg dataset write)
!       *   More description
!       *
!       * NOTE: When the arguments to a function include the info_id and
!       *       conflicting arguments (eg an info_id might have the value of
!       *       UPS_IO_INFO_NAME and the function UPS_IO_Dataset_write requires
!       *       the info_id and the "name" argument, the argument overrides the
!       *       value implied with the info_id.
!       * NOTE: When dealing with attributes, the entire attribute is used
!       *       as the "dataset section".  For example, NITEMS and NITEMS_TOTAL
!       *       will be the total number of items in the entire attribute
!       *       regardless of which pe wrote the attribute.
!       ------------------------------------------------------------------
!
!
!       --------------------------------
!       Dataset Distribution Information
!       --------------------------------
!       ----------------------------------------------------------------
!       Example of a dataset section:
!                 (row, column) pairs
!                 .............
!                 .(0,0) (0,1).
!        [begin]  .(1,0) (1,1).
!                 .(2,0) (2,1).
!        [end]    .(3,0) (3,1).
!                 .(4,0) (4,1).
!                 .............
!       DIMS:       [local num rows, local num columns]      = [3,2]
!       DIMS_TOTAL: [total num rows, total num columns       = [5,2]
!       NDIMS:      1 row and 1 column                       = 2
!       STARTS:      [starting row, starting column]         = [1,0]
!       NITEMS:       num items in dataset section           = 3 * 2
!       NITEMS_TOTAL: num items in dataset                  = 5 * 2
!       -------------------------------------------------------------
!       Purpose: each dimension count of dataset section
!       * Datatype: int8 array size UPS_IO_INFO_NDIMS
!       * Get: UPS_IO_INFO_DATA_DIST
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Set: UPS_IO_INFO_DATA_DIST
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Reading: The default sets this value to
!       *          read in what this pe number wrote
!       *          out. If this is not possible (eg
!       *          the number of pes that read does
!       *          not equal the number of pes that
!       *          wrote, the value is set to read
!       *          the entire dataset.
!       *          The user may change the value
!       *          of this array if they wish to read
!       *          in a different dataset section.
!       *          For example, to read in the whole
!       *          dataset, set dims to dims_total and the
!       *          starts array to 0s.
!       * Writing: Dimensions of dataset section to
!       *          write.
!       * The first element of this array is local
!       * rows and the second (if it exists) is local
!       * columns.
!       * In memory, data is a 1d array going along
!       * columns first.
!       * For example, a 5 row, 2 column dataset would
!       * lay in memory contiguously like:
!       *   r0c0,r0c1,r1c0,r1c1,r2c0,r2c1,r3c0,r3c1,...
      parameter ( UPS_IO_INFO_DIMS                = 0 )
!       Purpose: each dimenstion count of dataset
!       * Datatype: int8 array size UPS_IO_INFO_NDIMS
!       * Get: UPS_IO_INFO_DATA_DIST
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Set:
!       * Reading: This value is used for
!       *          user inquiry purpose only and must
!       *          not be modified.
!       * Writing: This value must not be changed.
!       *          It is derived from all the
!       *          processes dims settings.
!       * This value is for the whole dataset.
!       * See UPS_IO_INFO_DIMS above.
      parameter ( UPS_IO_INFO_DIMS_TOTAL          = 1 )
!       Purpose: number of dimensions (local=global)
!       * Datatype: int4
!       * Get: UPS_IO_INFO_DATA_DIST
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Set: UPS_IO_INFO_DATA_DIST
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Reading: This value is used for
!       *          user inquiry purpose only and must
!       *          not be modified.
!       * Writing: This value must be set.
!       * This must either be in the set
!       * [1,UPSP_IO_MAX_DIMS] (currently set to 10).
      parameter ( UPS_IO_INFO_NDIMS               = 2 )
!       Purpose: number of items in dataset section
!       * Datatype: int8
!       * Get: UPS_IO_INFO_DATA_DIST
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Set:
!       * Reading: This value is used for
!       *          user inquiry purpose only and must
!       *          not be modified.
!       * Writing: This value is used for
!       *          user inquiry purpose only and must
!       *          not be modified.
!       * number of items in a dims dimensioned section
      parameter ( UPS_IO_INFO_NITEMS              = 3 )
!       Purpose: number of items in dataset
!       * Datatype: int8
!       * Get: UPS_IO_INFO_DATA_DIST
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Set:
!       * Reading: This value is used for
!       *          user inquiry purpose only and must
!       *          not be modified.
!       * Writing: This value is used for
!       *          user inquiry purpose only and must
!       *          not be modified.
!       * size of dims_total dimensioned section
      parameter ( UPS_IO_INFO_NITEMS_TOTAL        = 4 )
!       Purpose: Which (row,column) the data starts
!       * Datatype: int8 array size UPS_IO_INFO_NDIMS
!       * Get: UPS_IO_INFO_DATA_DIST
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Set: UPS_IO_INFO_DATA_DIST
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Reading: The default sets this value to
!       *          read in what this pe number wrote
!       *          out. If this is not possible (eg
!       *          the number of pes that read does
!       *          not equal the number of pes that
!       *          wrote, the value is set to read
!       *          the entire dataset.
!       *          The user may change the value
!       *          of this array if they wish to read
!       *          in a different dataset section.
!       *          For example, to read in the whole
!       *          dataset, set dims to dims_total and the
!       *          starts array to 0s.
!       * Writing: This value is defaulted to less
!       *          than 0 which indicates each process
!       *          is writing a consecutive contiguous
!       *          chunk.
!       *          However, they user may set the starts
!       *          array if they wish to override the
!       *          default behavior.
      parameter ( UPS_IO_INFO_STARTS              = 5 )
!       Purpose: number of dimensions of process grid
!       * Datatype: int8 array size UPS_IO_INFO_NDIMS
!       * Get:
!       * Set: UPS_IO_INFO_DATA_DIST
!       * Reading: Not used
!       * Writing: The user may define how the dataset
!       *          is partitioned via a process grid.
!       *          This array defines the number of
!       *          processes along each dimension of
!       *          the dataset.
      parameter ( UPS_IO_INFO_PGRID_DIMS          = 11 )
!       Purpose: penum ordering in the process grid
!       * Datatype: int8 array size Prod(PGRID_DIMS)
!       *           This might be larger than numpes
!       *           given UPS_IO_INFO_PGRID_DIMS.
!       * Get:
!       * Set: UPS_IO_INFO_DATA_DIST
!       * Reading: Not used
!       * Writing: The default ordering is the same
!       *          order as the data buffer.  That
!       *          is, along the last dimension
!       *          first.  The user may override
!       *          that default by setting this.
      parameter ( UPS_IO_INFO_PGRID_ORDER         = 12 )
!       ----------------
!       List Information
!       ----------------
!       Purpose: name of item
!       * Datatype: char*
!       * Get: UPS_IO_INFO_LIST_ATTRIBUTES
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Set:
!       * Reading: overridden with non-path part of name
!       * Writing: overridden with non-path part of name
!       * The name will be relative to the path supplied
!       * to the info create call.
!       * Note: the return string not null-terminated.
      parameter ( UPS_IO_INFO_NAME                = 6 )
!       Purpose: length - no null-terminator
!       * Datatype: int8
!       * Get: UPS_IO_INFO_LIST_ATTRIBUTES
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Set:
!       * Reading: overridden
!       * Writing: overridden
      parameter ( UPS_IO_INFO_NAME_LENGTH         = 7 )
!       Purpose: full path to item
!       * Datatype: char*
!       * Get: UPS_IO_INFO_LIST_ATTRIBUTES
!       *      UPS_IO_INFO_LIST_MEMBERS
!       *      UPS_IO_INFO_DATA_DIST
!       * Set:
!       * Reading: overridden
!       * Writing: overridden
!       * For attributes, the path is the path to which
!       * the object is attached.
!       * Note: the return string not null-terminated.
      parameter ( UPS_IO_INFO_PATH                = 14 )
!       Purpose: length - no null-terminator
!       * Datatype: int8
!       * Get: UPS_IO_INFO_LIST_ATTRIBUTES
!       *      UPS_IO_INFO_LIST_MEMBERS
!       *      UPS_IO_INFO_DATA_DIST
!       * Set:
!       * Reading: overridden
!       * Writing: overridden
      parameter ( UPS_IO_INFO_PATH_LENGTH         = 15 )
!       Purpose: object type of item
!       * Datatype: UPS_IO_FILE_OBJECT_TYPE_enum
!       * Get: UPS_IO_INFO_LIST_ATTRIBUTES
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Set:
!       * Reading: overridden
!       * Writing: overridden
      parameter ( UPS_IO_INFO_OBJECT_TYPE         = 8 )
!       -------------------
!       General Information
!       -------------------
!       Purpose: datatype (dataset|attribute)
!       * Datatype: UPS_DT_Datatype_enum
!       * Get: UPS_IO_INFO_DATA_DIST
!       *      UPS_IO_INFO_LIST_ATTRIBUTES
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Set:
!       * Reading: overridden
!       * Writing: overridden
      parameter ( UPS_IO_INFO_DATATYPE            = 9 )
!       Purpose: size datatype (dataset|attribute)
!       * Datatype: int8
!       * Get: UPS_IO_INFO_DATA_DIST
!       *      UPS_IO_INFO_LIST_ATTRIBUTES
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Set:
!       * Reading: overridden
!       * Writing: overridden
!       *   (if obj is dataset|attribute)
!       * When the datatype is UPS_DT_STRING, the
!       * size returned will be the same as
!       * UPS_DT_CHAR.  Use UPS_IO_INFO_NITEMS to
!       * obtain the number of chars in the string
!       * (no additional null-terminator added).
!       -------
!       special
!       -------
      parameter ( UPS_IO_INFO_DATATYPE_SIZE       = 10 )
!       Purpose: set info_id to read entire dataset
!       * Datatype: (na) not used
!       * Get:
!       * Set: UPS_IO_INFO_DATA_DIST
!       *      UPS_IO_INFO_LIST_MEMBERS
!       * Reading: To read the entire dataset, get
!       *          an info_id then set this option.
!       *          By default, each process reads in
!       *          only what it wrote out.
!       * Writing: This value must not be set as it
!       *          will overwrite DIMS with
!       *          DIMS_TOTAL and set STARTS to 0.
      parameter ( UPS_IO_INFO_DATA_TOTAL          = 13 )
!
!


!
!-----------------------------------------------------------
!     Block from c enum: [UPS_IO_INFO_TYPE_enum]
!-----------------------------------------------------------
!       -----------------
!       types of info_ids
!       -----------------
!
!
!       distribution of data for a dataset
      parameter ( UPS_IO_INFO_DATA_DIST           = 0 )
!       attributes of the object
      parameter ( UPS_IO_INFO_LIST_ATTRIBUTES     = 1 )
!       members of the object (must be a group)
      parameter ( UPS_IO_INFO_LIST_MEMBERS        = 2 )
!
!


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

```
!     Block from c enum: [UPS_IO_LOC_ITEM_enum]
!------------------------------------------------------------
!       -----------------------------------------------------------------
!       * Values for UPS_IO_Loc_item_(get|set) for a loc_id
!       * loc_ids are obtained from UPS_IO_File_open and UPS_IO_Group_open
!       * Form:
!       *   Purpose: simple description
!       *   Datatype: datatype of item
!       *   Get: (yes/no)
!       *   Set: (yes/no)
!       *   More description
!       * -----------------------------------------------------------------
!
!
!       Purpose: the full path of the loc id
!       * Datatype: char array size UPS_IO_LOC_PATH_LENGTH
!       * Get: yes
!       * Set: no
!       * This is the path internal to the file.
!       * A file id has a length of 0.
!       * See UPS_IO_LOC_FILE_NAME
      parameter ( UPS_IO_LOC_PATH                 = 0 )
!       Purpose: the length of UPS_IO_LOC_PATH
!       * Datatype: int8
!       * Get: yes
!       * Set: no
!       * A file id has a length of 0.
      parameter ( UPS_IO_LOC_PATH_LENGTH          = 1 )
!       Purpose: the full path of the loc id
!       * Datatype: char array size
!       *           UPS_IO_LOC_FILE_NAME_LENGTH
!       * Get: yes
!       * Set: no
!       * This is the path to the file itself.
!       * See UPS_IO_LOC_PATH
      parameter ( UPS_IO_LOC_FILE_NAME            = 2 )
!       Purpose: length of UPS_IO_LOC_FILE_NAME
!       * Datatype: int8
!       * Get: yes
!       * Set: no
      parameter ( UPS_IO_LOC_FILE_NAME_LENGTH     = 3 )
!       Purpose: the file_id containing this loc_id
!       * Datatype: int4
!       * Get: yes
!       * Set: no
      parameter ( UPS_IO_LOC_FILE_ID              = 4 )
!       Purpose: name of sequential dataset to read
!       * Datatype: char array size
!       *           UPS_IO_LOC_DS_NEXT_R_LENGTH
!       * Get: yes
!       * Set: no
!       * The next sequential read will use this
!       * dataset name.
      parameter ( UPS_IO_LOC_DS_NEXT_R            = 5 )
!       Purpose: length of UPS_IO_LOC_DS_NEXT_R
!       * Datatype: int8
!       * Get: yes
!       * Set: no
      parameter ( UPS_IO_LOC_DS_NEXT_R_LENGTH     = 6 )
!       Purpose: name of sequential dataset to write
!       * Datatype: char array size
!       *          UPS_IO_LOC_DS_NEXT_W_LENGTH
!       * Get: yes
!       * Set: no
!       * The next sequential write will use this
!       * dataset name.
      parameter ( UPS_IO_LOC_DS_NEXT_W            = 7 )
!       Purpose: length of UPS_IO_LOC_DS_NEXT_W
!       * Datatype: int8
!       * Get: yes
!       * Set: no
      parameter ( UPS_IO_LOC_DS_NEXT_W_LENGTH     = 8 )
!       Purpose: number of sequential dataset to read
!       * Datatype: int8
!       * Get: yes
!       * Set: yes
!       * After a read, this number will be automatically
!       * incremented by one.
      parameter ( UPS_IO_LOC_DS_NUM_R             = 9 )
!       Purpose: number of sequential dataset to write
!       * Datatype: int8
!       * Get: yes
!       * Set: yes
!       * After a read, this number will be automatically
!       * incremented by one.
      parameter ( UPS_IO_LOC_DS_NUM_W             = 10 )
!       Purpose: get the protocol flavor of a loc_id
!       * Datatype: int4
!       * Get: yes
!       * Set: no
!       * For example, if using the protocol
!       * UPS_IO_PROTOCOL_HDF, you can get the HDF
!       * handle of the loc id for use in your own
!       * HDF calls.
      parameter ( UPS_IO_LOC_ID_PROTOCOL          = 11 )
!       Purpose: get the protocol flavor of the file
!       * Datatype: int4 (UPS_IO_PROTOCOL_enum)
!       * Get: yes
!       * Set: no
!       * Get the underlying protocol that will be
!       * used to read from or write to the file.
!       * Note that this protocol will be the same for
!       * the file id and all its children group ids.
      parameter ( UPS_IO_FILE_PROTOCOL            = 12 )
!       Purpose: get the open method for the file
!       * Datatype: int4 (UPS_IO_OPEN_METHOD_enum)
!       * Get: yes
!       * Set: no
!       * Get how the file associated with the
!       * location id was opened.
      parameter ( UPS_IO_FILE_OPEN_METHOD         = 13 )
!
!


!
!------------------------------------------------------------
!     Block from c enum: [UPS_IO_OPEN_METHOD_enum]
!------------------------------------------------------------
!       ------------
!       open methods
!       ------------
!
!
!       Create new one
!       * Files:  overwrite anything there
!       * Groups: open or create if needed
      parameter ( UPS_IO_OPEN_CREATE              = 0 )
!       Open existing one for read only
!       * Files|Groups: error if not already there
!       * Files:  read only access
!       * Groups: read/write access -
!       *         same as UPS_IO_OPEN_READ_WRITE
      parameter ( UPS_IO_OPEN_READ                = 1 )
!       Open existing one for read and write
!       * Files|Groups: error if not already there
!       * Files:  read only access
!       * Groups: read/write access -
!       *         same as UPS_IO_OPEN_READ
      parameter ( UPS_IO_OPEN_READ_WRITE          = 2 )
!
!


!
!------------------------------------------------------------
!     Block from c enum: [UPS_IO_PROTOCOL_enum]
!------------------------------------------------------------
!       -------------------------------------------------------------
!       Underlying IO protocol options
!       Use UPS_AA_Opt_get/UPS_AA_Opt_set with UPS_IO_OPT_PROTOCOL
!       You may set the following as environmental variables as well:
!         (in mutually exclusive groups)
!           - setenv UPS_IO_PROTOCOL_HDF
!             setenv UPS_IO_PROTOCOL_UDM
!       Remember that all environment variables must be propagated to
!       all the processes.
!       ---------------------------------------------------------------
!
!
!       (default) IO calls go to HDF
      parameter ( UPS_IO_PROTOCOL_HDF             = 0 )
!       IO calls go to MPI - not implemented
      parameter ( UPS_IO_PROTOCOL_MPI             = 1 )
!       IO calls go to UDM - not implemented
      parameter ( UPS_IO_PROTOCOL_UDM             = 2 )
!       IO calls go to Libsheaf - not implemented
      parameter ( UPS_IO_PROTOCOL_LIBSHEAF        = 3 )
!       personally typed by me - not implemented
!       as I only type at about 20 words/minute
      parameter ( UPS_IO_PROTOCOL_TYPEWRITER      = 4 )
!       Unknown - cannot be fathomed by UPS
      parameter ( UPS_IO_PROTOCOL_UNKNOWN         = 5 )
!
!



!
!------------------------------------------------------------
!     Block from c enum: [UPS_MS_INFO_ITEM_enum]
!------------------------------------------------------------
!       ------------------------------------------------------------------
!       Values for UPS_MS_Info_(get|set) for an info_id
!       * info_ids are obtained from UPS_MS_Info_create
!       * Form:
!       *   Purpose: simple description
!       *   Datatype: datatype of item
!       *   Get: If you can get this item
!       *   Set: If you can set this item
!       *   More description
!       *
!       ------------------------------------------------------------------
!
!
!       --------------------------------
!       Dataset Distribution Information
!       --------------------------------
!       Purpose: name of the file
!       * Datatype: char* array size
!       *   UPS_MS_INFO_FILE_NAME_L
!       * Get: yes
!       * Set: yes
!       * The name of the io file.
      parameter ( UPS_MS_INFO_FILE_NAME           = 0 )
```

```
!       Purpose: length - no null-terminator
!       * Datatype: int8
!       * Get: yes
!       * Set: no
      parameter ( UPS_MS_INFO_FILE_NAME_L         = 1 )
!       Purpose: full path to top level group
!       * Datatype: char* array size
!       *   UPS_MS_INFO_MESH_LOC_L
!       * Get: yes
!       * Set: yes
!       * This will be "." (default) if writing to the
!       * root group.
      parameter ( UPS_MS_INFO_MESH_LOC            = 2 )
!       Purpose: length - no null-terminator
!       * Datatype: int8
!       * Get: yes
!       * Set: no
      parameter ( UPS_MS_INFO_MESH_LOC_L          = 3 )
!       Purpose: description line 1
!       * Datatype: char* array size
!       *   UPS_MS_INFO_MESH_DESC_1
!       * Get: yes
!       * Set: yes
!       * Brief description - line 1
      parameter ( UPS_MS_INFO_MESH_DESC1          = 4 )
!       Purpose: length - no null-terminator
!       * Datatype: int8
!       * Get: yes
!       * Set: no
      parameter ( UPS_MS_INFO_MESH_DESC1_L        = 5 )
!       Purpose: description line 2
!       * Datatype: char* array size
!       *   UPS_MS_INFO_MESH_DESC_2
!       * Get: yes
!       * Set: yes
!       * Brief description - line 2
      parameter ( UPS_MS_INFO_MESH_DESC2          = 6 )
!       Purpose: length - no null-terminator
!       * Datatype: int8
!       * Get: yes
!       * Set: no
      parameter ( UPS_MS_INFO_MESH_DESC2_L        = 7 )
!       Purpose: The type of node id scheme
!       * Datatype: UPS_MS_NODEID_SCHEME_enum
!       * Get: yes
!       * Set: yes
      parameter ( UPS_MS_INFO_NODEID_SCHEME       = 8 )
!       Purpose: current solution time
!       * Datatype: real8
!       * Get: yes
!       * Set: yes
```

```
      parameter ( UPS_MS_INFO_SOLUTION_TIME       = 9 )
!
!


!
!------------------------------------------------------------
!     Block from c enum: [UPS_MS_NODEID_SCHEME_enum]
!------------------------------------------------------------
!       ------------------------
!       types of node id schemes
!       ------------------------
!
!
      parameter ( UPS_MS_NODEID_UNKNOWN           = 0 )
      parameter ( UPS_MS_NODEID_OFF               = 1 )
      parameter ( UPS_MS_NODEID_GIVEN             = 2 )
      parameter ( UPS_MS_NODEID_ASSIGN            = 3 )
      parameter ( UPS_MS_NODEID_IGNORE            = 4 )
!
!


!
!------------------------------------------------------------
!     Block from c enum: [UPS_UT_Alloc_enum]
!------------------------------------------------------------
!       --------------------------
!       types of memory allocation
!       --------------------------
!
!
      parameter ( UPS_UT_ALLOC_CALLOC             = 0 )
      parameter ( UPS_UT_ALLOC_MALLOC             = 1 )
      parameter ( UPS_UT_ALLOC_REALLOC            = 2 )
!
!


!
!------------------------------------------------------------
!     Block from c enum: [UPS_UT_CHECKSUM_TYPE_enum]
!------------------------------------------------------------
!       ------------------
!       types of checksums
!       ------------------
!
!
!       Cyclic Redundancy Check
      parameter ( UPS_UT_CHECKSUM_CRC             = 0 )
!
!
```

```
!
!------------------------------------------------------------
!     Block from c enum: [UPS_UT_Convert_enum]
!------------------------------------------------------------
!       ------------------------------------------------
!       possible conversion in the UPS_UT_Convert routine
!       ------------------------------------------------
!
!
!       --------------------
!       First one...not used
!       --------------------
      parameter ( UPS_UT_CONVERT_ENUM_FIRST       = 0 )
!       -------------------------------------------------
!       ups value to communication protocol (eg mpi) value
!       -------------------------------------------------
      parameter ( UPS_UT_TO_PROTOCOL_COMM         = 1 )
      parameter ( UPS_UT_TO_PROTOCOL_COMM_COMPARE = 2 )
      parameter ( UPS_UT_TO_PROTOCOL_DATATYPE     = 3 )
      parameter ( UPS_UT_TO_PROTOCOL_OP           = 4 )
      parameter ( UPS_UT_TO_PROTOCOL_PROCESS_ID   = 5 )
      parameter ( UPS_UT_TO_PROTOCOL_TAG_ID       = 6 )
!       -------------------------------------------------
!       communication protocol (eg mpi) value to ups value
!       -------------------------------------------------
      parameter ( UPS_UT_TO_UPS_COMM              = 7 )
      parameter ( UPS_UT_TO_UPS_COMM_COMPARE      = 8 )
      parameter ( UPS_UT_TO_UPS_DATATYPE          = 9 )
      parameter ( UPS_UT_TO_UPS_OP                = 10 )
      parameter ( UPS_UT_TO_UPS_PROCESS_ID        = 11 )
      parameter ( UPS_UT_TO_UPS_TAG_ID            = 12 )
!       ------------------------------------------------------------
!       ups to ups location datatype (eg for UPS_AA_MAXLOC operation)
!       ------------------------------------------------------------
      parameter ( UPS_UT_UPS_LOC_RED_DATATYPE     = 13 )
!       -----------------------
!       HDF conversion routines
!       -----------------------
      parameter ( UPS_UT_HDF_DATATYPE_FROM        = 14 )
      parameter ( UPS_UT_HDF_DATATYPE_TO          = 15 )
!       -----------------------
!       UDM conversion routines
!       -----------------------
      parameter ( UPS_UT_UDM_DATATYPE_FROM        = 16 )
      parameter ( UPS_UT_UDM_DATATYPE_TO          = 17 )
      parameter ( UPS_UT_UDM_INFOENUM_FROM        = 18 )
      parameter ( UPS_UT_UDM_INFOENUM_TO          = 19 )
      parameter ( UPS_UT_UDM_ITEMENUM_FROM        = 20 )
      parameter ( UPS_UT_UDM_ITEMENUM_TO          = 21 )
      parameter ( UPS_UT_UDM_ITEM_VALUE_FROM      = 22 )
      parameter ( UPS_UT_UDM_ITEM_VALUE_TO        = 23 )
      parameter ( UPS_UT_UDM_OBJTYPE_FROM         = 24 )
      parameter ( UPS_UT_UDM_OBJTYPE_TO           = 25 )
      parameter ( UPS_UT_UDM_OPEN_METHOD_FROM     = 26 )
      parameter ( UPS_UT_UDM_OPEN_METHOD_TO       = 27 )
!       -------------------
!       Last one...not used
!       -------------------
      parameter ( UPS_UT_CONVERT_ENUM_LAST        = 28 )
!
!



!
!------------------------------------------------------------
!     Block from c enum: [UPS_UT_Loc_structure_op_enum]
!------------------------------------------------------------
!       ---------------------------------------------------------
!       wind arrays onto loc structure or unwind struct onto arrays
!       ---------------------------------------------------------
!
!
      parameter ( UPS_UT_WIND_STRUCT              = 1 )
      parameter ( UPS_UT_UNWIND_STRUCT            = 2 )
!
!



!
!------------------------------------------------------------
!     Block from c enum: [UPS_UT_Loc_type_enum]
!------------------------------------------------------------
!       -----------------------------------------------------
!       when winding/unwinding, is the loc a scalar or a vector
!       -----------------------------------------------------
!
!
      parameter ( UPS_UT_SCALAR_LOC               = 1 )
      parameter ( UPS_UT_VECTOR_LOC               = 2 )
      parameter ( UPS_UT_VALUE_LOC                = 3 )
!
!



!
!------------------------------------------------------------
!     Block from c enum: [UPS_UT_Name_or_value_enum]
!------------------------------------------------------------
!       -----------------------
!       if getting name or value
!       -----------------------
!
```

```
!
!       name_or_value: int*
!       value_or_name: char** (NULL if not found)
      parameter ( UPS_UT_GET_NAME                 = 0 )
!       name_or_value: char*
!       value_or_name: int* (0 if not found)
      parameter ( UPS_UT_GET_VALUE                = 1 )
!       name_or_value: not_used
!       value_or_name: int* (number of items in list)
      parameter ( UPS_UT_GET_COUNT                = 2 )
```

C Reference Manual

This reference manual contains a listing of all of the user callable routines. They are listed alphabetically (which means they grouped by component).

See table UPS Packages (section 2 page 6).

General (section C.2, page 91)

Communication (section C.3, page 102)

Data parallel (section C.4, page 126)

Datatype (section C.5, page 139)

Error handling (section C.6, page 140)

Gather/scatter (section C.7, page 146)

Utilities (section C.9, page 230)

C.1 Organization of reference pages

Reference pages are organized as follows:

- **Name**

  The name of the function is listed.

- **Purpose**

  A brief description of its functionality.

- **Usage**

  An example of a call for the supported programming languages.

- **Arguments**

  A description of each argument.

  - **Argument Name**

    Name of the argument.

  - **Intent**

    in, out, or inout (both).

  - **Variable Type**

    Given the programming language, define the variable type.

    * `(na)`

      Argument does not exist in this language.

    * `(optional)`
      Argument is optional in this language. Additionally, a colon followed by a word may appear here. That word specifies how the optional variable is handled. '`(optional:UPS_DP_COMBINERM)`' would mean that if the variable is supplied, a call to the masked function is called under the hood.
    * `{[dim spec][:count spec[:datatype spec]]}`
      The `dim spec` specifies the allowable array dimensions. '`{0}`' means only a scalar value is allowed. '`{0-2,7,9}`' means Scalar, 1d, 2d, 7d, and 9d arrays are allowed. Contact `ups-team@lanl.gov` if you require additional dimensions.
      If the `dim spec` contains a variable, that means the shape must match the listed variable.
      The `count spec` describes the count of the variable. For example, '`{1:setup}`' means that 1d arrays are accepted and the count was defined during a setup call.
      The `datatype spec` describes the datatype of the variable. For example, '`{1:setup:setup}`' means that 1d arrays are accepted and the count and datatype were defined during a setup call. '`{1::setup}`' means that only the datatype was specified during a setup call.
    * `user_choice|UPS_KIND_[variable type]`
      For `user_choice`, the user is allowed to give different variable types.
      For example, a type of:
      '`UPS_KIND_REAL8` {0-2}'
      would mean the user could supply a
      - `REAL(KIND=UPS_KIND_REAL8)`
      - `REAL(KIND=UPS_KIND_REAL8), DIMENSION(:)`
      - `REAL(KIND=UPS_KIND_REAL8), DIMENSION(:,:)`
  - **Description**
    A description of the variable

- **Return Values**

  Most routines will return `UPS_OK` if successful.

- **Discussion**

  If necessary, a more detailed description of the routine.

- **Examples**

  If necessary, some examples of use.

- **See Also**

  Other relevant routines.

C.2 General

See the packages section (section 6.1, page 22 for a general description of this package.

This section contains an alphabetical listing of the general routines available in UPS. These routines may be called in conjunction with any UPS component.

UPS_AA_Abort()

Package

aa

Purpose

UPS_AA_Abort terminates all processes in the UPS communication environment. When an error condition occurs and an abnormal termination is needed, users should call this routine (as opposed to normal termination of UPS_AA_Terminate).

Usage

C	`ierr = UPS_AA_Abort`	`();`
Fortran	`call UPSF_AA_ABORT`	`(ierr)`
Fortran77	`call UPS_AA_ABORT`	`(ierr)`

Arguments

`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

A call to UPS_AA_Abort by any process causes UPS to attempt to terminate all processes. The user should be aware that successful termination of all processes is not always possible by UPS, especially when those processes are loosely coupled, such as a network of workstations. It is the user's responsibility to ensure that the processes have in fact completely terminated.

SeeAlso

`UPS_AA_Init` (page 92)
`UPS_AA_Terminate` (page 100)

UPS_AA_Init()

Package

aa

Purpose

UPS_AA_Init sets up and initializes the UPS environment.

Usage

C	`ierr = UPS_AA_Init`	`(argc, argv);`
Fortran	`call UPSF_AA_INIT`	`(ierr)`
Fortran77	`call UPS_AA_INIT`	`(ierr)`

Arguments

`argc`
Intent: in
C type: int
Fortran type: (na)
Fortran77 type: (na)
The number of character strings in argv.
This value will be passed to underlying communication layers if they need to be initialized (eg passed to MPI_Init).
The value set by the following call will override this value:
UPS_AA_Opt_set(UPS_AA_OPT_MAIN_LANG_ARGC, &argc)

`argv`
Intent: in
C type: char**
Fortran type: (na)
Fortran77 type: (na)
This value will be passed to underlying communication layers if they need to be initialized (eg passed to MPI_Init).
The value set by the following call will override this value:
UPS_AA_Opt_set(UPS_AA_OPT_MAIN_LANG_ARGV, &argv)

`ierr`
Intent: out
C type: (na) int return value
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

UPS_AA_Init initializes each of the UPS components. First, it performs the system tasks required to set up the data channels between all calling processes. Next, it sets up a variety of variables, pointers, and parameters needed by the UPS runtime environment.

Examples

See Getting Started (section 3 page 7) for a basic example.

Most likely users will call the Fortran interface if initializing a Fortran program and the C interface if initializing a C/C++ program. Using the C interface to initialize a Fortran program (or vise versa) can be a problem due to a retriction of MPI requiring the correct language interface of MPI_Init to be called. One can get around this problem by doing the following:

- Call the corresponding language (Fortran or C) UPS_AA_Init call

  By default, UPS assumes the language of the main program by the interface that is called for UPS_AA_Init.

- Call MPI_Init manually before calling UPS_AA_Init
  (and call MPI_Finalize after UPS_AA_Terminate).

- C main OR Fortran main calling Fortran interface example

  This example shows code fragments of a C and a Fortran main program calling a Fortran initialization routine.

  – C main snippet

```
#include "ups.h"
int main( int argc, char **argv ) {
  ...
  // my_fortran_init assumes a Fortran program so must set
  // options defining main language and command line args.
  // Set language and command line args before initializing
  // UPS.
  UPS_AA_MAIN_LANG_enum main_lang = UPS_AA_MAIN_LANG_C;
  UPS_AA_Opt_set( UPS_AA_MAIN_LANG, &main_lang );
  UPS_AA_Opt_set( UPS_AA_OPT_MAIN_LANG_ARGC, &argc );
  UPS_AA_Opt_set( UPS_AA_OPT_MAIN_LANG_ARGV, &argv );
  // Call fortran routine that initializes UPS.
  my_fortran_init_();
  ...
}
```

  – Fortran main snippet

```
program prog()
  USE UPS
  ...
  ! Nothing special needed since my_fortran_init assumes
  ! a Fortran program.
  ! Call Fortran routine that initializes UPS.
  call my_fortran_init()
  ...
end program prog
```

  – Fortran initialization routine my_fortran_init

```
subroutine my_fortran_init()
  USE UPS
  ...
  ! If called from C main, necessary options have already
  ! been set so can just call UPSF_AA_INIT
  integer(KIND=UPS_KIND_INT4) :: ierr
  call UPSF_AA_INIT( ierr )
  ...
end subroutine my_fortran_init
```

- C main OR Fortran main calling C interface example

  This example is a little more complex because in order to maintain separation of Fortran and C internally in UPS, you must call the Fortran UPS initialization interface in my_c_init_ when the main program is fortran.

  – C main snippet

```
int main( int argc, char **argv ) {
  ...
  // my_c_init assumes a C program so you do not need
  // to do anything special besides pass argc and argv
  // to my_c_init_.
  // If you want, you could call UPS_AA_Opt_set as in the
  // above C main to set the language/arguments.
  // In this case, the arguments argc/argv passed to
  // my_c_init_ will be ignored.
  // Call c routine that initializes UPS.
  my_c_init_(argc, argv);
  ...
}
```

  – Fortran main snippet

```
program prog()
  USE UPS
  ...
  ! my_c_init assumes a C program so need to set the language
  ! to Fortran (command line arguments not used so pass any
  ! values).  Set this before initializing UPS.
  integer(KIND=UPS_KIND_INT4) :: ierr, main_lang
  main_lang = UPS_AA_MAIN_LANG_F
  call UPSF_AA_OPT_SET( UPS_AA_MAIN_LANG, main_lang, ierr )
  ! call C routine that initializes UPS.
  call my_c_init(0,0)
end program prog
```

  – C initialization routine my_c_init_

```
#include "ups.h"
void my_c_init_(int argc, char **argv) {
  ...
  // Tricky part - a Fortran main must still call UPS Fortran
  // initialization (in order to keep Fortran and C separated
  // internally in UPS)
  int ierr;
  UPS_AA_MAIN_LANG_enum main_lang;
  UPS_AA_Opt_get( UPS_AA_MAIN_LANG, &main_lang );
  // default is C
  if( main_lang == UPS_AA_MAIN_LANG_UNSET ||
      main_lang == UPS_AA_MAIN_LANG_C ) {
    UPS_AA_Init( argc, argv );
  }
  else {
    UPS_AA_INIT(ierr)
  }
  ...
}
```

SeeAlso

`UPS_AA_Abort` (page 91)
`UPS_AA_Terminate` (page 100)

UPS_AA_Io_pe_get()

Package

aa

Purpose

Get the current io_pe. Various functions use the io_pe for communication.

Usage

C	`ierr = UPS_AA_Io_pe_get`	`(io_pe);`
Fortran	`call UPSF_AA_IO_PE_GET`	`(io_pe, ierr)`
Fortran77	`call UPS_AA_IO_PE_GET`	`(io_pe, ierr)`

Arguments

`io_pe`

Intent:	out
C type:	int*
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The current io_pe.

`ierr`

Intent:	out
C type:	(na) int return value
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

SeeAlso

`UPS_AA_Io_pe_get` (page 95)
`UPS_AA_Io_pe_set` (page 96)
`UPS_GS_Collate` (page 146)
`UPS_GS_Distribute` (page 147)
`UPS_IO_File_open` (page 198)

UPS_AA_Io_pe_set()

Package

aa

Purpose

Set the current io_pe. Various functions use the io_pe for communication.

Usage

C	`ierr = UPS_AA_Io_pe_set`	`(io_pe);`
Fortran	`call UPSF_AA_IO_PE_SET`	`(io_pe, ierr)`
Fortran77	`call UPS_AA_IO_PE_SET`	`(io_pe, ierr)`

Arguments

`io_pe`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The current io_pe.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

SeeAlso

`UPS_AA_Io_pe_get` (page 95)
`UPS_AA_Io_pe_set` (page 96)
`UPS_GS_Collate` (page 146)
`UPS_GS_Distribute` (page 147)
`UPS_IO_File_open` (page 198)

UPS_AA_Opt_get()

Package

aa

Purpose

Get an optimization parameter.

Usage

C	`ierr = UPS_AA_Opt_get`	`(opt_type, opt_value);`
Fortran	`call UPSF_AA_OPT_GET`	`(opt_type, opt_value, ierr)`
Fortran77	`call UPS_AA_OPT_GET`	`(opt_type, opt_value, ierr)`

Arguments

`opt_type`

Intent:	in
C type:	UPS_AA_OPT_TYPE_enum
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The optimization type.
Please see UPS_AA_OPT_TYPE_enum
(section B page 51)
for a listing of the possible option types.

`opt_value`

Intent:	out
C type:	void*
Fortran type:	user_choice {0:opt_type:opt_type}
Fortran77 type:	user_choice {0:opt_type:opt_type}

The optimization value.

`ierr`

Intent:	out
C type:	(na) int return value
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

error return value

ReturnValues

Returns UPS_OK if successful

Examples

See see UPS_AA_Opt_get (section C.2 page 97) for more information.

SeeAlso

`UPS_AA_Opt_get` (page 97)
`UPS_AA_Opt_set` (page 98)

UPS_AA_Opt_set()

Package

aa

Purpose

Set an optimization parameter.

Usage

C	`ierr = UPS_AA_Opt_set`	`(opt_type, opt_value);`
Fortran	`call UPSF_AA_OPT_SET`	`(opt_type, opt_value, ierr)`
Fortran77	`call UPS_AA_OPT_SET`	`(opt_type, opt_value, ierr)`

Arguments

`opt_type`

Intent:	in
C type:	UPS_AA_OPT_TYPE_enum
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The optimization type.
Please see UPS_AA_OPT_TYPE_enum
(section B page 51)
for a listing of the possible option types.

`opt_value`

Intent:	in
C type:	const void*
Fortran type:	user_choice {0-1:opt_type:opt_type}
Fortran77 type:	user_choice {0-1:opt_type:opt_type}

The optimization value.

`ierr`

Intent:	out
C type:	(na) int return value
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

error return value

ReturnValues

Returns UPS_OK if successful

Examples

See see UPS_AA_Opt_get (section C.2 page 97) for more information.

SeeAlso

`UPS_AA_Opt_get` (page 97)
`UPS_AA_Opt_set` (page 98)

UPS_AA_Statistics()

Package

aa

Purpose

UPS_AA_Statistics defines the level of statistics gathering will be applied to UPS. By default, statistics is turned on, but may be turned on/off via this routine or may be turned off by setting the UPS_STATISTICS_OFF environment variable.

The output file, ups_log.txt, is written by process 0 upon a call to UPS_AA_Terminate.

This call only affects the calling process. Different settings are allowable for different processes.

Usage

C	`ierr = UPS_AA_Statistics`	`(statistics_flag);`
Fortran	`call UPSF_AA_STATISTICS`	`(statistics_flag, ierr)`
Fortran77	`call UPS_AA_STATISTICS`	`(statistics_flag, ierr)`

Arguments

`statistics_flag`

Intent:	in
C type:	UPS_AA_Statistics_enum
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The level of statistics to be used.
See Statistics Flags
(section B page 58)
for the statistics options.

`ierr`

Intent:	out
C type:	(na) int return value
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

Most UPS functions take enough time to complete so that having statistics turned on will not degrade performance much. The statistics are meant to provide information on how the application is using UPS. This routine (and its ability to turn off statistics) is provided if you do not wish to know this information and/or the loss of some performance is an issue.

The UPS code location is parameterized by UPS_AA_Code_location_enum (section B page 39). The name/value pairs for the locations are printed in the output text file as well.

Statistics about the terminate routines is not gathered because information must be processed before packages are terminated. In general, the terminate routines merely free allocated memory and therefore do not take up much time.

UPS provides a simple perl script to visualize this data. This script is located in:

```
<UPS installation directory>/script/ups_aa_statistics_plot.pl.
```

To use the script, simply type:

```
ups_aa_statistics_plot.pl ups_log.txt
```

This will create ups_log.ps (a postscript file containing graphs produced by gnuplot). For a description or additional options, type in the name of the script:

```
ups_aa_statistics_plot.pl
```

Examples

Fortran:

```
call UPSF_AA_INIT( ierr )
if ( ierr  /= UPS_OK  ) then
  write (*,*) '**Error** UPS_AA_INIT failed'
  stop
endif

! by default, statistics is on.  Here, it is turned off.
call UPSF_AA_STATISTICS( UPS_STATISTICS_OFF, ierr )
if ( ierr  /= UPS_OK  ) then
   write (*,*) '**Error** UPS_AA_STATISTICS failed'
   stop
endif

...code...

call UPSF_AA_TERMINATE( ierr )
if ( ierr  /= UPS_OK  ) then
  write (*,*) '**Error** UPS_AA_TERMINATE failed'
  stop
endif
```

Notes

Please see UPS_AA_Statistics_enum (section B page 58) for environment variables that affect this call.

UPS_AA_Terminate()

Package

aa

Purpose

UPS_AA_Terminate shuts down everything under the control of UPS. For example, it terminates the communication environment (if the comm environment was initialized by UPS) and frees the memory UPS may have allocated. This routine should be used for normal completion of a program (as opposed to UPS_AA_Abort).

Usage

C	`ierr = UPS_AA_Terminate`	`();`
Fortran	`call UPSF_AA_TERMINATE`	`(ierr)`
Fortran77	`call UPS_AA_TERMINATE`	`(ierr)`

Arguments

`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

A call to UPS_AA_Terminate takes the calling process out of the UPS environment. No further interaction with UPS components is possible.

Examples

See Getting Started (section 3 page 7) for an example.

SeeAlso

`UPS_AA_Abort` (page 91)
`UPS_AA_Init` (page 92)

C.3 Communication

See the packages section (section 6.2, page 23 for a general description of this package.

This section contains an alphabetical listing of the communication routines available in UPS.

UPS_CM_Allgather()

Package

cm

Purpose

Perform functionality of MPI_Allgather where count/datatype is the same.

Usage

C	`ierr = UPS_CM_Allgather`	`(sendbuf, recvbuf, count, datatype);`
Fortran	`call UPSF_CM_ALLGATHER`	`(sendbuf, recvbuf, count, datatype, ierr)`
Fortran77	`call UPS_CM_ALLGATHER`	`(sendbuf, recvbuf, count, datatype, ierr)`

Arguments

`sendbuf`

Intent:	in
C type:	const void*
Fortran type:	user_choice {0-1}
Fortran77 type:	user_choice {0-1}

The starting address of the send buffer.

`recvbuf`

Intent:	out
C type:	void*
Fortran type:	user_choice {0-1}
Fortran77 type:	user_choice {0-1}

The starting address of the recv buffer.

`count`

Intent:	in
C type:	int
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The number of elements to be gathered from each process.

`datatype`

Intent:	in
C type:	UPS_DT_Datatype_enum
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Please see UPS_DT_Datatype_enum
(section B page 63)
for a listing of the possible datatypes.

`ierr`

Intent:	out
C type:	(na) int return value

Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

UPS_CM_Allreduce()

Package

cm

Purpose

Performs functionality of MPI_Allreduce.

Usage

C	`ierr = UPS_CM_Allreduce`	`(x, y, count, datatype, reduce_op);`
Fortran	`call UPSF_CM_ALLREDUCE`	`(x, y, count, datatype, reduce_op, ierr)`
Fortran77	`call UPS_CM_ALLREDUCE`	`(x, y, count, datatype, reduce_op, ierr)`

Arguments

`x`
Intent: in
C type: const void*
Fortran type: user_choice {0-4}
Fortran77 type: user_choice {0-4}
The starting address of the local values.

`y`
Intent: out
C type: void*
Fortran type: user_choice {x}
Fortran77 type: user_choice {x}
The starting address of the result of the reduction.
Note that this cannot be the same memory location as `x`.

`count`
Intent: in
C type: int
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The number of local values. That is, the length of `x`
in terms of the input datatype.

`datatype`
Intent: in
C type: UPS_DT_Datatype_enum
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The type of the local data.

	Please see UPS_DT_Datatype_enum (section B page 63) for a listing of the possible datatypes.	
`reduce_op`	*Intent*:	in
	C type:	UPS_AA_Operation_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The operation to perform. Please see UPS_AA_Operation_enum (section B page 56) for a listing of the possible operations.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

SeeAlso

`UPS_CM_Allreduce` (page 103)
`UPS_CM_Reduce` (page 115)

UPS_CM_Barrier()

Package

cm

Purpose

Performs functionality of MPI_Barrier.

Usage

C	`ierr = UPS_CM_Barrier`	`();`
Fortran	`call UPSF_CM_BARRIER`	`(ierr)`
Fortran77	`call UPS_CM_BARRIER`	`(ierr)`

Arguments

`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

SeeAlso

`UPS_CM_Barrier` (page 104)
`UPS_CM_Barrier_idle` (page 105)

UPS_CM_Barrier_idle()

Package

cm

Purpose

UPS_CM_Barrier_idle performs a the same functionality as UPS_CM_Barrier.

However, on many systems UPS_CM_Barrier does a busy-wait and thus does not allow the processor to do other tasks. UPS_CM_Barrier_idle attempts to perform a non-busy-wait which allows the processor to do other tasks (eg work on other threads). This is made possible by reducing the polling frequency (and thus likely extending the time a process is in UPS_CM_Barrier_idle).

Usage

C	`ierr = UPS_CM_Barrier_idle`	`();`
Fortran	`call UPSF_CM_BARRIER_IDLE`	`(ierr)`
Fortran77	`call UPS_CM_BARRIER_IDLE`	`(ierr)`

Arguments

`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

There may be significant time overhead when returning all processes from the ”idle” state back to the ”normal” state.

DO NOT USE THIS CALL IF YOU NEED TO MINIMIZE TIME IN A BARRIER.

This UPS_CM_Barrier_idle is meant to be used in mixed MPI/threaded code in which the MPI parallelization is coarse grained and the extra non-busy-wait time spent in UPS_CM_Barrier_idle is outweighed by that processor being able to work on its threads.

NOTE:

Current implementation uses a sleep system call `usleep`. Empirical evidence on ASCI Bluemountain dictates us to sleep on the order of 1/10 of a second.

On ASCI Red), fine grained system sleep does not exist. In this case, UPS uses the `sleep` system call - which has a minimum sleep time of 1 second.

SeeAlso

`UPS_CM_Barrier` (page 104)

`UPS_CM_Barrier_idle` (page 105)

UPS_CM_Bcast()

Package

cm

Purpose

Performs functionality of MPI_Bcast. Data chunking has been added to allow large buffers in which most MPI implementations would fail.

Usage

C	`ierr = UPS_CM_Bcast`	`(buf, count, datatype, root_pe);`
Fortran	`call UPSF_CM_BCAST`	`(buf, count, datatype, root_pe, ierr)`
Fortran77	`call UPS_CM_BCAST`	`(buf, count, datatype, root_pe, ierr)`

Arguments

`buf`
Intent: inout
C type: void*
Fortran type: user_choice {0-1}
Fortran77 type: user_choice {0-1}
On the root process, this input data is sent to all other processes. On the receiving processes, the data is placed here.

`count`
Intent: in
C type: int
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The number of local values. That is, the length of x in terms of the input datatype.

`datatype`
Intent: in
C type: UPS_DT_Datatype_enum
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Please see UPS_DT_Datatype_enum (section B page 63) for a listing of the possible datatypes.

`root_pe`
Intent: in
C type: int
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The process sending the data.

`ierr`
Intent: out
C type: (na) int return value
Fortran type: UPS_KIND_INT4 {0}

Fortran77 type: UPS_KIND_INT4 {0}
Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful

Discussion

UPS_CM_Bcast is designed to handle buffers of unlimited size. Such a feature is not always possible given direct use of the underlying communication protocol. UPS avoids this limitation by dividing the input buffer into segments which can be handled; this necessarily implies that the latency of the underlying communication protocol will be incurred by a multiple of the number of segments UPS forms.

UPS_CM_Context_free()

Package

cm

Purpose

Free UPS data associated with the context set by UPS_CM_Set_context.

Usage

C	`ierr = UPS_CM_Context_free`	`(context);`
Fortran	`call UPSF_CM_CONTEXT_FREE`	`(context, ierr)`
Fortran77	`call UPS_CM_CONTEXT_FREE`	`(context, ierr)`

Arguments

`context`
Intent: in
C type: const void*
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Actual C type: UPS_DT_PROTOCOL_COMM*
Users can call `UPS_DT_Sizeof` (section C.5 page 139) if they need to allocate space explicitly.
The process context, i.e. the identifier for the relevant group of processes.

`ierr`
Intent: out
C type: (na) int return value
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful

Discussion

This call only frees the additional data created by UPS for the context originally created by the user. In other words, the user must still call the "free" call by whatever communication protocol they are using (eg MPI_Comm_free) to free the context created and originally sent to UPS_CM_Set_context.

Upon UPS termination, this routine is automatically called for all contexts.

You may not free the context that is currently being used. Switch to a new context then free the old ones.

SeeAlso

`UPS_CM_Context_free` (page 107)
`UPS_CM_Get_context` (page 108)
`UPS_CM_Set_context` (page 118)

UPS_CM_Get_context()

Package

cm

Purpose

UPS_CM_Get_context returns the process context that is currently being used for communication.

Usage

C	`ierr = UPS_CM_Get_context`	`(context);`
Fortran	`call UPSF_CM_GET_CONTEXT`	`(context, ierr)`
Fortran77	`call UPS_CM_GET_CONTEXT`	`(context, ierr)`

Arguments

`context`
Intent: out
C type: void*
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Actual C type: UPS_DT_PROTOCOL_COMM*
The current context being used.
For the C interface, you are sending in a pointer
to a protocol context (eg a pointer to a MPI_Comm).
If allocating your own space, you need to ensure enough
space is present. You can call
`UPS_DT_Sizeof` (section C.5
page 139) with the argument
UPS_DT_PROTOCOL_COMM.

`ierr`
Intent: out
C type: (na) int return value
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

Note to MPI users. UPS operates within a duplicated version of the user input communicator. That is, the user communicator is copied into a separate process context through the use of MPI_Comm_dup. This allows UPS to operate on the same processes without interference between the user application, UPS, and other linked libraries. However, when the user asks for the current context via `UPS_CM_Get_context`, the version the user supplied to UPS through `UPS_CM_Set_context` is returned.

Examples

In the following example, a process wishes to set the process context to a new one, but reset the old one before returning.

Fortran:

```
call UPSF_CM_GET_CONTEXT( OLD_CONTEXT, ierr )
global_error = global_error + ierr
call UPSF_CM_SET_CONTEXT( NEW_CONTEXT, ierr )
global_error = global_error + ierr
! Do work
call UPSF_CM_SET_CONTEXT( OLD_CONTEXT, ierr )
global_error = global_error + ierr
call UPSF_CM_CONTEXT_FREE( NEW_CONTEXT, ierr )
return
```

Example for C users in allocating space for context:

C:

```
int sizeof_comm;
void *comm_current;
ierr += UPS_DT_sizeof( UPS_DT_PROTOCOL_COMM, &sizeof_comm );
comm_current = malloc( sizeof_comm );
ierr += UPS_CM_Get_context( comm_current );
```

SeeAlso

`UPS_CM_Context_free` (page 107)
`UPS_CM_Get_context` (page 108)
`UPS_CM_Set_context` (page 118)
`UPS_CM_P_group_item` (page 111)
`UPS_DT_Sizeof` (page 139)

UPS_CM_Get_numpes()

Package

cm

Purpose

Performs functionality of MPI_Comm_size.

Usage

C	`ierr = UPS_CM_Get_numpes`	`(numpes);`
Fortran	`call UPSF_CM_GET_NUMPES`	`(numpes, ierr)`
Fortran77	`call UPS_CM_GET_NUMPES`	`(numpes, ierr)`

Arguments

`numpes`	*Intent*:	out
	C type:	int*
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The number of processes in the current communicator.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

SeeAlso

`UPS_CM_P_group_item` (page 111)
`UPS_CM_Get_numpes` (page 110)
`UPS_CM_Get_penum` (page 110)

UPS_CM_Get_penum()

Package

cm

Purpose

Performs functionality of MPI_Comm_rank.

Usage

C	`ierr = UPS_CM_Get_penum`	`(penum);`
Fortran	`call UPSF_CM_GET_PENUM`	`(penum, ierr)`
Fortran77	`call UPS_CM_GET_PENUM`	`(penum, ierr)`

Arguments

`penum` *Intent*: out
C type: int*
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The integer identifier of the calling process as assigned in the input communicator.
This will be a number between 0 and number of processes -1 .

`ierr` *Intent*: out
C type: (na) int return value
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

SeeAlso

`UPS_CM_P_group_item` (page 111)
`UPS_CM_Get_penum` (page 110)
`UPS_CM_Get_numpes` (page 110)

UPS_CM_P_group_item()

Package

cm

Purpose

UPS_CM_P_group_item is a generalization of the UPS_CM_Get_<item> routines. This routine returns labeling information given the context, process group, and the item desired.

See the examples below for the type of information you can get.

Usage

C
```
ierr = UPS_CM_P_group_item  (context, p_group, p_group_item,
                             item);
```
Fortran
```
call UPSF_CM_P_GROUP_ITEM   (context, p_group, p_group_item,
                             item, ierr)
```
Fortran77
```
call UPS_CM_P_GROUP_ITEM    (context, p_group, p_group_item,
                             item, ierr)
```

Arguments

`context` *Intent*: in
C type: const void*
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Actual C type: UPS_DT_PROTOCOL_COMM*
The context you are interested in.
This must be a context either supplied to UPS_CM_Set_context (section C.3 page 118)

or the global context (eg MPI_COMM_WORLD).
To get the current context, one may use
UPS_CM_Get_context (section C.3 page 108)
For the C interface, if NULL is passed as this argument,
the current context obtained from UPS_CM_Get_context
will be used.

`p_group`	*Intent*:	in
	C type:	UPS_CM_P_group_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}

The process group you are interested in.
Please see UPS_CM_P_group_enum (section B page 60)
for a listing/explanation of different process groups.

`p_group_item`	*Intent*:	in
	C type:	UPS_CM_P_group_item_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}

The item wrt the context/process group to get.
Please see UPS_CM_P_group_item_enum (section B page 61)
for a listing/explanation of different process groups
items.

`item`	*Intent*:	out
	C type:	void*
	Fortran type:	UPS_KIND_INT4 {0,1:p_group_item}
	Fortran77 type:	UPS_KIND_INT4 {0,1:p_group_item}

The value of p_group_item wrt the context/process group.
The C type depends upon p_group_item.
The size of this array will vary depending on the
value of p_group_item.
Please see UPS_CM_P_group_item_enum (section B page 61)
for a listing/explanation of different process groups
items.

`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful

Discussion

Different architectures might have effectively different process groups. For example, on ASCI Blue Mountain, there is the concept of machine groups (groups of 8 128-processor smp's). On ASCI Red Sandia, there is no such concept so UPS_CM_P_GROUP_MACHINE_GROUP is set to

UPS_CM_P_GROUP_ALL (effectively making 1 machine group).

Examples

Example calls:
(The current context can be obtained by calling UPS_CM_GET_CONTEXT)

- process number (UPS_CM_Get_penum(&penum))

```
UPS_CM_P_group_item( <current context>,
                     UPS_CM_P_GROUP_ALL,
                     UPS_CM_P_GROUP_ITEM_PENUM,
                     &penum )
```

- number of processes (UPS_CM_Get_numpes(&numpes))

```
UPS_CM_P_group_item( <current context>,
                     UPS_CM_P_GROUP_ALL,
                     UPS_CM_P_GROUP_ITEM_NUMPES,
                     &numpes )
```

- getting the process number with respect to the box you are on (useful for computing box ”masters”)

```
UPS_CM_P_group_item( <current context>,
                     UPS_CM_P_GROUP_BOX,
                     UPS_CM_P_GROUP_ITEM_PENUM,
                     &boxpenum )
```

- getting the number of processes on the same box this process is on

```
UPS_CM_P_group_item( <current context>,
                     UPS_CM_P_GROUP_BOX,
                     UPS_CM_P_GROUP_ITEM_NUMPES,
                     &boxnumpes )
```

- getting the boxnum (for SGI, the box number)

```
UPS_CM_P_group_item( <current context>,
                     UPS_CM_P_GROUP_BOX,
                     UPS_CM_P_GROUP_ITEM_IDNUM,
                     &boxnum )
```

- getting the number of boxes

```
UPS_CM_P_group_item( <current context>,
                     UPS_CM_P_GROUP_BOX,
                     UPS_CM_P_GROUP_ITEM_NUMIDS,
                     &boxnum )
```

- getting the context of all the box masters (boxpenum is 0) (if boxpenum is not 0, the context will be whatever the NULL context for the underlying protocol is: MPI-¿MPI_COMM_NULL)

```
    UPS_CM_P_group_item( <current context>,
                         UPS_CM_P_GROUP_BOX,
                         UPS_CM_P_GROUP_ITEM_MSTR_CNTXT,
                         &box_master_context )
```

- getting the context of the processes on the same box as this process

```
    UPS_CM_P_group_item( <current context>,
                         UPS_CM_P_GROUP_BOX,
                         UPS_CM_P_GROUP_ITEM_ALL_CNTXT,
                         &box_context )
```

Suppose a process is sending info to another process (penum = penum_to) under some context. The sending process wishes to find the penum_to with respect to the global context (which might be different than the current context). Call this penum_to_global.

```
...
ierr  = UPS_OK;
ierr += UPS_CM_Get_context( &current_context );
ierr += UPS_CM_P_group_item( &current_context,
                             UPS_CM_P_GROUP_ALL,
                             UPS_CM_P_GROUP_ITEM_G_PENUMS,
                             &(penums_global[0]) );
if ( ierr != UPS_OK )
  {
     exit( ierr );
  }
penum_to_global = penums_global[penum_to];
...
```

Note, that since we are using the C interface and dealing with the current context, we could skip the call to UPS_CM_Get_context and go straight to:

```
ierr += UPS_CM_P_group_item( NULL,
                             UPS_CM_P_GROUP_ALL,
                             UPS_CM_P_GROUP_ITEM_G_PENUMS,
                             &(penums_global[0]) );
```

When using the Fortran interface, you must supply an actual context.

Now suppose you wish to have 2 sets of communication groups. The first group consists of the processes within a box. The second group consists of the master processes of each box. Those two contexts can be gotten by:

```
...
ierr  = UPS_OK;
ierr += UPS_CM_Get_context( &current_context );
ierr += UPS_CM_P_group_item( current_context,
                             UPS_CM_P_GROUP_BOX,
                             UPS_CM_P_GROUP_ITEM_ALL_CNTXT,
                             &processes_on_my_box_context );
ierr += UPS_CM_P_group_item( current_context,
                             UPS_CM_P_GROUP_BOX,
                             UPS_CM_P_GROUP_ITEM_MSTR_CNTXT,
                             &master_processes_on_each_box_context );
if ( ierr != UPS_OK )
  {
     exit( ierr );
  }
...
```

Note, if a process is not a master pe (penum=0 with respect to the box), its value of master_processes_on_each_box_context will be the null process of whatever underlying communication UPS is using. For example, if this is MPI, the value will be MPI_COMM_NULL.

SeeAlso

`UPS_CM_Get_context` (page 108)
`UPS_CM_Get_penum` (page 110)
`UPS_CM_Get_numpes` (page 110)

UPS_CM_Reduce()

Package

cm

Purpose

Performs the functionality of MPI_Reduce. However, in the UPS flavor, the input and output buffers may be the same. Also, buffer chunking is done to allow large buffers where most MPI implementations would fail.

Usage

C	`ierr = UPS_CM_Reduce`	`(x, y, count, datatype, reduce_op, root_pe);`
Fortran	`call UPSF_CM_REDUCE`	`(x, y, count, datatype, reduce_op, root_pe, ierr)`
Fortran77	`call UPS_CM_REDUCE`	`(x, y, count, datatype, reduce_op, root_pe, ierr)`

Arguments

x   *Intent*: inout
     C type: void*
     Fortran type: user_choice {0-1}
     Fortran77 type: user_choice {0-1}
     The starting address of the local values.

y   *Intent*: inout

	C type:	void*
	Fortran type:	user_choice {0-1}
	Fortran77 type:	user_choice {0-1}
	The starting address of the result of the reduction.	
`count`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The number of local values. That is, the length of `x` and `y` in terms of the input datatype.	
`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The type of the local data. Please see UPS_DT_Datatype_enum (section B page 63) for a listing of the possible datatypes.	
`reduce_op`	*Intent*:	in
	C type:	UPS_AA_Operation_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The operation to perform. Please see UPS_AA_Operation_enum (section B page 56) for a listing of the possible operations.	
`root_pe`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The process returning the result.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

If x and y are different, the contents of y after the reduction are undefined for all processes except the root_pe.

If x and y are the same, the contents of y after the reduction are unchanged for all processes except the root_pe. Performance may differ from the above case since UPS may have to dynamically allocate (and deallocate) intermediate storage buffers.

Please see the discussion about passing identical arguments from Fortran (section 5.5 page 17) for further information.

SeeAlso

`UPS_CM_Allreduce` (page 103)
`UPS_CM_Reduce` (page 115)

UPS_CM_Salltoall()

Package

cm

Purpose

Performs functionality of MPI_Alltoall except optimized for a count of the data is 1 and the data is sparse (mostly 0's).

Usage

C	`ierr = UPS_CM_Salltoall`	`(sendbuf, recvbuf, datatype);`
Fortran	`call UPSF_CM_SALLTOALL`	`(sendbuf, recvbuf, datatype, ierr)`
Fortran77	`call UPS_CM_SALLTOALL`	`(sendbuf, recvbuf, datatype, ierr)`

Arguments

`sendbuf`	*Intent*:	in
	C type:	const void*
	Fortran type:	user_choice {0-1:numpes}
	Fortran77 type:	user_choice {0-1:numpes}
	The data to be sent. Zeros are not sent.	
`recvbuf`	*Intent*:	out
	C type:	void*
	Fortran type:	user_choice {0-1:numpes}
	Fortran77 type:	user_choice {0-1:numpes}
	Index i contains data sent from process i. This must be different than sendbuf.	
`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The datatype of array in elements.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

There is no solid requirement for the degree of sparsity for which the use of this function would always result in performance above that provided by a more general all-to-all broadcast. Such a comparison involves implementation issues for both such functions, the locality of the processes that would be sending and receiving messages, the message traffic currently on the system, and so forth. We can say, however, that for execution on a machine with hundreds or thousands of processors, with each processor sending data to a few nearests neighbors, performance can be significantly better than that of a general all-to-all broadcast.

Consideration was given to building in a decision making capability regarding the degree of sparsity, and using a general all-to-all broadcast when appropriate. However, this would require global communication to determine the status of all participating processes, which would significantly adversely affect performance.

UPS_CM_Set_context()

Package

cm

Purpose

UPS_CM_Set_context sets the context of all communication to the input process context.
All processes in the context must make this call.

Usage

C	`ierr = UPS_CM_Set_context`	`(context);`
Fortran	`call UPSF_CM_SET_CONTEXT`	`(context, ierr)`
Fortran77	`call UPS_CM_SET_CONTEXT`	`(context, ierr)`

Arguments

`context` *Intent*: in
C type: const void*
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Actual C type: UPS_DT_PROTOCOL_COMM*
The process context, i.e. the identifier for the
relevant group of processes. The user is responsible
for the formation of a valid process context within the
context of the underlying communication layer
(e.g. MPI_Comm_split).
If called with NULL, it will be set with the original
context UPS was initialized with (eg MPI_COMM_WORLD).

`ierr` *Intent*: out
C type: (na) int return value
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

`UPS_CM_Set_context` allows the user to work within a specified group of processes, i.e. a process context. This removes the need for the user to pass in the process context for `UPS_CM` functions.

UPS creates information about the context with this call. If this information has already been collected, UPS does not do it again. This data may be freed with a call to UPS_CM_Context_free. This is not necessary as UPS will free its internal context data structures upon termination.

Note to MPI users.

1. UPS operates within a duplicated version of the user input communicator. That is, the user communicator is copied into a separate process context through the use of MPI_Comm_dup. This allows UPS to operate on the same processes without interference between the user application, UPS, and other linked libraries. However, when the user asks for the current communicator via `UPS_CM_Get_context`, the user version of the communicator is returned.

Examples

See the example in UPS_CM_Get_context (section C.3 page 108)

SeeAlso

`UPS_CM_Context_free` (page 107)
`UPS_CM_Get_context` (page 108)
`UPS_CM_Set_context` (page 118)

UPS_CM_Sm_free()

Package

cm

Purpose

Free a shared memory area obtained from UPS_CM_Sm_malloc. See UPS_CM_Sm_malloc (section C.3 page 122) for more information.

Usage

```
C   ierr = UPS_CM_Sm_free   (address);
```

Arguments

`address`	*Intent*:	inout
	C type:	volatile void**
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The address of the shared memory area obtained by UPS_CM_Sm_malloc to be free'd. This address is set to NULL on return.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	(na) no equivalent routine

Fortran77 type: (na) no equivalent routine
Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

This function only frees the shared memory area of this process. If other processes still have the shared memory mapped (eg have not called UPS_CM_Sm_free), they will still be able to access the shared memory area.

Any attempt by this process to access the shared memory area after this call should (if the OS is any good) generate a sigbus error.

Examples

See shared_memory.c (section 6.2.1 page 23) for a detailed example.

SeeAlso

`UPS_CM_Sm_free` (page 119)
`UPS_CM_Sm_malloc` (page 122)
`UPS_CM_Sm_get_item` (page 120)
`UPS_CM_Sm_set_item` (page 124)

UPS_CM_Sm_get_item()

Package

cm

Purpose

Get information about the shared memory routines.

Usage

C	`ierr = UPS_CM_Sm_get_item`	`(item_type, address, item);`
Fortran	`call UPSF_CM_SM_GET_ITEM`	`(item_type, address, item, ierr)`
Fortran77	`call UPS_CM_SM_GET_ITEM`	`(item_type, address, item, ierr)`

Arguments

`item_type` *Intent*: in
C type: UPS_AA_Mem_item_enum
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The type of info requested.
Please see UPS_AA_Mem_item_enum (section B page 49) for a listing/explanation of different items.

`address` *Intent*: inout
C type: volatile void*
Fortran type: UPS_KIND_ADDRESS {0}
Fortran77 type: UPS_KIND_ADDRESS {0}

The address of the sm area gotten by UPS_CM_SM_malloc. For some item_type values, this argument is not used.

`item`	*Intent*:	out
	C type:	void*
	Fortran type:	user_choice {0::item_type}
	Fortran77 type:	user_choice {0::item_type}

The output value of the item_type. See item_type above.

`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

The CM SM routines are just a combination of the POSIX shared memory routines (and some book keeping).

- UPS_CM_Sm_malloc combines shm_open, mmap, close, and shm_unlink
- UPS_CM_Sm_free uses munmap

For shared memory process group that is UPS_CM_P_GROUP_SELF (eg no shared memory possible), UPS_CM_Sm_malloc and UPS_CM_Sm_free simply use malloc() and free() respectively. In this case, user requested size will not be increased to fulfill page size requirements.

When using shared memory, page thrashing becomes an issue. Page thrashing will occur when 2 process are trying to read/write data into the same page in memory. The operating system will spend a lot of time telling the processes, ”Your page is invalid since someone changed information on it - get a new one”. If possible, try to spread the data ”owned” by processes to separate memory areas (unfortunately, this is system dependent). Then, when processes are modifying their data, they do not cause page thrashing with other processes. The page size can be gotten by either the C call getpagesize() or by

```
UPS_CM_Sm_get_item( UPS_CM_SM_ITEM_PAGESIZE, NULL, &pagesize )
```

Be aware as well that on many operating systems, shared memory is not physically assigned until it is first used. The operating system will try to place memory close to the processor that uses it. The result of this is that if you have the master pe initialize all the shared memory area, the OS tries to place all of the memory close to the master pe. This is not what you want if all the processes are going to be heavily modifying their area of the shared memory. In this case, you should have each pe initialize their own area.

The code locations (section B page 58) UPS_CM_LOC_SM_MALLOC and UPS_CM_LOC_SM_FREE make some data available to the user. When statistics are turned on (see UPS_AA_Statistics), the following info fields are recorded at the end of each alloc/free call:

1. Field ID 0: Total size (bytes) of shared mem allocated (+guard bytes)
2. Field ID 1: Total number of shared mem allocations active
3. Field ID 2: Size (bytes) of shared mem just alloc/freed (+guard bytes)

See UPS_UT_Mem_get_item for information about normal memory allocations and environment variables that alter the behavior of the shared memory routines.

Examples

See shared_memory.c (section 6.2.1 page 23) for a detailed example.

SeeAlso

`UPS_CM_Sm_free` (page 119)
`UPS_CM_Sm_malloc` (page 122)
`UPS_CM_Sm_get_item` (page 120)
`UPS_CM_Sm_set_item` (page 124)
`UPS_AA_Statistics` (page 99)

UPS_CM_Sm_malloc()

Package

cm

Purpose

Get a shared memory area for current context and process group that can connect to a shared memory area.

UPS_CM_Sm_get_item (section C.3 page 120) is used to get the value of the p_group. An example is also included in this routine.

See UPS_CM_P_group_enum (section B page 60) for information about p_group's. This value will often be UPS_CM_P_GROUP_BOX for machines with shared memory (it will be UPS_CM_P_GROUP_MACHINE_GROUP for some Irix clusters) and will be UPS_CM_P_GROUP_SELF for machines without shared memory.

If the p_group is UPS_CM_P_GROUP_SELF, then normal malloc is used to get memory for the process.

All pes in this p_group must call this function together.

Usage

```
 C   ierr = UPS_CM_Sm_malloc  (size, address);
```

Arguments

`size`	*Intent*:	inout
	C type:	long long*
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine

On input, this is the desired size in bytes for the shared memory area. Only relevant on the master pe (penum=0 for the process group).

On output, the actual total size available for use is returned. The total size allocated will be increased from the requested size in order to fit the requirement that the total memory length be an

integral number of pages.
So, UPS increases size until:

–¿ total_size = space for guard bytes + size

where total_size is an integral number of pages.

See UPS_UT_Mem_get_item for information about guard bytes (and how to change their size via environment variables).

`address`	*Intent*:	out
	C type:	volatile void**
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine

The address of the address of the memory area.
The actual value of the address returned is not necessarily the same virtual address among the processes that have access to the shared memory area. That is, while pointing to the same physical address, each process may have a different value for the virtual address.
If the value is NULL, allocation failed and an error will be returned.

`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine

Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Examples

See shared_memory.c (section 6.2.1 page 23) for a detailed example.

Notes

Please see the memory management variable and TMPDIR sections of UPS_AA_ENVIRONMENT_VARIABLES_enum (section B page 43) for environment variables that affect this call.

SeeAlso

`UPS_CM_Sm_free` (page 119)
`UPS_CM_Sm_malloc` (page 122)
`UPS_CM_Sm_get_item` (page 120)
`UPS_CM_Sm_set_item` (page 124)
`UPS_CM_Get_context` (page 108)
`UPS_CM_P_group_item` (page 111)
`UPS_CM_Set_context` (page 118)

`UPS_UT_Mem_get_item` (page 242)

UPS_CM_Sm_set_item()

Package

cm

Purpose

Set information about the shared memory routines.

Usage

C	`ierr = UPS_CM_Sm_set_item`	`(item_type, address, item);`
Fortran	`call UPSF_CM_SM_SET_ITEM`	`(item_type, address, item, ierr)`
Fortran77	`call UPS_CM_SM_SET_ITEM`	`(item_type, address, item, ierr)`

Arguments

`item_type`
Intent: in
C type: UPS_AA_Mem_item_enum
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The type of info requested.
Please see UPS_AA_Mem_item_enum (section B page 49) for a listing/explanation of different items.

`address`
Intent: inout
C type: volatile void*
Fortran type: UPS_KIND_ADDRESS {0}
Fortran77 type: UPS_KIND_ADDRESS {0}
The address of the sm area gotten by UPS_CM_SM_malloc. For some item_type values, this argument is not used.

`item`
Intent: out
C type: void*
Fortran type: user_choice {0::item_type}
Fortran77 type: user_choice {0::item_type}
The input value of the item_type. See item_type above.

`ierr`
Intent: out
C type: (na) int return value
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

Currently, the only value that can be set is UPS_AA_MEM_ITEM_P_GROUP. This is the process set that can access the same shared memory area. By default, this will be set:

- **UPS_CM_P_GROUP_BOX -** most systems with shared memory.
- **UPS_CM_P_GROUP_SELF -** systems without shared memory.

On certain systems, you might wish to change this to better fit your needs (or for testing purposes). Note that some values are not supported and an error will be returned in the UPS_CM_Sm_malloc() call.

Examples

See shared_memory.c (section 6.2.1 page 23) for a detailed example.

SeeAlso

`UPS_CM_Sm_free` (page 119)
`UPS_CM_Sm_malloc` (page 122)
`UPS_CM_Sm_get_item` (page 120)
`UPS_CM_Sm_set_item` (page 124)
`UPS_UT_Mem_get_item` (page 242)
`UPS_AA_Statistics` (page 99)

C.4 Data parallel

See the packages section (section 6.3, page 27 for a general description of this package.

This section contains an alphabetical listing of the data parallel routines available in UPS.

UPS_DP_Combiner()

Package

dp

Purpose

UPS_DP_Combiner performs the specified data parallel operation. (Options listed below.)

Usage

C	`ierr = UPS_DP_Combiner`	`(in, count, datatype, combiner_op, out);`
Fortran	`call UPSF_DP_COMBINER`	`(in, count, datatype, combiner_op, out, MASK=mask, ierr)`
Fortran77	`call UPS_DP_COMBINER`	`(in, count, datatype, combiner_op, out, ierr)`

Arguments

`in`
- *Intent*: in
- *C type*: const void*
- *Fortran type*: user_choice {0-1}
- *Fortran77 type*: user_choice {0-1}

The input vector of data.

`count`
- *Intent*: in
- *C type*: int
- *Fortran type*: UPS_KIND_INT4 {0}
- *Fortran77 type*: UPS_KIND_INT4 {0}

The number of elements in input array src

`datatype`
- *Intent*: in
- *C type*: UPS_DT_Datatype_enum
- *Fortran type*: UPS_KIND_INT4 {0}
- *Fortran77 type*: UPS_KIND_INT4 {0}

The type of the local data.
Please see UPS_DT_Datatype_enum
(section B page 63)
for a listing of the possible datatypes.

`combiner_op`
- *Intent*: in
- *C type*: UPS_AA_Operation_enum
- *Fortran type*: UPS_KIND_INT4 {0}
- *Fortran77 type*: UPS_KIND_INT4 {0}

The operation to perform.
Please see UPS_AA_Operation_enum

(section B page 56)
for a listing of the possible operations.

`out`	*Intent*:	out
	C type:	void*
	Fortran type:	user_choice {0}
	Fortran77 type:	user_choice {0}
	The output vector.	
`mask`	*Intent*:	in
	C type:	(na) see UPS_DP_Combinerm
	Fortran type:	(optional:UPS_DP_COMBINERM) UPS_KIND_INT4 {in}
	Fortran77 type:	(na) see UPS_DP_Combinerm
	true/false integer array specifying which elements of the input vector are to be operated upon.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

UPS_DP_Combiner is a template with respect to the high level view of the data parallel operations it will perform. First the operation is applied to the local data, then a global operation completes the function.

Note, in the `UPS_AA_MAXLOC` or `UPS_AA_MINLOC` functions, the integer location returned is 0 based. That is, if the first element is returned, the value will be 0.

SeeAlso

`UPS_DP_Combiner` (page 126)
`UPS_DP_Combinerm` (page 127)

UPS_DP_Combinerm()

Package

dp

Purpose

UPS_DP_Combinerm performs the specified data parallel operation. (Options listed below.)

The mask is an array specifying which elements do not participate.

Usage

C	`ierr = UPS_DP_Combinerm`	`(in, count, datatype, combiner_op, out, mask);`
Fortran77	`call UPS_DP_COMBINERM`	`(in, count, datatype, combiner_op, out, mask, ierr)`

Arguments

`in`
- *Intent*: in
- *C type*: const void*
- *Fortran type*: (na) see UPS_DP_Combiner
- *Fortran77 type*: user_choice {0-1}

The input vector of data.

`count`
- *Intent*: in
- *C type*: int
- *Fortran type*: (na) see UPS_DP_Combiner
- *Fortran77 type*: UPS_KIND_INT4 {0}

The number of elements in input array src

`datatype`
- *Intent*: in
- *C type*: UPS_DT_Datatype_enum
- *Fortran type*: (na) see UPS_DP_Combiner
- *Fortran77 type*: UPS_KIND_INT4 {0}

The type of the local data.
Please see UPS_DT_Datatype_enum
(section B page 63)
for a listing of the possible datatypes.

`combiner_op`
- *Intent*: in
- *C type*: UPS_AA_Operation_enum
- *Fortran type*: (na) see UPS_DP_Combiner
- *Fortran77 type*: UPS_KIND_INT4 {0}

The operation to perform.
Please see UPS_AA_Operation_enum
(section B page 56)
for a listing of the possible operations.

`out`
- *Intent*: out
- *C type*: void*
- *Fortran type*: (na) see UPS_DP_Combiner
- *Fortran77 type*: user_choice {0}

The output vector.

`mask`
- *Intent*: in
- *C type*: const int*
- *Fortran type*: (na) see UPS_DP_Combiner
- *Fortran77 type*: UPS_KIND_INT4 {in}

true/false integer array specifying which elements
of the input vector are to be operated upon.

`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	(na) see UPS_DP_Combiner
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

UPS_DP_Combiner is a template with respect to the high level view of the data parallel operations it will perform. First the operation is applied to the local data, then a global operation completes the function.

Note, in the `UPS_AA_MAXLOC` or `UPS_AA_MINLOC` functions, the integer location returned is 0 based. That is, if the first element is returned, the value will be 0. The location returned will be relative to the entire array (masked and unmasked elements). A call to UPS_DP_Number_mask can be made to find the location with respect to only those participating elements.

SeeAlso

`UPS_DP_Combiner` (page 126)
`UPS_DP_Combinerm` (page 127)

UPS_DP_Count_mask()

Package

dp

Purpose

UPS_DP_Count_mask, for each participating element of the mask, adds the integer 1 to the return value.

Usage

C	`ierr = UPS_DP_Count_mask`	`(mask, count, out);`
Fortran	`call UPSF_DP_COUNT_MASK`	`(mask, count, out, ierr)`
Fortran77	`call UPS_DP_COUNT_MASK`	`(mask, count, out, ierr)`

Arguments

`mask`	*Intent*:	in
	C type:	const int*
	Fortran type:	UPS_KIND_INT4 {0-1}
	Fortran77 type:	UPS_KIND_INT4 {0-1}
	boolean (int) array specifying which elements are to be operated on	
`count`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}

number of elements in input array src

`out`	*Intent*:	out
	C type:	int*
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The output value.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

SeeAlso

`UPS_DP_Number_mask` (page 133)

UPS_DP_Dot_product()

Package

dp

Purpose

UPS_DP_Dot_product returns the dot product $x^T y = (x, y)$ of the input vectors x and y.

Usage

C	`ierr = UPS_DP_Dot_product`	`(x, y, count, datatype, x_dot_y);`
Fortran	`call UPSF_DP_DOT_PRODUCT`	`(x, y, count, datatype, x_dot_y, MASK=mask, ierr)`
Fortran77	`call UPS_DP_DOT_PRODUCT`	`(x, y, count, datatype, x_dot_y, ierr)`

Arguments

`x`	*Intent*:	in
	C type:	const void*
	Fortran type:	user_choice {0-1}
	Fortran77 type:	user_choice {0-1}
	The input vector x.	
`y`	*Intent*:	in
	C type:	const void*
	Fortran type:	user_choice {x}
	Fortran77 type:	user_choice {x}
	The input vector y.	

`count`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The number of elements in vectors of x and y.	
`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The datatype of x and y.	
`x_dot_y`	*Intent*:	out
	C type:	void*
	Fortran type:	user_choice {0}
	Fortran77 type:	user_choice {0}
	The dot product of vector x and vector y.	
`mask`	*Intent*:	in
	C type:	(na) see UPS_DP_Dot_productm
	Fortran type:	(optional:UPS_DP_DOT_PRODUCTM) UPS_KIND_INT4 {x}
	Fortran77 type:	(na) see UPS_DP_Dot_productm
	true/false integer array specifying which elements of the input vector are to be operated upon.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

SeeAlso

`UPS_DP_Dot_productm` (page 131)

UPS_DP_Dot_productm()

Package

dp

Purpose

UPS_DP_Dot_productm extends the capability of UPS_DP_Dot_product by adding a masking capability.

Usage

C	`ierr = UPS_DP_Dot_productm`	`(x, y, count, datatype, x_dot_y, mask);`
Fortran77	`call UPS_DP_DOT_PRODUCTM`	`(x, y, count, datatype, x_dot_y, mask, ierr)`

Arguments

`x`

Intent:	in
C type:	const void*
Fortran type:	(na) see UPS_DP_Dot_product
Fortran77 type:	user_choice {0-1}

The input vector x.

`y`

Intent:	in
C type:	const void*
Fortran type:	(na) see UPS_DP_Dot_product
Fortran77 type:	user_choice {x}

The input vector y.

`count`

Intent:	in
C type:	int
Fortran type:	(na) see UPS_DP_Dot_product
Fortran77 type:	UPS_KIND_INT4 {0}

The number of elements in vectors of x and y.

`datatype`

Intent:	in
C type:	UPS_DT_Datatype_enum
Fortran type:	(na) see UPS_DP_Dot_product
Fortran77 type:	UPS_KIND_INT4 {0}

The datatype of x and y.

`x_dot_y`

Intent:	out
C type:	void*
Fortran type:	(na) see UPS_DP_Dot_product
Fortran77 type:	user_choice {0}

The dot product of vector x and vector y.

`mask`

Intent:	in
C type:	const int*
Fortran type:	(na) see UPS_DP_Dot_product
Fortran77 type:	UPS_KIND_INT4 {x}

true/false integer array specifying which elements are to be operated on.

`ierr`

Intent:	out
C type:	(na) int return value
Fortran type:	(na) see UPS_DP_Dot_product
Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

SeeAlso

`UPS_DP_Dot_product` (page 130)

UPS_DP_Number_mask()

Package

dp

Purpose

UPS_DP_Number_mask assigns elements of array "out", defined as participating by the mask, an ascending counting number. Elements not participating will not be changed.

Usage

C	`ierr = UPS_DP_Number_mask`	`(mask, count, out, lower_bound, upper_bound);`
Fortran	`call UPSF_DP_NUMBER_MASK`	`(mask, count, out, lower_bound, upper_bound, ierr)`
Fortran77	`call UPS_DP_NUMBER_MASK`	`(mask, count, out, lower_bound, upper_bound, ierr)`

Arguments

`mask`
Intent: in
C type: const int*
Fortran type: UPS_KIND_INT4 {0-1}
Fortran77 type: UPS_KIND_INT4 {0-1}
Boolean (int) array specifying which elements are to be operated on.

`count`
Intent: in
C type: int
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The number of elements in input array mask.

`out`
Intent: out
C type: int*
Fortran type: UPS_KIND_INT4 {mask}
Fortran77 type: UPS_KIND_INT4 {mask}
The output vector output.

`lower_bound`
Intent: out
C type: int*
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The lowest enumeration for this process.

`upper_bound`	*Intent*:	out
	C type:	int*
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The largest enumeration for this process.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

SeeAlso

`UPS_DP_Count_mask` (page 129)

UPS_DP_Sort()

Package

dp

Purpose

UPS_DP_Sort sorts the elements in the specified distributed array.

Usage

C	`ierr = UPS_DP_Sort`	`(buf, count, datatype);`
Fortran	`call UPSF_DP_SORT`	`(buf, count, datatype, ierr)`
Fortran77	`call UPS_DP_SORT`	`(buf, count, datatype, ierr)`

Arguments

`buf`	*Intent*:	inout
	C type:	void*
	Fortran type:	user_choice {0-1}
	Fortran77 type:	user_choice {0-1}
	On input, the unsorted data. On output, the sorted data.	
`count`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The number of elements in input array buf.	
`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	UPS_KIND_INT4 {0}

	Fortran77 type:	UPS_KIND_INT4 {0}
	The datatype of the input vector elements in buf.	
ierr	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

- Process

  This sort creates bins by taking a sample of data from each pe. Each pe sends its data to the respective bin on the respective pe. Those pe's sort their local data. Then the data is redistributed to the original counts on each pe.

- Pre-Sorting

  The original data is sorted as well in this process. So, there is no point in sorting the local data yourself before sending it to UPS_DP_Sort.

- Sorting Val-Loc Datatypes

  The Val-Loc datatypes may also be sorted (eg UPS_DT_DOUBLE_INT). This is useful if you want to attach info to the sorted values (eg [value, penum] or [value, original index]).

UPS_DP_Vector_norm()

Package

dp

Purpose

UPS_DP_Vector_norm returns the 2-norm, $\|x\|\_2 = \sqrt{x^T x}$, of the input vector x.

Usage

C	`ierr = UPS_DP_Vector_norm`	`(x, count, datatype, norm_x);`
Fortran	`call UPSF_DP_VECTOR_NORM`	`(x, count, datatype, norm_x, MASK=mask, ierr)`
Fortran77	`call UPS_DP_VECTOR_NORM`	`(x, count, datatype, norm_x, ierr)`

Arguments

x	*Intent*:	in
	C type:	void*
	Fortran type:	user_choice {0-1}
	Fortran77 type:	user_choice {0-1}
	The input vector x.	

`count`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The number of elements of vector x.	
`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The datatype of x.	
`norm_x`	*Intent*:	out
	C type:	void*
	Fortran type:	user_choice {0}
	Fortran77 type:	user_choice {0}
	The 2-norm of vector x.	
`mask`	*Intent*:	in
	C type:	(na) see UPS_DT_Vector_normm
	Fortran type:	(optional:UPS_DP_VECTOR_NORMM) UPS_KIND_INT4 {x}
	Fortran77 type:	(na) see UPS_DT_Vector_normm
	true/false integer array specifying which elements of the input vector are to be operated upon.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

`UPS_DP_Vector_norm` first computes the inner product of vector `x` using `UPS_DP_Dot_product`, then returns the square root of that value.

SeeAlso

`UPS_DP_Vector_normm` (page 136)

UPS_DP_Vector_normm()

Package

dp

Purpose

UPS_DP_Vector_normm extends the capability of UPS_DP_Vector_norm by adding a masking capability.

Usage

C	`ierr = UPS_DP_Vector_normm`	`(x, count, datatype, norm_x, mask);`
Fortran77	`call UPS_DP_VECTOR_NORMM`	`(x, count, datatype, norm_x, mask, ierr)`

Arguments

`x`
Intent: in
C type: const void*
Fortran type: (na) see UPS_DP_Vector_norm
Fortran77 type: user_choice {0-1}
The input vector x.

`count`
Intent: in
C type: int
Fortran type: (na) see UPS_DP_Vector_norm
Fortran77 type: UPS_KIND_INT4 {0}
The number of elements of vector x.

`datatype`
Intent: in
C type: UPS_DT_Datatype_enum
Fortran type: (na) see UPS_DP_Vector_norm
Fortran77 type: UPS_KIND_INT4 {0}
The datatype of x.

`norm_x`
Intent: out
C type: void*
Fortran type: (na) see UPS_DP_Vector_norm
Fortran77 type: user_choice {0}
The 2-norm of vector x.

`mask`
Intent: in
C type: const int*
Fortran type: (na) see UPS_DP_Vector_norm
Fortran77 type: UPS_KIND_INT4 {x}
true/false integer array specifying which elements are to be operated on.

`ierr`
Intent: out
C type: (na) int return value
Fortran type: (na) see UPS_DP_Vector_norm
Fortran77 type: UPS_KIND_INT4 {0}
Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

`UPS_DP_Vector_normm` first computes the inner product of vector `x` using `UPS_DP_Dot_productm`, then returns the square root of that value.

SeeAlso

`UPS_DP_Vector_norm` (page 135)

C.5 Datatypes

See the packages section (section 6.4, page 28 for a general description of this package.

This section contains an alphabetical listing of the routines available for use with UPS datatypes. These routines may be called in conjunction with any UPS component.

UPS_DT_Sizeof()

Package

dt

Purpose

UPS_DT_Sizeof returns the number of bytes allocated for each element of the input datatype.

Usage

C	`ierr = UPS_DT_Sizeof`	`(datatype, sizeof_datatye);`
Fortran	`call UPSF_DT_SIZEOF`	`(datatype, sizeof_datatye, ierr)`
Fortran77	`call UPS_DT_SIZEOF`	`(datatype, sizeof_datatye, ierr)`

Arguments

`datatype`

Intent: in
C type: UPS_DT_Datatype_enum
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The type of the data.
Please see UPS_DT_Datatype_enum
(section B page 63)
for a listing of the possible datatypes.

`sizeof_datatye`

Intent: out
C type: int*
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
size (as an int) of the datatype.
Value of -1 indicates datatype unknown.

`ierr`

Intent: out
C type: (na) integer return value
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Error return value

ReturnValues

Returns UPS_OK if successful.

Discussion

C language users will recognize this function as being analogous to the “sizeof” function.

C.6 Error handling

See the packages section (section 6.5, page 29 for a general description of this package.

This section contains an alphabetical listing of the error handling routines available in UPS.

UPS_ER_Get_wait_time()

Package

er

Purpose

Return the maximum time, in seconds, a process will spend after a call to `UPS_ER_Set_alarm` and before `UPS_ER_Unset_alarm`.

Usage

C	`ierr = UPS_ER_Get_wait_time`	`(max_wait_time);`
Fortran	`call UPSF_ER_GET_WAIT_TIME`	`(max_wait_time, ierr)`
Fortran77	`call UPS_ER_GET_WAIT_TIME`	`(max_wait_time, ierr)`

Arguments

`max_wait_time`	*Intent*:	out
	C type:	int*
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Time before alarm is triggered.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

The time waiting can be reset with a call to `UPS_ER_Set_wait_time`.

Notes

Please see the ER alarm variable section of UPS_AA_ENVIRONMENT_VARIABLES_enum (section B page 43) for environment variables that affect this call.

SeeAlso

`UPS_ER_Get_wait_time` (page 140)
`UPS_ER_Set_alarm` (page 142)
`UPS_ER_Set_wait_time` (page 143)
`UPS_ER_Unset_alarm` (page 144)

UPS_ER_Perror()

Package

er

Purpose

UPS_ER_Perror prints the input string and integer code to `stdout`. The message is prepended by the calling process number followed by "**UPS Error**".

Usage

C	`ierr = UPS_ER_Perror`	`(s1, info);`
Fortran	`call UPSF_ER_PERROR`	`(s1, info, ierr)`
Fortran77	`call UPS_ER_PERROR`	`(s1, info, ierr)`

Arguments

`s1`	*Intent*:	in
	C type:	const char*
	Fortran type:	UPS_KIND_CHAR {0}
	Fortran77 type:	UPS_KIND_CHAR {0}
	The user supplied string to be printed. This string must be null-terminated. As an example, Fortran users may pass in a string concatenated with the null-character: 'main (foo)'//ACHAR(0)	
`info`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The user supplied integer to be printed.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful

Discussion

The message is prepended by the calling process number (except for single process execution). The purpose of this function is to ensure that messages the user wants printed actually appear in a timely fashion. At the simplest level this means `fprintf(stdout,...)` followed by a buffer flush. The user should be aware, however, that we make no attempt at ordering messages among processes since that would require additional synchronization, and this may alter the behavior of execution.

UPS_ER_Set_alarm()

Package

er

Purpose

UPS_ER_Set_alarm sets an "alarm" to go off and terminate execution if the alarm is not reset (using `UPS_ER_Unset_alarm`) within the time specified with a prior call to `UPS_ER_Set_wait_time` or through the environment variable `UPS_ER_MAX_WAIT_TIME`.

Usage

C	`ierr = UPS_ER_Set_alarm`	`();`
Fortran	`call UPSF_ER_SET_ALARM`	`(ierr)`
Fortran77	`call UPS_ER_SET_ALARM`	`(ierr)`

Arguments

`ierr`

Intent:	out
C type:	(na) int return value
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

This function does not behave recursively. That is, successive calls simply reset the alarm clock of the calling process.

We originally incorporated alarms directly into UPS functions. However, performance was adversely affected due to the small granularity. Likewise, alarms should be placed in user applications sparingly.

DANGER: this will grab SIGALRM and pass it to upsp_er_sig_alarm. This might interfere with traps set by other functions (eg debuggers, profilers).

Examples

In the following example, the user sets the maximum time to wait for completion to 60 seconds. The alarm is set, and work is begun. If the work is not completed within 60 seconds, all processes are terminated. If, however, the work is completed within 60 seconds, the alarm is unset, and execution continues.

```
ierr = UPS_ER_Set_wait_time( 60 );
ierr = UPS_ER_Set_alarm();

( Do work.)

ierr = UPS_ER_Unset_alarm();
```

Notes

Please see the ER alarm variable section of UPS_AA_ENVIRONMENT_VARIABLES_enum (section B page 43) for environment variables that affect this call.

SeeAlso

`UPS_ER_Get_wait_time` (page 140)
`UPS_ER_Set_alarm` (page 142)
`UPS_ER_Set_wait_time` (page 143)
`UPS_ER_Unset_alarm` (page 144)

UPS_ER_Set_wait_time()

Package

er

Purpose

UPS_ER_Set_wait_time lets the caller over-ride the default maximum time waiting for an alarm to be re-set. This is designed to prevent program "hangs".

Usage

C	`ierr = UPS_ER_Set_wait_time`	`(max_wait_time);`
Fortran	`call UPSF_ER_SET_WAIT_TIME`	`(max_wait_time, ierr)`
Fortran77	`call UPS_ER_SET_WAIT_TIME`	`(max_wait_time, ierr)`

Arguments

`max_wait_time`

Intent:	in
C type:	int
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Time (seconds) before the alarm is triggered when the alarm is actived.

`ierr`

Intent:	out
C type:	(na) int return value
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

See `UPS_ER_Set_alarm` for details.

Examples

See the reference page for `UPS_ER_Set_alarm`.

Notes

Please see the ER alarm variable section of UPS_AA_ENVIRONMENT_VARIABLES_enum (section B page 43) for environment variables that affect this call.

SeeAlso

`UPS_ER_Get_wait_time` (page 140)
`UPS_ER_Set_alarm` (page 142)
`UPS_ER_Set_wait_time` (page 143)
`UPS_ER_Unset_alarm` (page 144)

UPS_ER_Unset_alarm()

Package

er

Purpose

UPS_ER_Unset_alarm disables the alarm that has been set to go off via a call to `UPS_ER_Set_alarm`.

Usage

C	`ierr = UPS_ER_Unset_alarm`	`();`
Fortran	`call UPSF_ER_UNSET_ALARM`	`(ierr)`
Fortran77	`call UPS_ER_UNSET_ALARM`	`(ierr)`

Arguments

`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

See `UPS_ER_Set_alarm`. This function resets the function to jump to during a SIGALRM from upsp_er_sig_alarm to whatever function was previously set (if any).

Examples

See the reference page for `UPS_ER_Set_alarm`.

SeeAlso

`UPS_ER_Get_wait_time` (page 140)
`UPS_ER_Set_alarm` (page 142)
`UPS_ER_Set_wait_time` (page 143)
`UPS_ER_Unset_alarm` (page 144)

C.7 Gather/scatter

See the packages section (section 6.6, page 30 for a general description of this package.

This section contains an alphabetical listing of the gather/scatter routines available in UPS.

UPS_GS_Collate()

Package

gs

Purpose

UPS_GS_Collate collects data from all participating processes onto the designated collector process. The data from individual processes need not be of the same count. (The collector process is by default set to process 0; this may be changed with a call to `UPS_AA_Io_pe_set`.)

Usage

C	`ierr = UPS_GS_Collate`	`(in, count, out, datatype);`
Fortran	`call UPSF_GS_COLLATE`	`(in, count, out, datatype, ierr)`
Fortran77	`call UPS_GS_COLLATE`	`(in, count, out, datatype, ierr)`

Arguments

`in`	*Intent*:	in
	C type:	void*
	Fortran type:	user_choice {0-1}
	Fortran77 type:	user_choice {0-1}
	The input data from each process.	
`count`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The number of elements the individual process is contributing.	
`out`	*Intent*:	out
	C type:	void*
	Fortran type:	user_choice {0-1}
	Fortran77 type:	user_choice {0-1}
	The location of the collected data on the collector process. (Significant only at the collector process.)	
`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The datatype of the elements in array “in”.	
`ierr`	*Intent*:	out

C type:	(na) int return value
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

(This discussion will be presented in terms of MPI functionality.) `UPS_GS_Collate` is essentially a call to `MPI_Gatherv`, where the root process doesn't have advance knowledge of the number of elements it is to receive from the participating processes. The user should be aware that this (usually) requires an overhead cost beyond what is necessary for a direct call to `MPI_Gatherv`.

SeeAlso

`UPS_GS_Collate` (page 146)
`UPS_GS_Distribute` (page 147)

UPS_GS_Distribute()

Package

gs

Purpose

UPS_GS_Distribute distributes the data from the `in` buffer on the distributor process to the other processes in the current processor context. (The distributor process is by default set to process 0; this may be changed with a call to `UPS_AA_Io_pe_set`.)

Usage

C	`ierr = UPS_GS_Distribute`	`(in, counts, out, datatype);`
Fortran	`call UPSF_GS_DISTRIBUTE`	`(in, counts, out, datatype, ierr)`
Fortran77	`call UPS_GS_DISTRIBUTE`	`(in, counts, out, datatype, ierr)`

Arguments

`in`

Intent:	in
C type:	void*
Fortran type:	user_choice {0-1:numpes}
Fortran77 type:	user_choice {0-1:numpes}

The data to be distributed.
(Significant only at the distributor process.)

`counts`

Intent:	in
C type:	int*
Fortran type:	UPS_KIND_INT4 {0-1:numpes}
Fortran77 type:	UPS_KIND_INT4 {0-1:numpes}

A participating process will receive counts[penum] elements.
(Significant only at the distributor process.)

`out`	*Intent*:	out
	C type:	void*
	Fortran type:	user_choice {0-1:counts}
	Fortran77 type:	user_choice {0-1:counts}
	The data received by each process.	
`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The datatype of the elements in the "in" buffer.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

(This discussion will be presented in terms of MPI functionality.) `UPS_GS_Distribute` is essentially a call `MPI_Scatterv`, where the receiving processes don't have prior knowledge regarding the number of elements they are to receive. The user should be aware that this (usually) requires an overhead cost beyond what is necessary for a direct call to `MPI_Scatterv`.

SeeAlso

`UPS_GS_Collate` (page 146)
`UPS_GS_Distribute` (page 147)

UPS_GS_Free()

Package

gs

Purpose

UPS_GS_Free destroys the gather/scatter database associated with the input handle `gs_id`.
If the gs_id is 0, all gs structures will be freed.
In order to help debug memory leaks, an error message is printed if there are any gs structures that have not been freed.

Usage

C	`ierr = UPS_GS_Free`	`(gs_id);`
Fortran	`call UPSF_GS_FREE`	`(gs_id, ierr)`
Fortran77	`call UPS_GS_FREE`	`(gs_id, ierr)`

Arguments

`gs_id`	*Intent*:	in

	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The handle for the associated gather/scatter database.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

SeeAlso

`UPS_GS_Setup` (page 165)

UPS_GS_Gather()

Package

gs

Purpose

UPS_GS_Gather copies specific indices from the globally distributed array `in` into the local array `out`. The indices copied are set up by a call to `UPS_GS_Setup` - which returns a handle, `gs_id`, for future gather/scatters.

There might be cases where one `gs_id` is to be used for several arrays at the same time. This is accomplished by the `count` argument. Depending on how the arrays are laid out in memory, either `UPS_GS_Gather` or `UPS_GS_Gather_multi` should be used. Below is an example of the two types and which function call one should use.

```
a(n) = the nth array
i(m) = the mth index (as set up in UPS_GS_Setup)

UPS_GS_Gather:
     a(1),i(n) | a(2),i(n) | a(3),i(n) ... (the n-th index is blocked)

UPS_GS_Gather_multi:
     a(m),i(1) | a(m),i(2) | a(m),i(3) ... (the m-th array is blocked)
```

Usage

C	`ierr = UPS_GS_Gather`	`(in, out, datatype, count, gs_id);`
Fortran	`call UPSF_GS_GATHER`	`(in, out, datatype, count, gs_id, ierr)`
Fortran77	`call UPS_GS_GATHER`	`(in, out, datatype, count, gs_id, ierr)`

Arguments

`in`	*Intent*:	in
	C type:	void*
	Fortran type:	user_choice {0-2:setup}
	Fortran77 type:	user_choice {0-2:setup}
	The local portion of the globally distributed array.	
`out`	*Intent*:	out
	C type:	void*
	Fortran type:	user_choice {0-2:setup}
	Fortran77 type:	user_choice {0-2:setup}
	Local array gotten from the global indices defined by `UPS_GS_Setup`.	
`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The datatype of the elements.	
`count`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Number of times to perform a gather.	
`gs_id`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The handle for the database set in a prior call to `UPS_GS_Setup`.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

Each pe "owns" an index space. Suppose pe 0 owns indices [0-2] and pe 1 owns indices [3-5]. In the gather, suppose both pe 0 and pe 1 wish to see the global vector. Here are some sample inputs:

process	`in`	`out`	indices
0	1, 2, 3	*currently undefined*	0, 1, 2, 3, 4, 5
1	4, 5, 6	*currently undefined*	5, 4, 3, 2, 1, 0

The above input says:

- process 0 owns indices 0, 1, and 2 (whose values are 1, 2, and 3 respectively). It wishes to see the global vector, in order, and
- process 1 owns indices 3, 4, and 5 (who values are 4, 5, and 6 respectively). It wishes to see the global vector in reverse order.

The desired result is:

process	`in`	`out`	indices
0	1, 2, 3	**1, 2, 3, 4, 5, 6**	0, 1, 2, 3, 4, 5
1	4, 5, 6	**6, 5, 4, 3, 2, 1**	5, 4, 3, 2, 1, 0

SeeAlso

`UPS_GS_Setup` (page 165)
`UPS_GS_Gather_multi` (page 153)
`UPS_GS_Scatter` (page 156)

UPS_GS_Gather_list()

Package

gs

Purpose

This function is the reverse of UPS_GS_Scatter_list. It takes the "listed" input and sends it out to the corresponging processes. In a mirror of UPS_GS_Scatter_list, the input is the "list" and the output is the same type of output as a normal UPS_GS_Gather call.

See see UPS_GS_Scatter_list (section C.7 page 158) for more information.

Usage

C	`ierr = UPS_GS_Gather_list`	`(in, out, datatype, count, gs_id);`
Fortran	`call UPSF_GS_GATHER_LIST`	`(in, out, datatype, count, gs_id, ierr)`
Fortran77	`call UPS_GS_GATHER_LIST`	`(in, out, datatype, count, gs_id, ierr)`

Arguments

`in`
Intent: in
C type: void*
Fortran type: user_choice {0-2:setup}
Fortran77 type: user_choice {0-2:setup}
The local "listed" array. The indices of the globally distributed array `out` to which these elements correspond to are defined by `UPS_GS_Setup`.

`out`
Intent: out
C type: void*

	Fortran type:	user_choice {0-2:setup}
	Fortran77 type:	user_choice {0-2:setup}
	The list values after corresponding to this process.	
`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The datatype of the elements.	
`count`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Number of times to perform a gather_list. Currently, only count value of 1 is supported.	
`gs_id`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The handle for the database set in a prior call to `UPS_GS_Setup`.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

See see UPS_GS_Scatter_list (section C.7 page 158) for more information.

Examples

See see UPS_GS_Scatter_list (section C.7 page 158) for more information.

SeeAlso

`UPS_GS_Setup` (page 165)
`UPS_GS_Gather` (page 149)
`UPS_GS_Scatter` (page 156)
`UPS_GS_Gather_list` (page 151)
`UPS_GS_Scatter_list` (page 158)
`UPS_GS_Get_item` (page 155)

UPS_GS_Gather_multi()

Package

gs

Purpose

UPS_GS_Gather_multi performs UPS_GS_Gather `count` times given the array of `in` and `out` addresses. See the purpose section of `UPS_GS_Gather` (section C.7 page 149) for a more detailed description.

Usage

C	`ierr = UPS_GS_Gather_multi`	`(in_multi, out_multi, datatype, count, gs_id);`
Fortran	`call UPSF_GS_GATHER_MULTI`	`(in_multi, out_multi, datatype, count, gs_id, ierr)`
Fortran77	`call UPS_GS_GATHER_MULTI`	`(in_multi, out_multi, datatype, count, gs_id, ierr)`

Arguments

`in_multi`	*Intent*:	in
	C type:	void**
	Fortran type:	UPS_KIND_ADDRESS {0-1}
	Fortran77 type:	UPS_KIND_ADDRESS {0-1}
	Array of input addresses	
`out_multi`	*Intent*:	out
	C type:	void**
	Fortran type:	UPS_KIND_ADDRESS {in_multi}
	Fortran77 type:	UPS_KIND_ADDRESS {in_multi}
	Array of output addresses	
`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Datatype of the elements	
`count`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Number of times to perform a gather.	
`gs_id`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The handle for the database set in a prior call to `UPS_GS_Setup`.	
`ierr`	*Intent*:	out

C type:	(na) int return value
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

- **Getting Addresses in Fortran**

  Getting an array of addresses in C is second nature. However, some work has to be done to get an equivalent array in Fortran. Some possibilities are:

  - **MPI_Address()**

    Send the first element of your array to this MPI function
  - **Write a C function**

    ```
    void GET_ADDRESS(
                       const void *ptr,
                       void **ptr_address
                       )
    {
       *ptr_address = (void*)ptr;
       return;
    }
    ```

    Call this function (like MPI_Address) from your Fortran routine. Store this address in a variable of type UPS_KIND_ADDRESS

  It is important that the size of the variable holding the address matches with what libups.a expects. This variable is simply cast as a `C void**`. One must be careful when compiling with 64 or n32 bit addressing.

  - **C**

    Just pass in a `void**`.
  - **Fortran**

    Just use a `UPS_KIND_ADDRESS` array
  - **Fortran77**

    When using 64 bit addressing, use an `int*8` array. When using n32 bit addressing, use an `int*4` array.

- **Performance**

  - **Fewer Communication Calls .vs. Increased Message Size**

    Combining messages is a good thing because it means, in general, fewer communication calls. However, these messages will be larger. There might be some threshold to message size where under a certain size, some protocol is used to transfer data and over that threshold, another protocol is used. Therefor, decreasing the number of communication calls but increasing the message size might actually hurt performance.

The user is encouraged to try the multi-call and the single call to see which offers the better performance (if any).

– **multi value should be small**

Essentially, gather and scatters are just copying data from one buffer to another. The multi-calls just put this data copying in a loop. For code simplicity/readability, this loop was placed at a high level. Thus, high values for the `multi` argument might not perform very well.

SeeAlso

`UPS_GS_Setup` (page 165)
`UPS_GS_Gather` (page 149)
`UPS_GS_Scatter` (page 156)

UPS_GS_Get_item()

Package

gs

Purpose

Get info about things in the gs package. Currently, the items that can be gotten from UPS_GS_Get_item are useful mainly for the UPS_GS_Scatter_list and UPS_GS_Gather_list functions.

Usage

C	`ierr = UPS_GS_Get_item`	`(item_type, gs_id, item);`
Fortran	`call UPSF_GS_GET_ITEM`	`(item_type, gs_id, item, ierr)`
Fortran77	`call UPS_GS_GET_ITEM`	`(item_type, gs_id, item, ierr)`

Arguments

`item_type`
Intent: in
C type: UPS_GS_Item_enum
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The type of info requested.
Please see UPS_GS_Item_enum (section B page 65) for a listing/explanation of different items.

`gs_id`
Intent: in
C type: int
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The handle for the database set in a prior call to `UPS_GS_Setup`.

`item`
Intent: out
C type: void*
Fortran type: user_choice {0-1:item_type:item_type}
Fortran77 type: user_choice {0-1:item_type:item_type}
The output value of the item_type. See item_type

above.

`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

When a gather/scatter communication pattern is set up via UPS_GS_Setup, it is known which global indices the process wishes to "deal with" since those indices are supplied to the setup call. However, it might not be known how other processes might access the indices owned by this process. This function allows access to this information.

Examples

See see UPS_GS_Scatter_list (section C.7 page 158) for a detailed example.

SeeAlso

`UPS_GS_Setup` (page 165)
`UPS_GS_Gather` (page 149)
`UPS_GS_Scatter` (page 156)
`UPS_GS_Gather_list` (page 151)
`UPS_GS_Scatter_list` (page 158)

UPS_GS_Scatter()

Package

gs

Purpose

UPS_GS_Scatter copies (with the `GS_func` operation) specific indices from the local array `in` into the globally distributed array `out`. The indices copied (then operated on) are set up by a call to `UPS_GS_Setup` - which returns a handle, `gs_id`, for future gather/scatters.

Usage

C	`ierr = UPS_GS_Scatter`	`(in, out, datatype, GS_func, count, gs_id);`
Fortran	`call UPSF_GS_SCATTER`	`(in, out, datatype, GS_func, count, gs_id, ierr)`
Fortran77	`call UPS_GS_SCATTER`	`(in, out, datatype, GS_func, count, gs_id, ierr)`

Arguments

`in`	*Intent*:	in
	C type:	void*
	Fortran type:	user_choice {0-2:setup}
	Fortran77 type:	user_choice {0-2:setup}

The local array. The indices of the globally distributed array `out` to which these elements correspond to are defined by `UPS_GS_Setup`.

`out`	*Intent*:	inout
	C type:	void*
	Fortran type:	user_choice {0-2:setup}
	Fortran77 type:	user_choice {0-2:setup}
	The local portion of the global distributed array.	
`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The datatype of the elements.	
`GS_func`	*Intent*:	in
	C type:	UPS_AA_Operation_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Please see UPS_AA_Operation_enum (section B page 56) for a listing of the possible operations.	
`count`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Number of times to perform a scatter. See the purpose section of `UPS_GS_Gather` (section C.7 page 149) for a more detailed description.	
`gs_id`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The handle for the database set in a prior call to `UPS_GS_Setup`.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

Each process "owns" an index space. Suppose process 0 owns indices [0-2] and process 1 owns indices [3-5]. In the scatter operation, process 0 and process 1 contribute to the global vector. Here are some sample inputs:

process	`in`	`out`	indices
0	1, 2, 3, 4, 5, 6	1, 2, 3	0, 1, 2, 3, 4, 5
1	2, 3, 4, 5, 6, 7	2, 3, 4	0, 1, 2, 3, 4, 5

Now let's say the operation is summation (`UPS_AA_SUM`). The above input says:

- process 0 owns global indices 0, 1, and 2 (whose values are currently 1, 2, and 3 respectively) and wishes to add 1, 2, 3, 4, 5, and 6 to global indices 0, 1, 2, 3, 4, 5 respectively.
- process 1 owns global indices 3, 4, and 5 (whose values are currently 2, 3, and 4 respectively) and wishes to add 2, 3, 4, 5, 6, and 7 to global indices 0, 1, 2, 3, 4, 5 respectively.

Upon return from `UPS_GS_Scatter`, the result is:

process	global index	`out` array
0	0	1 *(initial)* + 1 *(from pe0)* + 2 *(from pe1)* = **4**
	1	2 *(initial)* + 2 *(from pe0)* + 3 *(from pe1)* = **7**
	2	3 *(initial)* + 3 *(from pe0)* + 4 *(from pe1)* = **10**
1	3	2 *(initial)* + 4 *(from pe0)* + 5 *(from pe1)* = **11**
	4	3 *(initial)* + 5 *(from pe0)* + 6 *(from pe1)* = **14**
	5	4 *(initial)* + 6 *(from pe0)* + 7 *(from pe1)* = **17**

SeeAlso

`UPS_GS_Setup` (page 165)
`UPS_GS_Gather` (page 149)
`UPS_GS_Scatter_multi` (page 163)

UPS_GS_Scatter_list()

Package

gs

Purpose

This operation performs a particular type of scatter where, although the input is the same as a normal UPS_GS_Scatter call, the output is the list of all values that contributed to the indices owned by this process.

In a normal UPS_GS_Scatter call, all input values from all processes that contribute to any particular index owned by this process are combined by an operation (eg UPS_AA_SUM) and the value of that index is modified. For example, suppose in UPS_GS_Setup, a communication pattern was defined so that all processes accessed the first index owned by process 0. During UPS_GS_Gather, all processes would obtain a copy of the first index owned by process 0. During UPS_GS_Scatter, every process would combine their copy of the first index via some operation (eg UPS_AA_SUM) and the resulting value would be placed in first index owned by process 0. Processes could then get the value of this sum by then doing another UPS_GS_Gather. Of course, far more complex communication patterns are allowed.

In UPS_GS_Scatter_list, instead of combining the values of the contributing processes into 1 location, the values are copied sequentially to an output buffer. Using the above simple example:

```
          UPS_GS_Scatter: index[0]      =  pe0_val+pe1_val+pe2_val+...
     UPS_GS_Scatter_list: output_buffer = {pe0_val,pe1_val,pe2_val,...}
```

In general, the output buffer in UPS_GS_Scatter_list will contain a list of all the values that contributed to the indices owned by the calling process:

```
      output_buffer = [{indices contributing to index i},
                       {indices contributing to index j},
                       {indices contributing to index k}, ...]
```

In the output buffer, the ordering of the blocks of indices [i,j,k] is in order of smallest index to largest index. Indices that are not contributed to will not be in the output_buffer.

The ordering of the actual values in a particular set:

```
      {indices contributing to index m}
```

will be the same for any identical communication pattern. In other words, for any single gs_id obtained from UPS_GS_Setup, the same input buffers will always return the same output buffers for repeated UPS_GS_Scatter_list calls or repeated UPS_GS_Gather_list calls. Similarly, for one particular gs_id, corresponding elements of input array A and input array B will have the same positions in output array FOO and output array BAR. For example, one could do a UPS_GS_Scatter_list on integer input with a gs_id and then do UPS_GS_Scatter_list on float input with the same gs_id and have both the integer output and float output match up as would be expected.

Information can be obtained about the output buffer from a call to UPS_GS_Get_item:

- `UPS_GS_SUM_NUM_INDICES_ACCESSED`

  Total size of the output buffer

- `UPS_GS_NUM_INDICES_ACCESSED`

  Number of elements of each accessed index set ordered by index

  ```
  [|{indices contributing to index i}|,
   |{indices contributing to index j}|,
   |{indices contributing to index k}|, ...]
  ```

  Again, indices not accessed will not have sets in this list.

- `UPS_GS_INDICES_ACCESSED`

  Index values of the above set

  ```
  [i, j, k, ...]
  ```

- `UPS_GS_TOTAL_INDICES_ACCESSED`

  How many indices were accessed. (size of the arrays UPS_GS_NUM_INDICES_ACCESSED and UPS_GS_INDICES_ACCESSED)

- `UPS_GS_NUM_INDICES_ALL`

  Number of times all indices are accessed (indices not accessed will have a corresponding value of 0)

  ```
  [0, 0, |{indices contributing to index i}|, 0, |{j}|, |{k}|, ...]
  ```

  where the 0's correspond to indices not accessed. The nth element in this array is the number of times index n (0 based) was accessed. If the value is 0, then the nth index was not accessed.

See see UPS_GS_Get_info (section C.7 page 155) for other pieces of information that can be obtained. See the Examples section below for...an example.

Usage

C	`ierr = UPS_GS_Scatter_list`	`(in, out, datatype, count, gs_id);`
Fortran	`call UPSF_GS_SCATTER_LIST`	`(in, out, datatype, count, gs_id, ierr)`
Fortran77	`call UPS_GS_SCATTER_LIST`	`(in, out, datatype, count, gs_id, ierr)`

Arguments

`in`

Intent:	in
C type:	void*
Fortran type:	user_choice {0-2:setup}
Fortran77 type:	user_choice {0-2:setup}

The local array. The indices of the globally distributed array `out` to which these elements correspond to are defined by `UPS_GS_Setup`.

`out`

Intent:	out
C type:	void*
Fortran type:	user_choice {0-2:setup}
Fortran77 type:	user_choice {0-2:setup}

The list values which contribute to the indices owned by this process.

`datatype`

Intent:	in
C type:	UPS_DT_Datatype_enum
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The datatype of in/out buffers.

`count`

Intent:	in
C type:	int
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Number of times to perform a scatter_list. Currently, only count value of 1 is supported.

`gs_id`

Intent:	in
C type:	int
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The handle for the database set in a prior call to `UPS_GS_Setup`.

`ierr`

Intent:	out
C type:	(na) int return value
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

Performance...always nagging at me. The GS list functions are not speedy. Since lists of data are used, yet another level of indirection in the data copy loops is introduced. Also, strict ordering must be preserved so data compression and optimized send/recv ordering cannot be used. Do NOT use the list functions as a replacement for the normal gather/scatter functions - unless you don't care that they are slower and take up more memory. If you need additional functionality added to the normal gather/scatter functions, notify the UPS team (ups-team@lanl.gov).

Examples

The following ties together calls to UPS_GS_Setup, UPS_GS_Get_item, UPS_GS_Scatter_list, and UPS_GS_Gather_list.

C:

```
#include "ups.h"
#include <stdio.h>
void main()
{
  int
    i, j, gs_id, count = 5, penum, num_scatter_list_out,
    num_index_info, indices[5], my_space_size = 1, my_start_index = 0,
    *indices_accessed, *num_indices_accessed, *num_indices_all, position;
  double
    *scatter_list_output,
    scatter_list_input[5];
  UPS_AA_Init( NULL, NULL );
  UPS_CM_Get_penum( &(penum) );
  // perform a setup for some set of input parameters
  for( i = 0; i < count; i++ )
    {
      indices[i] = i;
    }
  UPS_GS_Setup( indices, my_space_size, my_start_index,
                UPS_GS_GLOBAL_INDEX, count, &(gs_id) );
  // get info about sizes of arrays
  UPS_GS_Get_item( UPS_GS_SUM_NUM_INDICES_ACCESSED, gs_id,
                   &(num_scatter_list_out) );
  UPS_GS_Get_item( UPS_GS_TOTAL_INDICES_ACCESSED, gs_id,
                   &(num_index_info) );
  // allocate arrays and get more info
  scatter_list_output  = (double*)malloc(num_scatter_list_out*sizeof(double));
  num_indices_accessed = (int*)malloc(num_index_info*sizeof(int));
  indices_accessed     = (int*)malloc(num_index_info*sizeof(int));
  num_indices_all      = (int*)malloc(my_space_size*sizeof(int));
  UPS_GS_Get_item( UPS_GS_NUM_INDICES_ACCESSED, gs_id,
                   num_indices_accessed );
  UPS_GS_Get_item( UPS_GS_INDICES_ACCESSED, gs_id,
                   indices_accessed );
  UPS_GS_Get_item( UPS_GS_NUM_INDICES_ALL, gs_id,
                   num_indices_all );
  // define the input buffer and scatter_list
  for( i = 0; i < count; i++ )
    {
      scatter_list_input[i] = penum;
    }
  UPS_GS_Scatter_list( scatter_list_input, scatter_list_output, UPS_DT_DOUBLE,
                       1, gs_id );
  // print out some results
  for( position = 0, i = 0; i < num_index_info; i++ )
    {
      printf( "method 1: penum %d index %d accessed %d",
              penum, indices_accessed[i], num_indices_accessed[i] );
      printf( " values: ");
      for( j = 0; j < num_indices_accessed[i]; j++ )
        {
          printf( " %f ", scatter_list_output[position++] );
        }
      printf( "\n" );
    }
  for( i = 0; i < my_space_size; i++ )
    {
      printf( "method 2: penum %d index %d accessed %d\n",
              penum, i, num_indices_all[i] );
    }
  // swap in/out buffers if you want to gather_list
  UPS_GS_Gather_list( scatter_list_output, scatter_list_input, UPS_DT_DOUBLE,
                      1, gs_id );
  UPS_GS_Free( gs_id );
  UPS_AA_Terminate( );
}
```

SeeAlso

`UPS_GS_Setup` (page 165)
`UPS_GS_Gather` (page 149)
`UPS_GS_Scatter` (page 156)
`UPS_GS_Gather_list` (page 151)
`UPS_GS_Scatter_list` (page 158)
`UPS_GS_Get_item` (page 155)

UPS_GS_Scatter_multi()

Package

gs

Purpose

UPS_GS_Scatter_multi performs UPS_GS_Scatter `multi` times given the array of `in` and `out` addresses. See the purpose section of `UPS_GS_Gather` (section C.7 page 149) for a more detailed description.

Usage

C	`ierr = UPS_GS_Scatter_multi`	`(in_multi, out_multi, datatype, GS_func, count, gs_id);`
Fortran	`call UPSF_GS_SCATTER_MULTI`	`(in_multi, out_multi, datatype, GS_func, count, gs_id, ierr)`
Fortran77	`call UPS_GS_SCATTER_MULTI`	`(in_multi, out_multi, datatype, GS_func, count, gs_id, ierr)`

Arguments

`in_multi`

Intent:	in
C type:	void**
Fortran type:	UPS_KIND_ADDRESS {0-1}
Fortran77 type:	UPS_KIND_ADDRESS {0-1}

Array of input addresses

`out_multi`

Intent:	inout
C type:	void**
Fortran type:	UPS_KIND_ADDRESS {in_multi}
Fortran77 type:	UPS_KIND_ADDRESS {in_multi}

Array of input addresses

`datatype`

Intent:	in
C type:	UPS_DT_Datatype_enum
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Datatype of the elements

`GS_func`

Intent:	in
C type:	UPS_AA_Operation_enum
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Operation to perform

`count`

Intent:	in
C type:	int
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Number of times to perform a scatter.

`gs_id`

Intent:	in
C type:	int
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The handle for the database set in a prior call to `UPS_GS_Setup`.

`ierr`

Intent:	out
C type:	(na) int return value

Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

See the discussion-section of UPS_GS_Gather_multi (page 153).

SeeAlso

`UPS_GS_Setup` (page 165)
`UPS_GS_Gather` (page 149)
`UPS_GS_Gather_multi` (page 153)
`UPS_GS_Scatter` (page 156)

UPS_GS_Setup()

Package

gs

Purpose

UPS_GS_Setup sets up a global distributed gather/scatter database and associated communication pattern based on the input list of (global) indices. The global index space must obey a simple variable block distribution, where each process owns a contiguous block of the index space. On-process references are expected as positive integers representing local indices. Off-process references are expected as negative integers representing a global index.

Usage

C
```
ierr = UPS_GS_Setup   (indices, my_space_size,
                       my_start_index,
                       start_index_type, count, gs_id);
```
Fortran
```
call UPSF_GS_SETUP    (indices, my_space_size,
                       my_start_index,
                       start_index_type, count, gs_id,
                       ierr)
```
Fortran77
```
call UPS_GS_SETUP     (indices, my_space_size,
                       my_start_index,
                       start_index_type, count, gs_id,
                       ierr)
```

Arguments

`indices`

Intent: in
C type: const int*
Fortran type: UPS_KIND_INT4 {0-1}
Fortran77 type: UPS_KIND_INT4 {0-1}
Array containing the global index number of the nodes you are interested in. Normal index values will be between the starting index of process 0 and the last index of last process. Indices outside of this range follow different

rules depending upon their values. Several consecutive
negative constants are defined as follows:
1) index < first index of process 0 + UPS_GS_INDEX_ZERO:
Gather: value does not overwrite output
Scatter: value does not contribute
2) index = first index of process 0 + UPS_GS_INDEX_ZERO:
Gather: sets output value to 0
Scatter: value does not contribute
3) index = first index of process 0 + UPS_GS_INDEX_SKIP:
Gather: value does not overwrite output
Scatter: value does not contribute
4) index > last index of last process:
Gather: value does not overwrite output
Scatter: value does not contribute

`my_space_size`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The number of indices local to the calling process.	
`my_start_index`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The value for the first index (see start_index_type).	
`start_index_type`	*Intent*:	in
	C type:	UPS_GS_Index_type_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Variable defining to what my_start_index points. (`UPS_GS_LOCAL_INDEX`) points to index owned by this PE. (`UPS_GS_GLOBAL_INDEX`) points to index owned by PE 0.	
`count`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The length of the indices array.	
`gs_id`	*Intent*:	inout
	C type:	int*
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Integer returned by ups to be used as an id for future gather/scatter operations. If given a non-0 value, UPS will try to use that ID (no guarantee).	
`ierr`	*Intent*:	out

C type:	(na) int return value
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Return UPS_OK if successful

Discussion

We illustrate the use of this function with two small examples. Suppose we have the following situation:

1. three participating processes,
2. space size for processors: 5, 10, and 15, and
3. each process wants to deal with the last index that each process owns.

The user would input the following variables:

`process`	`indices`	`count`	`my_space_size`	`my_start_index`	`start_index_type`	`gs_id`
0	4, 14, 29	3	5	0	`UPS_GS_GLOBAL_INDEX`	1
1	4, 14, 29	3	10	0	`UPS_GS_GLOBAL_INDEX`	1
2	4, 14, 29	3	15	0	`UPS_GS_GLOBAL_INDEX`	1

Another example describes the same situation but with a different starting index. Here are the variables that would be set:

`process`	`indices`	`count`	`my_space_size`	`my_start_index`	`start_index_type`	`gs_id`
0	4, 14, 29	3	5	0	`UPS_GS_LOCAL_INDEX`∗	1
1	-1, 9, 24	3	10	0	`UPS_GS_LOCAL_INDEX`	1
2	-10, 0, 15	3	15	1∗∗	`UPS_GS_LOCAL_INDEX`	1

∗ Note different starting index type.
∗∗ Note different starting index.

SeeAlso

`UPS_GS_Setup_s_global` (page 167)
`UPS_GS_Setup_s_local` (page 170)
`UPS_GS_Setup_study` (page 173)
`UPS_GS_Gather` (page 149)
`UPS_GS_Scatter` (page 156)

UPS_GS_Setup_s_global()

Package

gs

Purpose

UPS_GS_Setup_s_global is an additional method for defining the communication pattern for future gather/scatter operations.

In this method, the user specifies the global indices they wish to deal (global_indices), and then the order they will appear in their local array (local_indices).

Usage

C	`ierr = UPS_GS_Setup_s_global`	`(global_indices, my_space_size, my_start_index, start_index_type, local_indices, local_index_start, count, gs_id);`
Fortran	`call UPSF_GS_SETUP_S_GLOBAL`	`(global_indices, my_space_size, my_start_index, start_index_type, local_indices, local_index_start, count, gs_id, ierr)`
Fortran77	`call UPS_GS_SETUP_S_GLOBAL`	`(global_indices, my_space_size, my_start_index, start_index_type, local_indices, local_index_start, count, gs_id, ierr)`

Arguments

`global_indices`

Intent: in
C type: const int*
Fortran type: UPS_KIND_INT4 {0-1}
Fortran77 type: UPS_KIND_INT4 {0-1}
Array containing the global index number of the nodes you are interested in. Normal index values will be between the starting index of process 0 and the last index of last process. Indices outside of this range follow different rules depending upon their values. Several consecutive negative constants are defined as follows:
1) index < first index of process 0 + UPS_GS_INDEX_ZERO:
Gather: value does not overwrite output
Scatter: value does not contribute
2) index = first index of process 0 + UPS_GS_INDEX_ZERO:
Gather: sets output value to 0
Scatter: value does not contribute
3) index = first index of process 0 + UPS_GS_INDEX_SKIP:
Gather: value does not overwrite output
Scatter: value does not contribute
4) index > last index of last process:
Gather: value does not overwrite output
Scatter: value does not contribute

`my_space_size`

Intent: in
C type: int
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The number of indices local to the calling process.

`my_start_index`

Intent: in

	C type: int
	Fortran type: UPS_KIND_INT4 {0}
	Fortran77 type: UPS_KIND_INT4 {0}
	The value for the first index (see start_index_type).

`start_index_type`	*Intent*: in *C type*: UPS_GS_Index_type_enum *Fortran type*: UPS_KIND_INT4 {0} *Fortran77 type*: UPS_KIND_INT4 {0} Variable defining to what my_start_index points. (`UPS_GS_LOCAL_INDEX`) points to index owned by this PE. (`UPS_GS_GLOBAL_INDEX`) points to index owned by PE 0.
`local_indices`	*Intent*: in *C type*: const int* *Fortran type*: UPS_KIND_INT4 {0-1} *Fortran77 type*: UPS_KIND_INT4 {0-1} Contains the mapping of the global_indices array to the local array. All values must point to valid locations in the local array.
`local_index_start`	*Intent*: in *C type*: int *Fortran type*: UPS_KIND_INT4 {0} *Fortran77 type*: UPS_KIND_INT4 {0} The value of the first element in the local array. This provides a basis for the values in the local_indices array. So, typically, in C, this value will be 0 and in fortran, the value will be 1.
`count`	*Intent*: in *C type*: int *Fortran type*: UPS_KIND_INT4 {0} *Fortran77 type*: UPS_KIND_INT4 {0} The length of the global_indices array.
`gs_id`	*Intent*: inout *C type*: int* *Fortran type*: UPS_KIND_INT4 {0} *Fortran77 type*: UPS_KIND_INT4 {0} Integer returned by ups to be used as an id for future gather/scatter operations. If given a non-0 value, UPS will try to use that ID (no guarantee).
`ierr`	*Intent*: out *C type*: (na) int return value *Fortran type*: UPS_KIND_INT4 {0} *Fortran77 type*: UPS_KIND_INT4 {0} Return status. Returns UPS_OK if successful.

ReturnValues

Return UPS_OK if successful

Examples

The following describe what certain values would mean:

- global_indices: 200,400

  Gather-from or scatter-to the 200th and 400th global indices.
- local_indices: 100,20

  Use these indices as destinations of a gather or as inputs of a scatter.
- local_index_start: 1

  The 100 above is the 100th element. If local_index_start had been 0, 100 would have meant the 101st element.
- count: 2

  Size of global_indices (and local_indices).
- my_space_size: 300

  The 200th index is onpe, but the 400th index is offpe.
- my_start_index: 1

  The 200 above is the 200th element. If my_start_index had been 0, 200 would have been the 201st element.

SeeAlso

`UPS_GS_Setup` (page 165)
`UPS_GS_Setup_s_local` (page 170)
`UPS_GS_Setup_study` (page 173)

UPS_GS_Setup_s_local()

Package

gs

Purpose

UPS_GS_Setup_s_local is an additional method for defining the communication pattern for future gather/scatter operations.

In this method, the user specifies the indices they wish to deal with in terms of two arrays: index_pe and index_value. The order in which they will appear in their local array is specified by local_indices.

Usage

C	`ierr = UPS_GS_Setup_s_local`	`(index_pe, index_value, index_value_start, local_indices, local_index_start, count, gs_id);`
Fortran	`call UPSF_GS_SETUP_S_LOCAL`	`(index_pe, index_value, index_value_start, local_indices, local_index_start, count, gs_id, ierr)`
Fortran77	`call UPS_GS_SETUP_S_LOCAL`	`(index_pe, index_value, index_value_start, local_indices, local_index_start, count, gs_id, ierr)`

Arguments

`index_pe`

Intent: in
C type: const int*
Fortran type: UPS_KIND_INT4 {0-1}
Fortran77 type: UPS_KIND_INT4 {0-1}

This array (along with index_value below) define which indices you want to deal with. A unique index may be determined by specifying which pe owns an index (index_pe) and what the local value of that index is (index_value).
Normal index values will be between 0 through number of pe - 1. Values outside of this range follow different rules depending upon their values. Several consecutive negative constants are defined as follows:

1) index_pe < UPS_GS_INDEX_ZERO:
    Gather: value does not overwrite output
    Scatter: value does not contribute
2) index_pe = UPS_GS_INDEX_ZERO:
    Gather: sets output value to 0
    Scatter: value does not contribute
3) index = UPS_GS_INDEX_SKIP:
    Gather: value does not overwrite output
    Scatter: value does not contribute
4) index > number of processes:
    Gather: value does not overwrite output
    Scatter: value does not contribute

`index_value`

Intent: in
C type: int*
Fortran type: UPS_KIND_INT4 {index_pe}
Fortran77 type: UPS_KIND_INT4 {index_pe}

Array containing indices normalized to the pe owning the index (with the first index owned by the pe having the value index_value_start (below). This array, along with index_pe (above) determine a unique index.

`index_value_start`

Intent:	in
C type:	int
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The label for the first index owned by a pe. Normally, this will be 1 for Fortran programs and 0 for C programs.

`local_indices`

Intent:	in
C type:	const int*
Fortran type:	UPS_KIND_INT4 {0-1}
Fortran77 type:	UPS_KIND_INT4 {0-1}

Contains the mapping of the global_indices array to the local array. All values must point to valid locations in the local array.

`local_index_start`

Intent:	in
C type:	int
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The value of the first element in the local array. This provides a basis for the values in the local_indices array. So, typically, in C, this value will be 0 and in fortran, the value will be 1.

`count`

Intent:	in
C type:	int
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The length of the index arrays.

`gs_id`

Intent:	inout
C type:	int*
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Integer returned by ups to be used as an id for future gather/scatter operations.
If given a non-0 value, UPS will try to use that ID (no guarantee).

`ierr`

Intent:	out
C type:	(na) int return value
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Return UPS_OK if successful

Examples

The following describe what certain values would mean:

- index_pe: 2, 4, 2

  The three indices are owned by pe's 2, 4, and 2 respectively.
- index_value: 100, 20, 1000

  The 100th index of pe 2, the 20th index of pe 4, and the 1000th index of pe 2.
- index_value_start: 1

  The value of 100 for index_value is the 100th element owned by pe 2. If the value had been 0 for index_value_start, 100 would have meant the 101st element index owned by pe 2.
- local_indices: 6, 3, 1

  Use these indices as destinations of a gather or as inputs of a scatter.
- local_index_start: 1

  The value of 6 for local_indices is the 6th element. If the value had been 0 for local_index_start, 6 would have meant the 7th element.
- count: 3

  Size of index_pe, index_value, and local_indices.

SeeAlso

`UPS_GS_Setup` (page 165)
`UPS_GS_Setup_s_global` (page 167)
`UPS_GS_Setup_study` (page 173)

UPS_GS_Setup_study()

Package

gs

Purpose

Allow the user to define certain parameters that dictate how the GS-setup routine should operate. A call to this function dictates how all future GS-setup routines made by this process will operate. If you wish to change the behavior of setup calls for all processes, all processes must make a call to UPS_GS_Setup_study with the same settings.

Usage

C	`ierr = UPS_GS_Setup_study`	`(study_setting);`
Fortran	`call UPSF_GS_SETUP_STUDY`	`(study_setting, ierr)`
Fortran77	`call UPS_GS_SETUP_STUDY`	`(study_setting, ierr)`

Arguments

`study_setting`	*Intent*:	in
	C type:	UPS_GS_Setup_study_type_enum
	Fortran type:	UPS_KIND_INT4 {0}

	Fortran77 type: UPS_KIND_INT4 {0} Describe specifics about how UPS_GS_Setup should be conducted. See page 69: `UPS_GS_Setup_study_type_enum` for a discussion of the different options allowable.
`ierr`	*Intent*: out *C type*: (na) int return value *Fortran type*: UPS_KIND_INT4 {0} *Fortran77 type*: UPS_KIND_INT4 {0} Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful

Discussion

See `UPS_GS_Setup_study_type_enum` (page 69) for a discussion of the different options allowable.
See `UPS_AA_OPT_TYPE_enum` (page 51) for additional GS options callable from `UPS_AA_Opt_set()` (page 98) that affect these settings.
See `UPS_GS_Item_enum` (page 65) for additional options callable from `UPS_GS_Get_item()` (page 155) to see what information can be obtained about the setup call.

- Compression Performance

  Compression creates a list of unique indices from the initial list of indices given in a setup call. Finding the unique indices can be costly (and the cost grows as the square of the number of indices).

  To make compression worth it (time wise), you have to do enough gather/scatter calls per setup call. Data about how much compression was done in the setup can be obtained from the output statistics file (see UPS_AA_Statistics). The code location UPS_GS_LOCP_SETUP_COMPRESSION has the following data fields:

  – 0: Number of indices before compression
  – 1: Number of indices after compression
  – 2: compression percentage

SeeAlso

`UPS_GS_Setup` (page 165)
`UPS_GS_Setup_s_global` (page 167)
`UPS_GS_Setup_s_local` (page 170)
`UPS_GS_Setup_study` (page 173)
`UPS_GS_Get_item` (page 155)
`UPS_AA_Statistics` (page 99)
`UPS_AA_Opt_set` (page 98)

C.8 File IO

See the packages section (section 6.7, page 32 for a general description of this package.

This section contains an alphabetical listing of the io routines available in UPS.

UPS_IO_Attr_read()

Package

io

Purpose

Read an attribute from an object.

Usage

C	`ierr = UPS_IO_Attr_read`	`(attribute_name, object_name, object_id, datatype, buf);`
Fortran	`call UPSF_IO_ATTR_READ`	`(attribute_name, object_name, object_id, datatype, buf, ierr)`
Fortran77	`call UPS_IO_ATTR_READ`	`(attribute_name, object_name, object_id, datatype, buf, ierr)`

Arguments

`attribute_name`

Intent:	in
C type:	const char*
Fortran type:	UPS_KIND_CHAR {0}
Fortran77 type:	UPS_KIND_CHAR {0}

The name of the attribute.
This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`object_name`

Intent:	in
C type:	const char*
Fortran type:	UPS_KIND_CHAR {0}
Fortran77 type:	UPS_KIND_CHAR {0}

The name of the object (wrt object_id).

The path to the object is obtained from object_name relative to object_id. If object_name is ".", object_id actually points to the object.

This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`object_id`

Intent:	in
C type:	int

	Fortran type: UPS_KIND_INT4 {0}
	Fortran77 type: UPS_KIND_INT4 {0}
	An object id that qualifies location of object_name. This id can be an id obtained from IO package open functions.

`datatype`	*Intent*: in *C type*: UPS_DT_Datatype_enum *Fortran type*: UPS_KIND_INT4 {0} *Fortran77 type*: UPS_KIND_INT4 {0} Datatype of buf. If necessary, the routine UPS_UT_Dt_change is used to convert the data datatype to this datatype. Please see UPS_DT_Datatype_enum (section B page 63) for a listing/description of the possible values.
`buf`	*Intent*: out *C type*: void* *Fortran type*: user_choice {0-1:UPS_IO_Info_create} *Fortran77 type*: user_choice {0-1:UPS_IO_Info_create} The value of the attribute. The entire attribute that was written out is read into buf. There is no ability to read out a portion of an attribute. If you need to do this, you might consider writing a dataset instead. The size of the buffer that will be read (and various other items of information) may be obtained from UPS_IO_Info_item_get (section C.8 page 220).
`ierr`	*Intent*: out *C type*: (na) int return value *Fortran type*: UPS_KIND_INT4 {0} *Fortran77 type*: UPS_KIND_INT4 {0} Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

The following is a C example of writing/reading attributes associated with datasets and groups that have already been created. Please see the examples listed above for a demonstration on how these can be created.

In UPS_IO_Attr_write_s, the size of the attribute is specified by the ndims and dims arguments. In UPS_IO_Attr_write, only 1 item is written (if a string, the characters up to but not including the null-terminator are written).

All processes must have the same data (ndims, dims, buffer, ...) When writing an attribute.

C:

- Write an int attribute to an existing group with id group_id:

```
 UPS_IO_Attr_write( "io_pe", ".", group_id,
                    UPS_DT_INT, &io_pe_value );
-or-
 ndims = 1;
 dims[0] = 1;
 UPS_IO_Attr_write_s( "io_pe", ".", group_id,
                      UPS_DT_INT, ndims, dims, &io_pe_value );
```

- Write a string attribute to a named dataset "hello dataset" relative to group_id:

```
 UPS_IO_Attr_write( "hello message", "hello dataset", group_id,
                    UPS_DT_STRING, "Hello - everything is fine" );
-equivalent to-
 ndims = 1;
 dims[0] = strlen( "Hello - everything is fine" );
 UPS_IO_Attr_write_s( "hello message", "hello dataset", group_id,
                      UPS_DT_STRING, ndims, dims,
                      "Hello - everything is fine" );
```

  Note that in the above example, exactly the characters specified by dims will be written to (and thus read from) the file. If you wish to write (and hence read) a null-terminator, the string buffer needs to have a null-terminator at the end and specify a dims value with UPS_IO_Attr_write_s to include it.

- Read in a specific attribute:

  Note: as mentioned above, only the characters written will be read. Only the characters "Hello - everything is fine" are put into buf - no additional null terminator is added.

```
 UPS_IO_Attr_read( "hello message", "hello dataset", group_id,
                   UPS_DT_STRING, character_array );
```

One can get information about an attribute by using the UPS query functions:

```
UPS_IO_Info_create_self( "hello message", "hello dataset", group_id,
                         &count, &info_id );
```

If found, count will be set to 1 and info_id will be set.

If you want to get an array of info ids of all the attributes attached to an object, you can call:

```
UPS_IO_Info_count( "hello dataset", group_id,
                   UPS_IO_INFO_LIST_ATTRIBUTES, num_attrs )
UPS_IO_Info_create( "hello dataset", group_id,
                    UPS_IO_INFO_LIST_ATTRIBUTES, info_id_array )
```

This will get an array of info ids upon which you can use UPS_IO_Info_item_get to get things like the name of the attribute.

See UPS_IO_File_open (section C.8 page 198) for more examples of querying the file.

SeeAlso

`UPS_IO_Attr_read` (page 175)
`UPS_IO_Attr_write` (page 178)
`UPS_IO_Info_create` (page 214)
`UPS_IO_Info_create_self` (page 217)

UPS_IO_Attr_write()

Package

io

Purpose

Write an attribute to an object.
This is a synchronization point for all processes accessing the file.

Usage

C	`ierr = UPS_IO_Attr_write`	`(attribute_name, object_name, object_id, datatype, buf);`
Fortran	`call UPSF_IO_ATTR_WRITE`	`(attribute_name, object_name, object_id, datatype, buf, ierr)`
Fortran77	`call UPS_IO_ATTR_WRITE`	`(attribute_name, object_name, object_id, datatype, buf, ierr)`

Arguments

`attribute_name`
Intent: in
C type: const char*
Fortran type: UPS_KIND_CHAR {0}
Fortran77 type: UPS_KIND_CHAR {0}
The name of the attribute.
This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)

	name(1:name_length)//ACHAR(0)
`object_name`	*Intent*: in *C type*: const char* *Fortran type*: UPS_KIND_CHAR {0} *Fortran77 type*: UPS_KIND_CHAR {0} The name of the object (wrt object_id). The path to the object is obtained from object_name relative to object_id. If object_name is ".", object_id actually points to the object. This string must be null-terminated. As an example, Fortran users may pass in a string concatenated with the null-character: 'my string here without null terminator'//ACHAR(0) name(1:name_length)//ACHAR(0)
`object_id`	*Intent*: in *C type*: int *Fortran type*: UPS_KIND_INT4 {0} *Fortran77 type*: UPS_KIND_INT4 {0} An object id that qualifies location of object_name. This id can be an id obtained from IO package open functions.
`datatype`	*Intent*: in *C type*: UPS_DT_Datatype_enum *Fortran type*: UPS_KIND_INT4 {0} *Fortran77 type*: UPS_KIND_INT4 {0} Datatype of buf. Please see UPS_DT_Datatype_enum (section B page 63) for a listing/description of the possible values.
`buf`	*Intent*: in *C type*: const void* *Fortran type*: user_choice {0-1:UPS_IO_Info_create} *Fortran77 type*: user_choice {0-1:UPS_IO_Info_create} The value of the attribute. This argument must be the same on all processes. The buffer is a single item of type datatype. If the datatype is UPS_DT_STRING, buf must be null-terminated and the characters written to the file will be the non-null characters.
`ierr`	*Intent*: out *C type*: (na) int return value *Fortran type*: UPS_KIND_INT4 {0} *Fortran77 type*: UPS_KIND_INT4 {0} Return status. Returns UPS_OK if successful.

ReturnValues

Returns 0 if successful

Discussion

This routine makes a call UPS_IO_Attr_write_s with the ndims and dims arguments filled in. Please see UPS_IO_Attr_write_s (section C.8 page 180) for more information.

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_Attr_read` (page 175)
`UPS_IO_Attr_write` (page 178)
`UPS_IO_Attr_write_s` (page 180)
`UPS_IO_Info_create` (page 214)

UPS_IO_Attr_write_s()

Package

io

Purpose

Write an attribute to an object.
This is a synchronization point for all processes accessing the file.

Usage

C	`ierr = UPS_IO_Attr_write_s`	`(attribute_name, object_name, object_id, datatype, ndims, dims, buf);`
Fortran	`call UPSF_IO_ATTR_WRITE_S`	`(attribute_name, object_name, object_id, datatype, ndims, dims, buf, ierr)`
Fortran77	`call UPS_IO_ATTR_WRITE_S`	`(attribute_name, object_name, object_id, datatype, ndims, dims, buf, ierr)`

Arguments

`attribute_name`

Intent: in
C type: const char*
Fortran type: UPS_KIND_CHAR {0}
Fortran77 type: UPS_KIND_CHAR {0}
The name of the attribute.
This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`object_name`

Intent: in
C type: const char*
Fortran type: UPS_KIND_CHAR {0}
Fortran77 type: UPS_KIND_CHAR {0}
The name of the object (wrt object_id).

The path to the object is obtained from object_name relative to object_id. If object_name is ".", object_id actually points to the object.

This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`object_id`

Intent: in
C type: int
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
An object id that qualifies location of object_name.
This id can be an id obtained from IO package open functions.

`datatype`

Intent: in
C type: UPS_DT_Datatype_enum
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Datatype of buf.
Please see UPS_DT_Datatype_enum (section B page 63) for a listing/description of the possible values.

`ndims`

Intent: in
C type: int
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Number of dimensions of the attribute.
This argument must be the same on all processes.

`dims`

Intent: in

	C type:	long long*
	Fortran type:	UPS_KIND_INT8 {0-1:ndims}
	Fortran77 type:	UPS_KIND_INT8 {0-1:ndims}
	The size of each dimension. This argument must be the same on all processes.	
`buf`	*Intent*:	in
	C type:	const void*
	Fortran type:	user_choice {0-1:UPS_IO_Info_create}
	Fortran77 type:	user_choice {0-1:UPS_IO_Info_create}
	The value of the attribute. This argument must be the same on all processes. The data written to the file is dictated by the ndims and dims arguments.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns 0 if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_Attr_read` (page 175)
`UPS_IO_Attr_write` (page 178)
`UPS_IO_Attr_write_s` (page 180)
`UPS_IO_Info_create` (page 214)

UPS_IO_Dataset_read()

Package

io

Purpose

Reads subset (described by info_data_id) of a dataset.

Usage

C	`ierr = UPS_IO_Dataset_read`	`(name, group_id, datatype, info_data_id, buf);`
Fortran	`call UPSF_IO_DATASET_READ`	`(name, group_id, datatype, info_data_id, buf, ierr)`
Fortran77	`call UPS_IO_DATASET_READ`	`(name, group_id, datatype, info_data_id, buf, ierr)`

Arguments

`name`

Intent: in
C type: const char*
Fortran type: UPS_KIND_CHAR {0}
Fortran77 type: UPS_KIND_CHAR {0}
The name of the dataset (wrt group_id).

The path to the dataset is obtained from name relative to group_id.

This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`group_id`

Intent: in
C type: int
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
A group id that qualifies location of object_name.
This id can be an id obtained from IO package open functions.

`datatype`

Intent: in
C type: UPS_DT_Datatype_enum
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Datatype of buf.
If necessary, the routine UPS_UT_Dt_change is used to convert the data datatype to this datatype.
Please see UPS_DT_Datatype_enum (section B page 63) for a listing/description of the possible values.

`info_data_id`

Intent: in

	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Id of info struct describing how the dataset is distributed. See UPS_IO_Info_create (section C.8 page 214) for more information.	
`buf`	*Intent*:	out
	C type:	void*
	Fortran type:	user_choice {0-4:UPS_IO_Info_create}
	Fortran77 type:	user_choice {0-4:UPS_IO_Info_create}
	The values of the dataset. If a string, the buffer will not have any additional null-terminator written to the buffer. The length may be obtained from UPS_IO_Info_item_get (section C.8 page 220).	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns 0 if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

A "dataset" is logically n-dimensional. However, in memory the data is laid out contiguously in one dimension. In the examples section, I demonstrate what information each process must set to describe dataset and the order in which UPS expects to see dataset values.

When reading a dataset, the default behavior is to have each process read in the same dataset section that was written by the corresponding pe. When a user writes a dataset, additional information is written to the file describing which proceses wrote which part of the dataset. When it is not possible for the processes to read in what the corresponding process wrote out (eg the number of processes has changed from write to read), the user must specify what part of the dataset to read. Before a read, the user has the ability to to get information about the dataset and modify what will be read in. See the examples below.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

An example...whew!, suppose you have a 1d array of global node ids where each node id corresponds to 2d x/y/z coordinate array. Suppose, then, there are 7 nodes and 3 processes. One might split up the nodes such that process 0 "owns" the first 3, process 1 owns the next 2, and process 2 owns the last 2.

```
Node IDs         Coordinates
   1           1.1, 1.2, 1.3---|--> process 0
   2           2.1, 2.2, 2.3   |
   3           3.1, 3.2, 3.3---|

   4           4.1, 4.2, 4.3---|--> process 1
   5           5.1, 5.2, 5.3---|

   6           6.1, 6.2, 6.3---|--> process 2
   7           7.1, 7.2, 7.3---|
```

- Specifying UPS_IO_INFO_NDIMS and UPS_IO_INFO_DIMS (these must always be specified)

  When the data is distributed contiguously in processor order, only the number of dimensions of the dataset and the dimensions local to the process need be defined.

  When writing the Node IDs, the arguments will be:

```
All:        ndims_nid   = 1
Process 0: dims_nid    = [3]
           buf_nid[]   = [1,2,3]
Process 1: dims_nid    = [2]
           buf_nid[]   = [4,5]
Process 2: dims_nid    = [2]
           buf_nid[]   = [6,7]
```

  When writing the Coordinates, the arguments will be:

```
All:       ndims_coord = 2
Process 0: dims_coord  = [3,3]
           buf_coord[] = [1.1,1.2,1.3,2.1,2.2,2.3,3.1,3.2,3.3]
Process 1: dims_coord  = [2,3]
           buf_coord[] = [4.1,4.2,4.3,5.1,5.2,5.3]
Process 2: dims_coord  = [2,3]
           buf_coord[] = [6.1,6.2,6.3,7.1,7.2,7.3]
```

  Note that all dimensions except for the first dimension must be the same. For processes not writing any data, they would set the dims of the first dimension to be 0.

  Also note the ordering of the buffer data (buff_nid and buff_coord). When traveling through memory, the last dimension (in the buff_coord case, the "column number") moves fastest. The user must ensure that the memory layout of their buffer is consistent with what UPS expects.

  Now, what does this look like in a program? We do the following steps:

1. Create two info_ids that contain information about how the datasets are distributed. To create empty infos, supply "" as a name to UPS_IO_Info_create (and the group_id argument will not be used).

   ```
   UPS_IO_Info_create( "", 0, UPS_IO_INFO_DATA_DIST,
                       &info_id_nid );
   UPS_IO_Info_create( "", 0, UPS_IO_INFO_DATA_DIST,
                       &info_id_coord );
   ```

2. Set ndims and dims info in the info_id's.
   Remember, due to the assumed data layout, each process only need know its local sizes.

   ```
   UPS_IO_Info_item_set( info_id_nid,   UPS_IO_INFO_NDIMS,
                         &ndims_nid   );
   UPS_IO_Info_item_set( info_id_nid,   UPS_IO_INFO_DIMS,
                         dims_nid     );
   UPS_IO_Info_item_set( info_id_coord, UPS_IO_INFO_NDIMS,
                         &ndims_coord );
   UPS_IO_Info_item_set( info_id_coord, UPS_IO_INFO_DIMS,
                         dims_coord   );
   ```

   Note, if dims/ndims were the same for both datasets, we could have just created 1 info_id and used it multiple times in UPS_IO_Dataset_write. Since the datasets have different sizes, we must create a new info_id.

3. Now, we're ready to write the datasets (into opened group_id):

   ```
   UPS_IO_Dataset_write( "node ids",    group_id, UPS_DT_INT,
                         info_id_nid,   buf_nid );
   UPS_IO_Dataset_write( "coordinates", group_id, UPS_DT_DOUBLE,
                         info_id_coord, buf_coord );
   ```

4. Free the infos created:

   ```
   UPS_IO_Info_free( 1, info_id_nid );
   UPS_IO_Info_free( 1, info_id_coord );
   ```

- Specifying UPS_IO_INFO_STARTS

  Although not needed in this example due to the data layout, one can manually specify where each process will start writing its data. Setting UPS_IO_INFO_STARTS will override the internal default settings.

  Note: UPS_IO_INFO_NDIMS and UPS_IO_INFO_DIMS must still be defined in the same way as the above example.

  When writing the Node IDs, the starting positions will be:

  ```
  Process 0: starts_nid   = [0]
  Process 1: starts_nid   = [3]
  Process 2: starts_nid   = [5]
  ```

When writing the Coordinates, the starting positions will be:

```
Process 0: starts_coord = [0,0]
Process 1: starts_coord = [3,0]
Process 2: starts_coord = [5,0]
```

When setting information, make the following calls in addition to the ones setting UPS_IO_INFO_NDIMS and UPS_IO_INFO_DIMS:

```
UPS_IO_Info_item_set( info_id_nid,   UPS_IO_INFO_STARTS,
                      &starts_nid   );
UPS_IO_Info_item_set( info_id_coord, UPS_IO_INFO_STARTS,
                      &starts_coord );
```

- Specifying UPS_IO_INFO_PGRID_DIMS and UPS_IO_INFO_PGRID_ORDER

  Although not needed in this example due to the data layout, one can specify a process grid which defines how a dataset is split among the processes. These settings are superseded if UPS_IO_INFO_STARTS is also defined.

  Note: UPS_IO_INFO_NDIMS and UPS_IO_INFO_DIMS must still be defined in the same way as the first example.

  UPS_IO_INFO_PGRID_DIMS must have the same dimensionality as UPS_IO_INFO_DIMS. You are defining the number of processes along each dimension.

  UPS_IO_INFO_PGRID_ORDER is of size Product(UPS_IO_INFO_PGRID_DIMS) and defines the ordering of the processes in the process grid. The default process ordering is the same as data value ordering - along the last dimension first. So, in this example it is not needed but is defined anyway.

  When writing the Node IDs, the process grid values must be set by all processes and will be:

```
All processes: pgrid_dims_nid    = [3]
               pgrid_order_nid   = [0,1,2]
```

  When writing the Coordinates, the process grid values must be set by all processes and will be:

```
All processes: pgrid_dims_coord  = [3,1]
               pgrid_order_coord = [0,1,2]
```

  When setting information, make the following calls in addition to the ones setting UPS_IO_INFO_NDIMS and UPS_IO_INFO_DIMS:

```
UPS_IO_Info_item_set( info_id_nid,   UPS_IO_INFO_PGRID_DIMS,
                      &pgrid_dims_nid    );
UPS_IO_Info_item_set( info_id_nid,   UPS_IO_INFO_PGRID_ORDER,
                      &pgrid_order_nid   );
UPS_IO_Info_item_set( info_id_coord, UPS_IO_INFO_PGRID_DIMS,
                      &pgrid_dims_coord  );
```

```
UPS_IO_Info_item_set( info_id_coord, UPS_IO_INFO_PGRID_ORDER,
                      &pgrid_order_coord );
```

O.K., now there are a couple of dataset written. How would you read them back in? With the following calls:

1. Close the above file and open it again for reading.
2. Define a couple of infos describing how the dataset is distributed. As we are reading in the same way we wrote them out, we can call UPS_IO_Info_create and create filled-infos.

   ```
   UPS_IO_Info_create( "node ids",    group_id,
                       UPS_IO_INFO_DATA_DIST, &info_id_nid   );
   UPS_IO_Info_create( "coordinates", group_id,
                       UPS_IO_INFO_DATA_DIST, &info_id_coord );
   ```

   Whalla - values for internal IO package data is set correctly to read in the dataset just as it was written out.

   Note: You can get the info id associated with a dataset in various ways:

   - Use UPS_IO_Info_create as above
   - Use a similar UPS function

     ```
     UPS_IO_Info_create_self( "", "node ids", group_id, &count,
                              &info_id_nid );
     ```

     If found, count will be set to 1 and info_id_nid will be the info id.
   - Get a listing of the members of a group

     ```
     UPS_IO_Info_count( ".", group_id, UPS_IO_INFO_LIST_MEMBERS,
                        num_info_ids )
     UPS_IO_Info_create( ".", group_id, UPS_IO_INFO_LIST_MEMBERS,
                         info_id_array )
     ```

     This will get an array of info ids upon which you can use UPS_IO_Info_item_get to get things like name and object type (dataset or group).

   See UPS_IO_File_open (section C.8 page 198) for more examples of querying the file.

   As mentioned in the discussion section above, the default is to read in how it was written. If this is not possible (eg the number of processes reading does not equal the number of processes writing), the user must specify what part of the dataset to read - as in the next example.

   If you wish to read in a different dataset section than what is defaulted, for example you wish to read in the entire dataset even though you are reading with the same number of pes that were used for writing, you must do the following after calling UPS_IO_Info_create:

   (a) Get dims_total dataset info:

   ```
   UPS_IO_Info_item_get( info_id_coord, UPS_IO_INFO_DIMS_TOTAL,
                         dims_total );
   ```

(b) reset dims to dims_total and starts to start at beginning:

```
UPS_IO_Info_item_set( info_id_coord, UPS_IO_INFO_DIMS,
                      dims_total );
starts[0] = 0;
starts[1] = 0;
UPS_IO_Info_item_set( info_id_coord, UPS_IO_INFO_STARTS,
                      starts );
```

3. Read the datasets:

```
UPS_IO_Dataset_read( "node ids",    group_id, UPS_DT_INT,
                     info_id_nid,   buf_nid );
UPS_IO_Dataset_read( "coordinates", group_id, UPS_DT_DOUBLE,
                     info_id_coord, buf_coord );
```

4. Free the infos created:

```
UPS_IO_Info_free( 1, info_id_nid );
UPS_IO_Info_free( 1, info_id_coord );
```

Note, UPS_IO_Info_create (and then UPS_IO_Info_item_get) can be used to query existing datasets for information (just as shown in UPS_IO_Attr_read).

Sequential File Access

The UPS IO routines are designed so that the file objects (groups, datasets, and attributes) are accessed via names. This way, you can move around the file quickly and the file itself is self descriptive (hopefully).

However, if you wish to operate in a sequential mode, you may. The basic operating procedure would be:

1. Open file for creation
2. Write dataset, write dataset, write dataset, ...
3. Close file
4. Open file for read
5. Read dataset, read dataset, read dataset, ...
   The first dataset read corresponds to the first dataset written
6. Close file

No names of datasets are given to the write or the read calls. Internally, UPS keeps track of which dataset is being written or read and then increments a simple counter when the write or read is finished.

Through calls to UPS_IO_Loc_item_get, the user can obtain the name of the dataset and then use UPS_IO_Info_create with that name to get information about the dataset. This is not required.

The following is an example of sequential write to a file. In this example, only a single dset_id is created because the datasets have the same layout. Note that names are not supplied to the write or read calls.

```
UPS_IO_File_open( "seq_file.h5", UPS_IO_OPEN_CREATE, &file_id );
UPS_IO_Info_create( "", file_id, UPS_IO_INFO_DATA_DIST, &dset_id );
UPS_IO_Info_item_set( dset_id, UPS_IO_INFO_NDIMS, ndims );
UPS_IO_Info_item_set( dset_id, UPS_IO_INFO_DIMS, dims );
UPS_IO_Dataset_write( "", file_id, UPS_DT_DOUBLE, dset_id, first_dset );
UPS_IO_Dataset_write( "", file_id, UPS_DT_DOUBLE, dset_id, second_dset );
UPS_IO_Dataset_write( "", file_id, UPS_DT_DOUBLE, dset_id, third_dset );
UPS_IO_File_close( file_id );
```

In the read, I demonstrate how you can get the name of the dataset and feed that into UPS_IO_Info_create to get an info_id that has information about the dataset. This makes it easier to read if you are reading exactly how you wrote out. This is not necessary. You could create an empty info_id, fill it with the correct ndims, dims, and starts, then read the dataset.

Again, since the three datasets have the same layout, I can reuse the same dset_id. If they did not, I would call UPS_IO_Info_create before each read.

```
UPS_IO_File_open( "seq_file.h5", UPS_IO_OPEN_READ, &file_id );
UPS_IO_Loc_item_get( file_id, UPS_IO_LOC_DS_NEXT_R, dset_name );
UPS_IO_Info_create( dset_name, file_id, UPS_IO_INFO_DATA_DIST,
                    &dset_id );
UPS_IO_Dataset_read( "", file_id, UPS_DT_DOUBLE, dset_id, first_dset );
UPS_IO_Dataset_read( "", file_id, UPS_DT_DOUBLE, dset_id, second_dset );
UPS_IO_Dataset_read( "", file_id, UPS_DT_DOUBLE, dset_id, third_dset );
UPS_IO_File_close( file_id );
```

Sequential access of a file is not encouraged because such files are not as self descriptive as files with an organized structure and whose object names mean something.

Creating a Dataset from Pieces

There may be times when you wish to create a global dataset but you only have access to a subset of the pieces at one time. You can call UPS_IO_Dataset_write() multiple times with the following caveats:

- UPS_IO_Dataset_write() is a collective call.

  All processes must still call UPS_IO_Dataset_write. If a process has no data, set UPS_IO_INFO_DIMS to 0s.

- You must set UPS_IO_INFO_NDIMS, UPS_IO_INFO_DIMS, UPS_IO_INFO_STARTS and UPS_IO_INFO_DIMS_TOTAL.

  You cannot use the default or UPS_IO_INFO_PGRID settings and you must know the total size of the dataset beforehand.

- The datatype, UPS_IO_INFO_NDIMS, UPS_IO_INFO_DIMS_TOTAL, cannot change.

- You must set UPS_IO_INFO_DIMS and UPS_IO_INFO_STARTS when reading the dataset back.

  Saying "must" is a little strong..."should" perhaps is better. Using the underlying protocol UPS_IO_PROTOCOL_HDF, the default values gotten from UPS_IO_Info_create() ("what this process wrote is what it reads") will get each process only the data it wrote during the last UPS_IO_Dataset_write(). However, other protocols might behave differently.

So, better safe than sorry - set dims and starts manually.

SeeAlso

`UPS_IO_Dataset_read` (page 183)
`UPS_IO_Dataset_write` (page 191)
`UPS_IO_Info_create` (page 214)
`UPS_IO_Loc_item_get` (page 224)
`UPS_IO_Loc_item_set` (page 225)

UPS_IO_Dataset_write()

Package

io

Purpose

Writes subset (described by info_data_id) of a dataset.
This is a synchronization point for all processes accessing the file.

Usage

C	`ierr = UPS_IO_Dataset_write`	`(name, group_id, datatype, info_data_id, buf);`
Fortran	`call UPSF_IO_DATASET_WRITE`	`(name, group_id, datatype, info_data_id, buf, ierr)`
Fortran77	`call UPS_IO_DATASET_WRITE`	`(name, group_id, datatype, info_data_id, buf, ierr)`

Arguments

`name`

Intent:	in
C type:	const char*
Fortran type:	UPS_KIND_CHAR {0}
Fortran77 type:	UPS_KIND_CHAR {0}

The name of the dataset (wrt group_id).

The path to the dataset is obtained from name relative to group_id.

This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`group_id`

Intent:	in
C type:	int
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

A group id that qualifies location of object_name.
This id can be an id obtained from IO package open functions.

`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Datatype of buf. Please see UPS_DT_Datatype_enum (section B page 63) for a listing/description of the possible values.	
`info_data_id`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Id of info struct describing how the dataset is distributed. See UPS_IO_Info_create (section C.8 page 214) for more information.	
`buf`	*Intent*:	in
	C type:	const void*
	Fortran type:	user_choice {0-4:UPS_IO_Info_create}
	Fortran77 type:	user_choice {0-4:UPS_IO_Info_create}
	The values of the dataset. If a string, the buffer is not null-terminated. The size is dictated by dims of info_data_id.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

Notes

Please see UPS_IO_ACCESS_PES_enum (section B page 70) for environment variables that affect this call.

SeeAlso

`UPS_IO_Dataset_read` (page 183)
`UPS_IO_Dataset_write` (page 191)
`UPS_IO_Info_create` (page 214)
`UPS_IO_Loc_item_get` (page 224)
`UPS_IO_Loc_item_set` (page 225)

UPS_IO_Ds_r_s()

Package

io

Purpose

Wrap several UPS calls to allow for easier reading/writing of a globally distributed array dataset in which every process owns a single value.

This is a synchronization point for all processes accessing the file.

Usage

C	`ierr = UPS_IO_Ds_r_s`	`(name, group_id, datatype, buf);`
Fortran	`call UPSF_IO_DS_R_S`	`(name, group_id, datatype, buf, ierr)`
Fortran77	`call UPS_IO_DS_R_S`	`(name, group_id, datatype, buf, ierr)`

Arguments

`name`
Intent: in
C type: const char*
Fortran type: UPS_KIND_CHAR {0}
Fortran77 type: UPS_KIND_CHAR {0}
The name of the dataset (wrt group_id).

The path to the dataset is obtained from name relative to group_id.

This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`group_id`
Intent: in
C type: int
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}

A group id that qualifies location of object_name.
This id can be an id obtained from IO package open functions.

`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Datatype of buf. Please see UPS_DT_Datatype_enum (section B page 63) for a listing/description of the possible values.	
`buf`	*Intent*:	out
	C type:	void*
	Fortran type:	user_choice {0}
	Fortran77 type:	user_choice {0}
	The single value. The process will read what the corresponding process wrote. An error occurs if called with a different number of processes.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Upon writing, an array will be created consisting of each value from each process.

Upon reading, each process will read in the value it wrote out. An error occurs if reading with a different number of processes that were used to write. In this case, one must use the general UPS_IO_Dataset_read and specify what exactly should be read.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

Notes

This routine basically wraps several UPS calls:

1. UPS_IO_Info_create

   Get info id

2. UPS_IO_Dataset_read

   Read in the one value this pe wrote

3. UPS_IO_Info_free

   Free the info id created

Please see UPS_IO_ACCESS_PES_enum (section B page 70) for environment variables that affect this call.

SeeAlso

`UPS_IO_Dataset_read` (page 183)
`UPS_IO_Dataset_write` (page 191)
`UPS_IO_Ds_r_s` (page 193)
`UPS_IO_Ds_w_s` (page 195)

UPS_IO_Ds_w_s()

Package

io

Purpose

Wrap several UPS calls to allow for easier reading/writing of a globally distributed array dataset in which every process owns a single value.

This is a synchronization point for all processes accessing the file.

Usage

C	`ierr = UPS_IO_Ds_w_s`	`(name, group_id, datatype, buf);`
Fortran	`call UPSF_IO_DS_W_S`	`(name, group_id, datatype, buf, ierr)`
Fortran77	`call UPS_IO_DS_W_S`	`(name, group_id, datatype, buf, ierr)`

Arguments

`name`	*Intent*:	in
	C type:	const char*
	Fortran type:	UPS_KIND_CHAR {0}
	Fortran77 type:	UPS_KIND_CHAR {0}

The name of the dataset (wrt group_id).

The path to the dataset is obtained from name relative to group_id.

This string must be null-terminated.

As an example, Fortran users may pass in a string
concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`group_id`

Intent:	in
C type:	int
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

A group id that qualifies location of object_name.
This id can be an id obtained from IO package open
functions.

`datatype`

Intent:	in
C type:	UPS_DT_Datatype_enum
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Datatype of buf.
Please see UPS_DT_Datatype_enum (section B page 63)
for a listing/description of the possible values.

`buf`

Intent:	in
C type:	const void*
Fortran type:	user_choice {0}
Fortran77 type:	user_choice {0}

The single value.
If the datatype is UPS_DT_STRING, buf must be
null-terminated and the characters written to
the file will be the non-null characters.

`ierr`

Intent:	out
C type:	(na) int return value
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Upon writing, an array will be created consisting of each value from each process.

Upon reading, each process will read in the value it wrote out. An error occurs if reading with a different number of processes that were used to write. In this case, one must use the general UPS_IO_Dataset_read and specify what exactly should be read.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

Notes

This routine basically wraps several UPS calls:

1. UPS_IO_Info_create/UPS_IO_Info_item_set

   Get info id and set values so that each process writes a single value

2. UPS_IO_Dataset_write

   Write out the one value

3. UPS_IO_Info_free

   Free the info id created

Please see UPS_IO_ACCESS_PES_enum (section B page 70) for environment variables that affect this call.

SeeAlso

`UPS_IO_Dataset_read` (page 183)
`UPS_IO_Dataset_write` (page 191)
`UPS_IO_Ds_r_s` (page 193)
`UPS_IO_Ds_w_s` (page 195)

UPS_IO_File_close()

Package

io

Purpose

Close a file_id opened with UPS_IO_File_open. This must be called after closing all group_ids opened with the file_id.

This is a synchronization point for all processes accessing the file.

Usage

C	`ierr = UPS_IO_File_close`	`(file_id);`
Fortran	`call UPSF_IO_FILE_CLOSE`	`(file_id, ierr)`
Fortran77	`call UPS_IO_FILE_CLOSE`	`(file_id, ierr)`

Arguments

`file_id`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}

	File id obtained by UPS_IO_File_open.
`ierr`	*Intent*: out *C type*: (na) int return value *Fortran type*: UPS_KIND_INT4 {0} *Fortran77 type*: UPS_KIND_INT4 {0} Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

File ids obtained by UPS_IO_File_open must be closed by UPS_IO_File_close. Group ids obtained by UPS_IO_Group_open must be closed by UPS_IO_Group_close.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_File_close` (page 197)
`UPS_IO_File_open` (page 198)
`UPS_IO_Group_close` (page 209)
`UPS_IO_Group_open` (page 210)
`UPS_IO_Info_create` (page 214)

UPS_IO_File_open()

Package

io

Purpose

Opens the file and returns a file_id.

This is a synchronization point for all processes accessing the file.

Usage

C	`ierr = UPS_IO_File_open`	`(name, open_method, file_id);`
Fortran	`call UPSF_IO_FILE_OPEN`	`(name, open_method, file_id, ierr)`
Fortran77	`call UPS_IO_FILE_OPEN`	`(name, open_method, file_id, ierr)`

Arguments

`name`

Intent: in
C type: const char*
Fortran type: UPS_KIND_CHAR {0}
Fortran77 type: UPS_KIND_CHAR {0}
The name of the file.

This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`open_method`

Intent: in
C type: UPS_IO_OPEN_METHOD_enum
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Types of opening (ie open for read, create, ...)
Please see UPS_IO_OPEN_METHOD_enum (section B page 82)

`file_id`

Intent: inout
C type: int*
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
File id. This id may be passed to UPS_IO_Group_open calls.
This id must be closed with UPS_IO_File_close.
By default, UPS sets this value and will make it the same value as returned by the underlying protocol (eg an HDF location id).
You may use your own non-negative value by calling UPS_AA_Opt_set with UPS_IO_OPT_LOC_ID_USERS set to UPS_DT_TRUE.

`ierr`

Intent: out
C type: (na) int return value
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Return status. Returns UPS_OK if successful.

ReturnValues

Returns 0 if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

File ids obtained by UPS_IO_File_open must be closed by UPS_IO_File_close. Group ids obtained by UPS_IO_Group_open must be closed by UPS_IO_Group_close.

HIGH LEVEL DISCUSSION

The following is a basic discussion of IO package file. I'll start with a definition of objects.

Objects: All objects are named (and thus referenced) by a string. A file contains the following basic objects:

1. **Groups**

   Groups provide a way to structure a file. Much the same as Unix directories, groups are containers for datasets (see below) and, in fact, other groups. Thus, a "directory structure" can be created.

   After opening, an integer id is returned and is used as a "handle" to that group. This id is used in other IO package functions.

   In general a file is a special case of a group (and thus on opening has an associated ID that can be used like a group id).

2. **Datasets**

   A Dataset, as the name suggests, is a collection of data. Typically, this is an array that was distributed among the processes (although it can be data that only exists on a subset of the process set).

   Datasets exist in groups (similar to Unix files existing in a directory).

3. **Attributes**

   Attributes are smaller, less robust datasets that are "attached" to a group or dataset (as opposed to existing autonomously in a group). They allow you to describe the object they are attached to. For example, you could create a dataset called "coordinates" and attach a string attributed named "units" with a value of "fathoms".

With the above objects, you can create structured, self-describing files.

Many of the IO package calls are synchronization points (all processes must call them and the order of the calls to synchronization points must be the same). For the most part, an IO package function will be a synchronization point whenever the file changes (eg writing a dataset or creating a group). Even if, for example, a process is not contributing data to a dataset write, it still must make the call to UPS_IO_Dataset_write.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

The following shows how to create a "directory structure" in a file. The same type of thing can be done to traverse an existing directory structure.

1. Set options dictating how the file will be used:

   Options are set via a call to UPS_AA_Opt_set (section C.2 page 98). An example is:

   ```
   MPI_Info mpi_info;
   UPS_IO_ACCESS_PES_enum file_access;
   ierr += MPI_Info_create( &mpi_info );
   ierr += MPI_Info_set( mpi_info, "striping_factor", "64" );
   ierr += UPS_AA_Opt_set( UPS_IO_OPT_INFO, &mpi_info );
   file_access = UPS_IO_ACCESS_IO_PE;
   ierr += UPS_AA_Opt_set( UPS_IO_OPT_ACCESS_WRITE, &file_access );
   assert( !ierr );
   ```

2. Open a file:

   ```
   UPS_IO_File_open( "my_file.dat", UPS_IO_OPEN_CREATE, &file_id );
   ```

3. Open a group in the file:

   ```
   UPS_IO_Group_open( "group_a", file_id, UPS_IO_OPEN_CREATE, &group_a_id );
   ```

4. Open another group in the file:

   ```
   UPS_IO_Group_open( "group_b", file_id, UPS_IO_OPEN_CREATE, &group_b_id );
   ```

5. Open a subgroup under group_a:

   ```
   UPS_IO_Group_open( "group_a_sub", group_a_id, UPS_IO_OPEN_CREATE,
                      &group_a_sub_id );
   ```

6. When finished, close everything up (with the file close last)

   ```
   UPS_IO_Group_close( group_a_sub_id );
   UPS_IO_Group_close( group_b_id );
   UPS_IO_Group_close( group_a_id );
   UPS_IO_File_close( file_id );
   ```

Now that the file exists, one can open the file/groups by doing the following (note the change of the argument UPS_IO_OPEN_CREATE to UPS_IO_OPEN_READ):

```
UPS_IO_File_open( "my_file.dat", UPS_IO_OPEN_READ, &file_id );
UPS_IO_Group_open( "group_a", file_id, UPS_IO_OPEN_READ, &group_a_id );
```

If the directory structure is unknown, queries can be made about the file. In the following examples, when the file was created a dataset with an attribute was also created. This will help in demonstrating how the count and create calls are used.

- **Getting the members of a group**

  In this example, an array of info ids containing information about the members of a group is obtained. Then each member is processed.

  UPS_IO_Info_count is used for getting the number of info ids that will be created when calling UPS_IO_Info_create.

  Note: name_array[i] is a pointer to a string that is long enough to hold the name of the i'th attribute plus a null terminator. Also, it has been initialized to null since the name returned by UPS_IO_Info_item_get is not null-terminated. Thus, this name can be passed to other functions that expect the name to be null terminated.

  ```
      UPS_IO_Info_count( ".", group_a_id, UPS_IO_LIST_MEMBERS,
                         &count );
      UPS_IO_Info_create( ".", group_a_id, UPS_IO_LIST_MEMBERS,
                          info_id_array );
      for( i = 0; i < count; i++ )
        {
          UPS_IO_Info_item_get( info_id_array[i], UPS_IO_INFO_NAME,
                                name_array[i] );
          printf( "Member %d name is %s.\n", i, name_array[i] );
          UPS_IO_Info_item_get( info_id_array[i], UPS_IO_INFO_OBJECT_TYPE,
                                object_type_array[i] );
          if( object_type_array[i] == UPS_IO_FILE_OBJECT_GROUP )
            {
              printf( "And it was a group.\n" );
            }
          if( object_type_array[i] == UPS_IO_FILE_OBJECT_DATASET )
            {
              printf( "And it was a dataset.\n" );
            }
        }
      UPS_IO_Info_free( count, info_id_array );
  ```

- **Getting the attributes of a dataset**

  In this case, the object_name and object_id arguments to UPS_IO_Info_count/UPS_IO_Info_create point to a specific dataset.

  Note: when reading an attribute, the entire attribute is read into the buffer. There is no way to read part of an attribute.

  Note: when reading a string attribute, only the exact number of characters that were written are read back in. Unless an explicit size that included the null terminator (UPS_IO_Attr_write_s) was used to write the attribute, the null terminator was not written to the attribute. Thus, space was allocated for 1 more than number_of_items and that last character was initialized to the null terminator in order to get the printf to work.

```
UPS_IO_Info_count( "hello dataset", group_id,
                   UPS_IO_INFO_LIST_ATTRIBUTES, &count );
UPS_IO_Info_create( "hello dataset", group_id,
                     UPS_IO_INFO_LIST_ATTRIBUTES, info_id_array);
for( i = 0; i < count; i++ )
  {
    UPS_IO_Info_item_get( info_id_array[i], UPS_IO_INFO_NAME,
                          name_array[i] );
    printf( "Attribute %d name is %s.\n", i, name_array[i] );
    UPS_IO_Info_item_get( info_id_array[i], UPS_IO_INFO_DATATYPE,
                          &datatype );
    UPS_IO_Info_item_get( info_id_array[i], UPS_IO_INFO_NITEMS,
                          &number_of_items );
    if( datatype == UPS_DT_STRING )
      {
        character_array = calloc(number_of_items + 1, sizeof( char ));
        printf( "The datatype is an array of characters.\n" );
        UPS_IO_Attr_read( name_array[i], "hello dataset", group_id,
                          UPS_DT_STRING, character_array );
        printf( "With a value of %s.\n", character_array );
      }
    else
      {
         // error code for invalid datatype could go here
      }
  }
UPS_IO_Info_free( count, info_id_array );
```

- **Using a filter to examine a specific attribute**
  One can use filters to limit the number of matches in UPS_IO_Info_count/UPS_IO_Info_create.

```
UPS_IO_Filter_set( "hello message", UPS_IO_FILTER_INFO );
UPS_IO_Info_count( "hello dataset", group_id,
                   UPS_IO_INFO_LIST_ATTRIBUTES, &count );
assert( count == 1 );
UPS_IO_Info_create( "hello dataset", group_id,
                    UPS_IO_INFO_LIST_ATTRIBUTES,
                    &single_info_id );
UPS_IO_Info_item_get( single_info_id, UPS_IO_INFO_NITEMS,
                      num_chars );
printf( "The 'hello message' string attribute has %lli chars.",
        num_chars );
UPS_IO_Info_free( count, &single_info_id );
UPS_IO_Filter_set( "", UPS_IO_FILTER_INFO_SET );
```

Note: in the above example, the call to UPS_IO_Info_count is done merely to verify a count of 1. If you REALLY know the attribute is there, you can just assume a count of 1 and go straight to the UPS_IO_Info_create call.

Also, the UPS_IO_Filter command lasts until reset by the next UPS_IO_Filter call (and is removed with a null-terminated string of length 0 (C: "", Fortran: ACHAR(0)).

For the same effect, one could use UPS_IO_Info_create_self:

```
UPS_IO_Info_create_self( "hello message", "hello dataset", group_id,
                         &count, &single_info_id );
UPS_IO_Info_free( count, &single_info_id );
```

So, why have both ways (create self and filtering)? There are more complex filters one can do (see the next example).

- **Getting all info_ids recursively**

  One can get a listing of all the members (NOT attributes) recursively from a particular object_name+object_id.

  ```
  UPS_IO_Filter_set( "* /", UPS_IO_FILTER_INFO );
  UPS_IO_Info_count( "group_a", file_id,
                     UPS_IO_INFO_LIST_MEMBERS, &num_recursive_members );
  UPS_IO_Info_create( "group_a", file_id,
                      UPS_IO_INFO_LIST_MEMBERS,
                      info_id_array );
  UPS_IO_Info_free( num_recursive_members, info_id_array );
  UPS_IO_Filter_set( "", UPS_IO_FILTER_INFO_SET );
  ```

  Note: there is not space between "*" and "/".

- **Getting specific info_ids recursively**

  One can get a listing of all the members of (NOT attributes) of a specific name recursively from a particular object_name+object_id.

  ```
  UPS_IO_Filter_set( "* /hello dataset", UPS_IO_FILTER_INFO );
  UPS_IO_Info_count( ".", file_id,
                     UPS_IO_INFO_LIST_MEMBERS, &num_matches );
  UPS_IO_Filter_set( "", UPS_IO_FILTER_INFO_SET );
  ```

  Note: there is not space between "*" and "/".

  This example returns the number of members named "hello dataset" found recursively from the top level directory.

Removing objects from the file

If you wish to remove an object from the file, use UPS_IO_Rm. This call is recursive. The following will remove group_a and all of its contents recursively. In this case, it is just sub_group_a. If sub_group_a also had objects beneath it, they would be removed from the file as well:

```
UPS_IO_Rm( "", ".", group_a_id );
```

Which is the same as

```
UPS_IO_Rm( "", "group_a", file_id );
```

The first argument to UPS_IO_Rm is for removing specific attributes. Leave as "" for removing entire datasets or groups.

Notes

Please see UPS_IO_OPEN_METHOD_enum (section B page 82) for a description of the different open methods.

Please see UPS_IO_ACCESS_PES_enum (section B page 70) for environment variables that affect this call.

SeeAlso

`UPS_IO_File_close` (page 197)
`UPS_IO_File_open` (page 198)
`UPS_IO_Group_close` (page 209)
`UPS_IO_Group_open` (page 210)
`UPS_IO_Info_create` (page 214)
`UPS_IO_Rm` (page 227)

UPS_IO_File_type()

Package

io

Purpose

Returns the type of protocol that wrote the file. If the protocol cannot be determined, protocol is set to UPS_IO_PROTOCOL_UNKNOWN

Usage

C	`ierr = UPS_IO_File_type`	`(name, protocol);`
Fortran	`call UPSF_IO_FILE_TYPE`	`(name, protocol, ierr)`
Fortran77	`call UPS_IO_FILE_TYPE`	`(name, protocol, ierr)`

Arguments

`name`
Intent: in
C type: const char*
Fortran type: UPS_KIND_CHAR {0}
Fortran77 type: UPS_KIND_CHAR {0}
The name of the file.

This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`protocol`
Intent: out
C type: UPS_IO_PROTOCOL_enum*
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The type of prototocol that wrote the file.

	If unknown, protocol is set to UPS_IO_PROTOCOL_UNKNOWN. Please see UPS_IO_PROTOCOL_enum (section B page 82)
`ierr`	*Intent*: out *C type*: (na) int return value *Fortran type*: UPS_KIND_INT4 {0} *Fortran77 type*: UPS_KIND_INT4 {0} Return status. Returns UPS_OK if successful.

ReturnValues

Returns 0 if no errors. If the protocol was unknown but there were no errors, 0 will still be the value of ierr and protocol will be UPS_IO_PROTOCOL_UNKNOWN.

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

If able, UPS will use the appropriate protocol to read the file. If unable (eg the protocol is UPS_IO_PROTOCOL_UNKNOWN.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.
- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.
- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_AA_Opt_set` (page 98)

UPS_IO_Filter_get()

Package

io

Purpose

Get filter values. Currently, the filter is only used for UPS_IO_Info_count and UPS_IO_Info_create calls. The filter modifies the number of matches in those calls.

Usage

C	`ierr = UPS_IO_Filter_get`	`(filter_type, buf);`
Fortran	`call UPSF_IO_FILTER_GET`	`(filter_type, buf, ierr)`
Fortran77	`call UPS_IO_FILTER_GET`	`(filter_type, buf, ierr)`

Arguments

`filter_type`	*Intent*:	in
	C type:	UPS_IO_FILTER_TYPE_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The filter type. Please see UPS_IO_FILTER_TYPE_enum (section B page 73) for a listing of the possible values.	
`buf`	*Intent*:	out
	C type:	void*
	Fortran type:	user_choice {0-1:filter_type:filter_type}
	Fortran77 type:	user_choice {0-1:filter_type:filter_type}
	The value associated with filter_type. If a string, the buffer will not have any additional null-terminator written to the buffer. The length may be obtained from UPS_IO_Filter_get (section C.8 page 206).	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

  This also contains examples of using UPS_IO_Filter_set to create a filters that modify the results of UPS_IO_Info_count and UPS_IO_Info_create calls.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_Filter_get` (page 206)
`UPS_IO_Filter_set` (page 208)
`UPS_IO_Info_count` (page 212)
`UPS_IO_Info_create` (page 214)
`UPS_IO_Info_create_self` (page 217)
`UPS_IO_Info_free` (page 219)
`UPS_IO_Info_item_get` (page 220)
`UPS_IO_Info_item_set` (page 222)

UPS_IO_Filter_set()

Package

io

Purpose

Set filter values. Currently, the filter is only used for UPS_IO_Info_count and UPS_IO_Info_create calls. The filter limits the number of matches in those calls.

Usage

C	`ierr = UPS_IO_Filter_set`	`(filter_type, buf);`
Fortran	`call UPSF_IO_FILTER_SET`	`(filter_type, buf, ierr)`
Fortran77	`call UPS_IO_FILTER_SET`	`(filter_type, buf, ierr)`

Arguments

`filter_type`

Intent:	in
C type:	UPS_IO_FILTER_TYPE_enum
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The filter type.
Please see UPS_IO_FILTER_TYPE_enum
(section B page 73)
for a listing of the possible values.

`buf`

Intent:	in
C type:	const void*
Fortran type:	user_choice {0-1:filter_type:filter_type}
Fortran77 type:	user_choice {0-1:filter_type:filter_type}

The value associated with filter_type.

If setting a string, it must be null-terminated.
As an example, Fortran users may pass in a string
concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)

`ierr`

Intent:	out
C type:	(na) int return value
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_Filter_get` (page 206)
`UPS_IO_Filter_set` (page 208)
`UPS_IO_Info_count` (page 212)
`UPS_IO_Info_create` (page 214)
`UPS_IO_Info_create_self` (page 217)
`UPS_IO_Info_free` (page 219)
`UPS_IO_Info_item_get` (page 220)
`UPS_IO_Info_item_set` (page 222)

UPS_IO_Group_close()

Package

io

Purpose

Close a group_id opened with UPS_IO_Group_open

Usage

```
C           ierr = UPS_IO_Group_close   (group_id);
Fortran     call UPSF_IO_GROUP_CLOSE    (group_id, ierr)
Fortran77   call UPS_IO_GROUP_CLOSE     (group_id, ierr)
```

Arguments

`group_id`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}

	Fortran77 type:	UPS_KIND_INT4 {0}
	Group id obtained by UPS_IO_Group_open.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.
File ids obtained by UPS_IO_File_open must be closed by UPS_IO_File_close. Group ids obtained by UPS_IO_Group_open must be closed by UPS_IO_Group_close.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_File_close` (page 197)
`UPS_IO_File_open` (page 198)
`UPS_IO_Group_close` (page 209)
`UPS_IO_Group_open` (page 210)
`UPS_IO_Info_create` (page 214)

UPS_IO_Group_open()

Package

io

Purpose

Opens a group and returns a group_id
If the open_method is UPS_IO_OPEN_CREATE, parent directories will be created if needed.

This is a synchronization point for all processes accessing the file when creating a group.

Usage

C	`ierr = UPS_IO_Group_open`	`(name, group_id_parent, open_method, group_id);`
Fortran	`call UPSF_IO_GROUP_OPEN`	`(name, group_id_parent, open_method, group_id, ierr)`
Fortran77	`call UPS_IO_GROUP_OPEN`	`(name, group_id_parent, open_method, group_id, ierr)`

Arguments

`name`

Intent: in
C type: const char*
Fortran type: UPS_KIND_CHAR {0}
Fortran77 type: UPS_KIND_CHAR {0}
The name of the group (wrt group_id_parent)

This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`group_id_parent`

Intent: in
C type: int
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Group id where to create group.
This id can be an id obtained from IO package open functions. Thus, a file can have a structure of groups like a unix directory structure.

`open_method`

Intent: in
C type: UPS_IO_OPEN_METHOD_enum
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Types of opening (ie open for read, create, ...)
Please see UPS_IO_OPEN_METHOD_enum (section B page 82)

`group_id`

Intent: inout
C type: int*
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Group id. This id may be passed to UPS_IO_Group_open calls.
This id must be closed with UPS_IO_Group_close.
By default, UPS sets this value and will make it the same value as returned by the underlying protocol (eg an HDF location id).
You may use your own non-negative value by calling

	UPS_AA_Opt_set with UPS_IO_OPT_LOC_ID_USERS set to UPS_DT_TRUE.
`ierr`	*Intent*: out *C type*: (na) int return value *Fortran type*: UPS_KIND_INT4 {0} *Fortran77 type*: UPS_KIND_INT4 {0} Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.
File ids obtained by UPS_IO_File_open must be closed by UPS_IO_File_close. Group ids obtained by UPS_IO_Group_open must be closed by UPS_IO_Group_close.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_File_close` (page 197)
`UPS_IO_File_open` (page 198)
`UPS_IO_Group_close` (page 209)
`UPS_IO_Group_open` (page 210)
`UPS_IO_Info_create` (page 214)

UPS_IO_Info_count()

Package

io

Purpose

Return the number of info_ids associated with the object given the info_type. This might be used for allocating the info_ids integer array to be passed into UPS_IO_Info_create.

A filter may be applied to modify the results by calling UPS_IO_Filter_set.

Usage

C	`ierr = UPS_IO_Info_count`	`(object_name, object_id, info_type, count);`
Fortran	`call UPSF_IO_INFO_COUNT`	`(object_name, object_id, info_type, count, ierr)`
Fortran77	`call UPS_IO_INFO_COUNT`	`(object_name, object_id, info_type, count, ierr)`

Arguments

`object_name`

Intent:	in
C type:	const char*
Fortran type:	UPS_KIND_CHAR {0}
Fortran77 type:	UPS_KIND_CHAR {0}

The name of the object (wrt object_id).
If object_name is ".", object_id is the id of the object.

The path to the object is obtained from object_name relative to object_id. If object_name is ".", object_id actually points to the object.

This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)

`object_id`

Intent:	in
C type:	int
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

An object id that qualifies location of object_name.
This id can be an id obtained from IO package open functions.

`info_type`

Intent:	in
C type:	UPS_IO_INFO_TYPE_enum
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The info type in question.
Please see UPS_IO_INFO_TYPE_enum
(section B page 79)
for a listing of the possible values.

`count`

Intent:	out
C type:	int*
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Return the number of info_ids associated with the object given the info_type.

`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_Filter_get` (page 206)
`UPS_IO_Filter_set` (page 208)
`UPS_IO_Info_count` (page 212)
`UPS_IO_Info_create` (page 214)
`UPS_IO_Info_create_self` (page 217)
`UPS_IO_Info_free` (page 219)
`UPS_IO_Info_item_get` (page 220)
`UPS_IO_Info_item_set` (page 222)

UPS_IO_Info_create()

Package

io

Purpose

Create and return info_ids associated with the object and info_type. The number of ids returned is found with a call to UPS_IO_Info_count.

A filter may be applied to modify the results by calling UPS_IO_Filter_set.

Usage

C	`ierr = UPS_IO_Info_create`	`(object_name, object_id, info_type, info_ids);`
Fortran	`call UPSF_IO_INFO_CREATE`	`(object_name, object_id, info_type, info_ids, ierr)`
Fortran77	`call UPS_IO_INFO_CREATE`	`(object_name, object_id, info_type, info_ids, ierr)`

Arguments

`object_name`
Intent: in
C type: const char*
Fortran type: UPS_KIND_CHAR {0}
Fortran77 type: UPS_KIND_CHAR {0}
The name of the object (wrt object_id).
If object_name is ".", object_id is the id of the object.

The path to the object is obtained from object_name relative to object_id. If object_name is ".", object_id actually points to the object.

For info_type of UPS_IO_INFO_DATA_DIST:
An object_name of 0 length ("") will create an empty info.
An object_name of non-0 length will fill the info with data on how penum filled the dataset.

This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`object_id`
Intent: in
C type: int
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
An object id that qualifies location of object_name.
This id can be an id obtained from IO package open functions.

`info_type`
Intent: in
C type: UPS_IO_INFO_TYPE_enum
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The info type in question.
Please see UPS_IO_INFO_TYPE_enum
(section B page 79)
for a listing of the possible values.

`info_ids`	*Intent*:	out
	C type:	int*
	Fortran type:	UPS_KIND_INT4 {0-1:UPS_IO_Info_count}
	Fortran77 type:	UPS_KIND_INT4 {0-1:UPS_IO_Info_count}
	This array contains info ids that are used in other IO info functions. Essentially, these ids are handles to internal structs. The number of elements in the array can be obtained by UPS_IO_Info_count. The user is responsible for calling UPS_IO_Info_free when finished with these ids.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_Filter_get` (page 206)
`UPS_IO_Filter_set` (page 208)
`UPS_IO_Info_count` (page 212)
`UPS_IO_Info_create` (page 214)
`UPS_IO_Info_create_self` (page 217)
`UPS_IO_Info_free` (page 219)
`UPS_IO_Info_item_get` (page 220)
`UPS_IO_Info_item_set` (page 222)

UPS_IO_Info_create_self()

Package

io

Purpose

Create an info_id for the object at object_name and object_loc or get an info_id for the attribute at attribute_name, object_name, and object_loc. If the object or attribute does not exist, info_id is not set and the count returned will be 0.

Usage

C	`ierr = UPS_IO_Info_create_self`	`(attribute_name, object_name, object_id, count, info_id);`
Fortran	`call UPSF_IO_INFO_CREATE_SELF`	`(attribute_name, object_name, object_id, count, info_id, ierr)`
Fortran77	`call UPS_IO_INFO_CREATE_SELF`	`(attribute_name, object_name, object_id, count, info_id, ierr)`

Arguments

`attribute_name`

Intent: in
C type: const char*
Fortran type: UPS_KIND_CHAR {0}
Fortran77 type: UPS_KIND_CHAR {0}

If you wish to get an info_id of a specific attribute, this is the name of the attribute. Otherwise, set this value to the null terminator [C="", Fortran=ACHAR(0)] and the info_id will be that of the object at object_name and object_id.

This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`object_name`

Intent: in
C type: const char*
Fortran type: UPS_KIND_CHAR {0}
Fortran77 type: UPS_KIND_CHAR {0}

The name of the object (wrt object_id).
If object_name is ".", object_id is the id of the object.

The path to the object is obtained from object_name relative to object_id. If object_name is ".", object_id actually points to the object.

This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)

name(1:name_length)//ACHAR(0)

`object_id`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	An object id that qualifies location of object_name. This id can be an id obtained from IO package open functions.	
`count`	*Intent*:	out
	C type:	int*
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	If the object or attribute is found, the count will be 1 and info_id will be set (and info_id must be freed by with a call to UPS_IO_Info_free). Otherwise, count will be 0 and info_id will not be set.	
`info_id`	*Intent*:	inout
	C type:	int*
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	If the object or attribute is found, the count will be 1 and info_id will be set (and info_id must be freed by with a call to UPS_IO_Info_free). Otherwise, count will be 0 and info_id will not be set. The info_id can be used in other IO functions. Essentially, these ids are handles to internal structs.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

The functionality of this call is a combination of UPS_IO_Filter_set, UPS_IO_Info_count, and UPS_IO_Info_create. I found myself writing code using the above functions to find if an object existed and get info about it. It was getting tedious so I decided to put it in a function.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_Filter_get` (page 206)
`UPS_IO_Filter_set` (page 208)
`UPS_IO_Info_count` (page 212)
`UPS_IO_Info_create` (page 214)
`UPS_IO_Info_create_self` (page 217)
`UPS_IO_Info_free` (page 219)
`UPS_IO_Info_item_get` (page 220)
`UPS_IO_Info_item_set` (page 222)

UPS_IO_Info_free()

Package

io

Purpose

Free info ids

Usage

C	`ierr = UPS_IO_Info_free`	`(count, info_ids);`
Fortran	`call UPSF_IO_INFO_FREE`	`(count, info_ids, ierr)`
Fortran77	`call UPS_IO_INFO_FREE`	`(count, info_ids, ierr)`

Arguments

`count`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The number of elements in info_ids array	
`info_ids`	*Intent*:	in
	C type:	const int*
	Fortran type:	UPS_KIND_INT4 {0-1}
	Fortran77 type:	UPS_KIND_INT4 {0-1}
	An array containing info ids that were obtained	

	by UPS_IO_Info_create.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.
- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.
- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_Info_create` (page 214)
`UPS_IO_Info_create_self` (page 217)
`UPS_IO_Info_free` (page 219)

UPS_IO_Info_item_get()

Package

io

Purpose

Get the value of a private data variable defined by info_id, info_item, and info_type.

Usage

C	`ierr = UPS_IO_Info_item_get`	`(info_id, info_item, buf);`
Fortran	`call UPSF_IO_INFO_ITEM_GET`	`(info_id, info_item, buf, ierr)`
Fortran77	`call UPS_IO_INFO_ITEM_GET`	`(info_id, info_item, buf, ierr)`

Arguments

`info_id`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The id of the info to deal with. ID's are obtained from UPS_IO_Info_create.	
`info_item`	*Intent*:	in
	C type:	UPS_IO_INFO_ITEM_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The info item in question. Please see UPS_IO_INFO_ITEM_enum (section B page 74) for a listing of the possible values.	
`buf`	*Intent*:	out
	C type:	void*
	Fortran type:	user_choice {0-1:info_item:info_item}
	Fortran77 type:	user_choice {0-1:info_item:info_item}
	The value of the info_item. If a string, the buffer will not have any additional null-terminator written to the buffer. The length may be obtained from UPS_IO_Info_item_get (section C.8 page 220).	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_Filter_get` (page 206)
`UPS_IO_Filter_set` (page 208)
`UPS_IO_Info_count` (page 212)
`UPS_IO_Info_create` (page 214)
`UPS_IO_Info_create_self` (page 217)
`UPS_IO_Info_free` (page 219)
`UPS_IO_Info_item_get` (page 220)
`UPS_IO_Info_item_set` (page 222)

UPS_IO_Info_item_set()

Package

io

Purpose

Set the value of a private data variable defined by info_id, info_item, and info_type.

Usage

C	`ierr = UPS_IO_Info_item_set`	`(info_id, info_item, buf);`
Fortran	`call UPSF_IO_INFO_ITEM_SET`	`(info_id, info_item, buf, ierr)`
Fortran77	`call UPS_IO_INFO_ITEM_SET`	`(info_id, info_item, buf, ierr)`

Arguments

`info_id`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The id of the info to deal with.	
	ID's are obtained from UPS_IO_Info_create.	
`info_item`	*Intent*:	in
	C type:	UPS_IO_INFO_ITEM_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The info item in question.	
	Please see UPS_IO_INFO_ITEM_enum	
	(section B page 74)	
	for a listing of the possible values.	
`buf`	*Intent*:	in
	C type:	const void*
	Fortran type:	user_choice {0-1:info_item:info_item}
	Fortran77 type:	user_choice {0-1:info_item:info_item}
	The value of the info_item.	
	If a string, it must be null-terminated.	
	As an example, Fortran users may pass in a string	

concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)

`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

Notes

- Setting some options invalidates others (requiring you to reset them if you had set the previously).

  - UPS_IO_INFO_DIMS: UPS_IO_INFO_DIMS_TOTAL
  - UPS_IO_INFO_NDIMS: Everything
  - UPS_IO_INFO_STARTS: UPS_IO_INFO_DIMS_TOTAL, UPS_IO_INFO_PGRID_DIMS, UPS_IO_INFO_PGRID_ORDER
  - UPS_IO_INFO_PGRID_DIMS: UPS_IO_INFO_STARTS
  - UPS_IO_INFO_PGRID_ORDER: UPS_IO_INFO_STARTS

  From writing UPS regression tests, I have found it easy to create bugs when trying to reuse info_ids for several dataset writes and changing the parameters via UPS_IO_Info_item_set().

  As long as you do not change the parameters, reuse is fine. However, if you do change the parameters, I would suggest just getting a "fresh" info_id and setting all the parameters again. There should not be much of a performance loss doing this.

SeeAlso

`UPS_IO_Filter_get` (page 206)
`UPS_IO_Filter_set` (page 208)
`UPS_IO_Info_count` (page 212)
`UPS_IO_Info_create` (page 214)
`UPS_IO_Info_create_self` (page 217)
`UPS_IO_Info_free` (page 219)
`UPS_IO_Info_item_get` (page 220)
`UPS_IO_Info_item_set` (page 222)

UPS_IO_Loc_item_get()

Package

io

Purpose

Get the value of a private data variable defined by loc_id and loc_item.

Usage

C	`ierr = UPS_IO_Loc_item_get`	`(loc_id, loc_item, buf);`
Fortran	`call UPSF_IO_LOC_ITEM_GET`	`(loc_id, loc_item, buf, ierr)`
Fortran77	`call UPS_IO_LOC_ITEM_GET`	`(loc_id, loc_item, buf, ierr)`

Arguments

`loc_id`

Intent:	in
C type:	int
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The id of the loc to deal with.
ID's are obtained from UPS_IO_File_open and
UPS_IO_Group_open.

`loc_item`

Intent:	in
C type:	UPS_IO_LOC_ITEM_enum
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

The loc item in question.
Please see UPS_IO_LOC_ITEM_enum
(section B page 79)
for a listing of the possible values.

`buf`

Intent:	out
C type:	void*
Fortran type:	user_choice {0-1:loc_item:loc_item}
Fortran77 type:	user_choice {0-1:loc_item:loc_item}

The value of the loc_item.
If a string, the buffer will not have any additional
null-terminator written to the buffer.
The length may be obtained from UPS_IO_Loc_item_get
(section C.8 page 224).

`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_File_open` (page 198)
`UPS_IO_Group_open` (page 210)
`UPS_IO_Info_item_get` (page 220)
`UPS_IO_Info_item_set` (page 222)
`UPS_IO_Loc_item_get` (page 224)
`UPS_IO_Loc_item_set` (page 225)

UPS_IO_Loc_item_set()

Package

io

Purpose

Get the value of a private data variable defined by loc_id and loc_item.

Usage

C	`ierr = UPS_IO_Loc_item_set`	`(loc_id, loc_item, buf);`
Fortran	`call UPSF_IO_LOC_ITEM_SET`	`(loc_id, loc_item, buf, ierr)`
Fortran77	`call UPS_IO_LOC_ITEM_SET`	`(loc_id, loc_item, buf, ierr)`

Arguments

`loc_id`	*Intent*:	in
	C type:	int
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The id of the loc to deal with. ID's are obtained from UPS_IO_File_open and UPS_IO_Group_open.	
`loc_item`	*Intent*:	in
	C type:	UPS_IO_LOC_ITEM_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The loc item in question. Please see UPS_IO_LOC_ITEM_enum (section B page 79) for a listing of the possible values.	
`buf`	*Intent*:	in
	C type:	const void*
	Fortran type:	user_choice {0-1:loc_item:loc_item}
	Fortran77 type:	user_choice {0-1:loc_item:loc_item}
	The value of the loc_item. If a string, it must be null-terminated. As an example, Fortran users may pass in a string concatenated with the null-character: 'my string here without null terminator'//ACHAR(0)	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful

Discussion

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_Info_item_get` (page 220)
`UPS_IO_Info_item_set` (page 222)
`UPS_IO_Loc_item_get` (page 224)
`UPS_IO_Loc_item_set` (page 225)
`UPS_IO_Loc_item_get` (page 224)
`UPS_IO_Loc_item_set` (page 225)

UPS_IO_Rm()

Package

io

Purpose

Remove an object (group, dataset, attribute) from a file. The remove is recursive.
This is a synchronization point for all processes accessing the file.

Usage

C	`ierr = UPS_IO_Rm`	`(attribute_name, object_name, object_id);`
Fortran	`call UPSF_IO_RM`	`(attribute_name, object_name, object_id, ierr)`
Fortran77	`call UPS_IO_RM`	`(attribute_name, object_name, object_id, ierr)`

Arguments

`attribute_name`

Intent: in
C type: const char*
Fortran type: UPS_KIND_CHAR {0}
Fortran77 type: UPS_KIND_CHAR {0}
The name of the attribute to remove. If you wish to remove the object defined by object_name and object_id, set attribute_name to be the null terminator [C="", Fortran=ACHAR(0)].
This string must be null-terminated.
As an example, Fortran users may pass in a string concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`object_name`

Intent: in
C type: const char*
Fortran type: UPS_KIND_CHAR {0}
Fortran77 type: UPS_KIND_CHAR {0}
The name of the object (wrt object_id).
If attribute_name is the null terminator, this is the object to remove.

The path to the object is obtained from object_name
relative to object_id. If object_name is ".",
object_id actually points to the object.

This string must be null-terminated.
As an example, Fortran users may pass in a string
concatenated with the null-character:
'my string here without null terminator'//ACHAR(0)
name(1:name_length)//ACHAR(0)

`object_id`

Intent: in
C type: int
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
An object id that qualifies location of object_name.
This id can be an id obtained from IO package open
functions (UPS_IO_File_open, UPS_IO_Group_open).

`ierr`

Intent: out
C type: (na) int return value
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Return status. Returns UPS_OK if successful.

ReturnValues

Returns 0 if successful

Discussion

When an object is removed, it creates a hole in the file. A write during the same run will attempt to use the hole (if it can fit) created by the rm. Hole reuse is not done inbetween runs. Thus rm/write/rm/write/... will cause the file to grow. To "repack" the file, use the command line executable ups_io_cp to copy a file to a new location.

Existing info ids (from UPS_IO_Info_create) and location ids (from UPS_IO_Group_open) that "point at" the removed objects should be freed (UPS_IO_Info_free or UPS_IO_Group_close respectively) before a call to UPS_IO_Rm. The object is not guaranteed to be removed until all references to it have been closed.

See UPS_IO_File_open (section C.8 page 198) for a high level discussion of the structure of an IO package file.

Examples

See the following for discussions on using the io package:

- UPS_IO_File_open (section C.8 page 198)

  This contains discussions on file/group creation and using UPS_IO_Info_create to get information about members of a group.

- UPS_IO_Dataset_read (section C.8 page 183)

  This contains discussions on dataset writing/reading and using UPS_IO_Info_create to describe the shape of the dataset.

- UPS_IO_Attr_read (section C.8 page 175)

  This contains discussions on attribute writing/reading and using UPS_IO_Info_create to get information about an objects attributes.

SeeAlso

`UPS_IO_Group_open` (page 210)
`UPS_IO_Group_close` (page 209)
`UPS_IO_Info_create` (page 214)
`UPS_IO_Info_free` (page 219)

C.9 Utilities

See the packages section (section 6.8, page 35 for a general description of this package.

This section contains an alphabetical listing of the utility routines available in UPS.

UPS_UT_Binary_op()

Package

ut

Purpose

UPS_DP_Binary_op performs a specified operation on 2 vectors (Options specified below). The effect is to convert 2 arrays of count elements each into 1 array of count elements. This function is used in UPS internals but is also provided for the users benefit.

Usage

C
```
ierr = UPS_UT_Binary_op  (src_a, src_b, datatype,
                          operation, count, dest);
```

Arguments

`src_a`

Intent:	in
C type:	void*
Fortran type:	(na) no equivalent routine
Fortran77 type:	(na) no equivalent routine

The input vector of data.

`src_b`

Intent:	in
C type:	void*
Fortran type:	(na) no equivalent routine
Fortran77 type:	(na) no equivalent routine

The other input vector of data.

`datatype`

Intent:	in
C type:	UPS_DT_Datatype_enum
Fortran type:	(na) no equivalent routine
Fortran77 type:	(na) no equivalent routine

The type of the data.
Please see UPS_DT_Datatype_enum
(section B page 63)
for a listing of the possible datatypes.

`operation`

Intent:	in
C type:	UPS_AA_Operation_enum
Fortran type:	(na) no equivalent routine
Fortran77 type:	(na) no equivalent routine

The operation to perform.
Please see UPS_AA_Operation_enum
(section B page 56)
for a listing of the possible operations.

`count`

Intent:	in

	C type:	int
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The number of elements in input array src_[ab]	
`dest`	*Intent*:	out
	C type:	void*
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The output vector.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

This routine was added to provide a common routine to do binary operations. If a vector is provided, then the operation is done on the first element of each array, stored in dest, then the second element...and so on.

NOTE: For increased performance, this routine does not check for UPS initialization.

SeeAlso

`UPS_UT_Binary_op` (page 230)
`UPS_UT_Binary_opm` (page 231)
`UPS_UT_Reduce_op` (page 244)
`UPS_UT_Reduce_opm` (page 246)

UPS_UT_Binary_opm()

Package

ut

Purpose

UPS_UT_Binary_opm extends the capability of UPS_UT_Binary_op by adding a masking capability.

Usage

C
```
ierr = UPS_UT_Binary_opm  (src_a, src_b, mask, datatype,
                           operation, count, dest);
```

Arguments

`src_a`	*Intent*:	in
	C type:	void*
	Fortran type:	(na) no equivalent routine

	Fortran77 type:	(na) no equivalent routine
	The input vector of data.	
`src_b`	*Intent*:	in
	C type:	void*
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The input vector of data.	
`mask`	*Intent*:	in
	C type:	const int*
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	boolean (int) array specifying which elements of the input vector are to be operated upon.	
`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The type of the data. Please see UPS_DT_Datatype_enum (section B page 63) for a listing of the possible datatypes.	
`operation`	*Intent*:	in
	C type:	UPS_AA_Operation_enum
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The operation to perform. Please see UPS_AA_Operation_enum (section B page 56) for a listing of the possible operations.	
`count`	*Intent*:	in
	C type:	int
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The number of elements in input array src_[ab]	
`dest`	*Intent*:	out
	C type:	void*
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The output vector.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

This routine was added to provide a common routine to do binary operations. If a vector is provided, then the operation is done on the first element of each array, stored in dest, then the second element...and so on.

NOTE: For increased performance, this routine does not check for UPS initialization.

SeeAlso

`UPS_UT_Binary_op` (page 230)
`UPS_UT_Binary_opm` (page 231)
`UPS_UT_Reduce_op` (page 244)
`UPS_UT_Reduce_opm` (page 246)

UPS_UT_Checksum_get()

Package

ut

Purpose

Return a int8 (long long) checksum value computed from a buffer. This checksum can be used as a check of data corruption.

Usage

C	`ierr = UPS_UT_Checksum_get`	`(buf, count, datatype, checksum_type, checksum_value);`
Fortran	`call UPSF_UT_CHECKSUM_GET`	`(buf, count, datatype, checksum_type, checksum_value, ierr)`
Fortran77	`call UPS_UT_CHECKSUM_GET`	`(buf, count, datatype, checksum_type, checksum_value, ierr)`

Arguments

`buf`

Intent:	in
C type:	void*
Fortran type:	user_choice {0-3}
Fortran77 type:	user_choice {0-3}

The buffer of data to compute the checksum on.

`count`

Intent:	in
C type:	long long
Fortran type:	UPS_KIND_INT8 {0}
Fortran77 type:	UPS_KIND_INT8 {0}

The number of elements of type datatype in buf

`datatype`

Intent:	in
C type:	UPS_DT_Datatype_enum

Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The datatype of the elements.
Please see UPS_DT_Datatype_enum (section B page 63)

`checksum_type`

Intent: in
C type: UPS_UT_CHECKSUM_TYPE_enum
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The type of checksum for perform
Please see UPS_UT_CHECKSUM_TYPE_enum (section B page 85)
for a listing/explanation of different items.
Currently, only UPS_UT_CHECKSUM_CRC is supported.

`checksum_value`

Intent: out
C type: long long*
Fortran type: UPS_KIND_INT8 {0}
Fortran77 type: UPS_KIND_INT8 {0}
The checksum value computed on buf.
NOTE: this value is a signed quantity.
See the discussion below.

`ierr`

Intent: out
C type: (na) int return value
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

- checksum_value is signed

  In order to allow the same interface (and same checksum_value) to be returned from both the C and Fortran interfaces, I decided to have checksum_value be a signed quantity. Internally checksum_value is likely to be unsigned and then cast to a signed quantity.

  A problem arises when comparing two signed checksum values that, when converted from unsigned quantities, end up being -0 and +0. There is no way to distinguish between the two. So, the checksums will be different yet you will think they are the same.

  However, the probability of the above happening is very small. First, the checksum value of a data buffer would have to result in a (+/-)0. Then, the other checksum would have to result in the other (-/+)0. For a 64 bit random checksums, that would be:

  ```
          1/(2^63) * 1/(2^64) = 1/(2^127)  (via Sunlung Suen)
  ```

  As checksums are never truly random, the actual probability would be somewhat less than this. The CRC checksum method will produce fairly random sequences so this should not be a problem. Experiments pending....

Examples

Suppose you wanted to verify that data is being sent correctly:

1. UPS_UT_Checksum_get on send buffer to get checksum_send
2. Sender: Send buffer and checksum_send
3. Receiver: Recv buffer and checksum_send
4. UPS_UT_Checksum_get on recv buffer to get checksum_recv
5. checksum_recv and checksum_send must be the same

UPS_UT_Convert()

Package

ut

Purpose

UPS_UT_Convert converts between UPS and other package (eg MPI) parameters.
This function is used in UPS internals but is also provided for the users benefit.

Usage

C	`ierr = UPS_UT_Convert`	`(in, out, convert_type);`
Fortran	`call UPSF_UT_CONVERT`	`(ierr)`
Fortran77	`call UPS_UT_CONVERT`	`(in, out, convert_type, ierr)`

Arguments

`in`
- *Intent*: in
- *C type*: const void*
- *Fortran type*: (na) no equivalent routine
- *Fortran77 type*: user_choice {0}

in converted to out

`out`
- *Intent*: out
- *C type*: void*
- *Fortran type*: (na) no equivalent routine
- *Fortran77 type*: user_choice {0}

in converted to out

`convert_type`
- *Intent*: in
- *C type*: UPS_UT_Convert_enum
- *Fortran type*: (na) no equivalent routine
- *Fortran77 type*: UPS_KIND_INT4 {0}

The type of conversion.
See UPS_UT_Convert_enum (section B page 86)
for a listing of the possible conversion types.

`ierr`
- *Intent*: out
- *C type*: (na) int return value
- *Fortran type*: UPS_KIND_INT4 {0}

Fortran77 type: UPS_KIND_INT4 {0}
Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

UPS needs to have a consistent definition of types (variables types, reduction operations, etc.). This function lets us convert between UPS types to "other package types".

UPS_UT_Dt_change()

Package

ut

Purpose

Assign to out_buf (out_datatype) the values of in_buf (in_datatype). If no conversion is possible, an error is returned.

Usage

C	`ierr = UPS_UT_Dt_change`	`(in_buf, in_datatype, out_datatype, count, out_buf);`
Fortran77	`call UPS_UT_DT_CHANGE`	`(in_buf, in_datatype, out_datatype, count, out_buf, ierr)`

Arguments

`in_buf`
Intent: inout
C type: void*
Fortran type: (na) only F77
Fortran77 type: user_choice {0-4::in_datatype}
The input buffer (count number of elements)

`in_datatype`
Intent: in
C type: UPS_DT_Datatype_enum
Fortran type: (na) only F77
Fortran77 type: user_choice {0-4::in_datatype}
Datatype of in_buf.
Please see UPS_DT_Datatype_enum (section B page 63) for a listing/description of the possible values.

`out_datatype`
Intent: in
C type: UPS_DT_Datatype_enum
Fortran type: (na) only F77
Fortran77 type: user_choice {0-4::in_datatype}
Datatype of out_buf.
Please see UPS_DT_Datatype_enum (section B page 63) for a listing/description of the possible values.

`count`	*Intent*:	in
	C type:	long long
	Fortran type:	(na) only F77
	Fortran77 type:	user_choice {0-4::in_datatype}
	Number of elements in both in_buf and out_buf	
`out_buf`	*Intent*:	inout
	C type:	void*
	Fortran type:	(na) only F77
	Fortran77 type:	user_choice {0-4::in_datatype}
	The output buffer (count number of elements).	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	(na) only F77
	Fortran77 type:	user_choice {0-4::in_datatype}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns 0 if successful

Discussion

This routine simply does a copy of 1 datatype to another via a C assignment. So, any caveats for C copying apply here as well. For example, copying a floating value to an integer value will truncate and copying a long long to an int might produce overflow problems.

This operation will likely be irreversible: float to an int back to a float will likely give you a different value than what you started with.

I have conversion to/from character data (UPS_DT_CHAR) because it was because I use it in the HDF-EnSight reader libusert-HDF. This conversion will give different output depending upon the architecture. Be careful when using this conversion - do not expect the same answer.

This routine is mainly used in UPS internals but is also provided to the user.

UPS_UT_Get_name_or_value()

Package

ut

Purpose

Return the name or value given the value or name respectively. In the case of a tie, the first match is returned.

Usage

C `ierr = UPS_UT_Get_name_or_value (name_value_struct, name_or_value, get_name_or_value, value_or_name);`

Arguments

`name_value_struct`	*Intent*:	in
	C type:	const UPS_UT_Name_value_struct*

	Fortran type:	(na) not implemented
	Fortran77 type:	(na) not implemented
	the name_value struct	
`name_or_value`	*Intent*:	in
	C type:	const void*
	Fortran type:	(na) not implemented
	Fortran77 type:	(na) not implemented
	name or value to look for	
`get_name_or_value`	*Intent*:	in
	C type:	UPS_UT_Name_or_value_enum
	Fortran type:	(na) not implemented
	Fortran77 type:	(na) not implemented
	If looking for a name or a value. See UPS_UT_Name_or_value_enum (section B page 87) for the different options.	
`value_or_name`	*Intent*:	out
	C type:	void*
	Fortran type:	(na) not implemented
	Fortran77 type:	(na) not implemented
	value or name corresponding to name_or_value. 0 (value) or NULL (name) is returned if not found. See ierr below.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	(na) not implemented
	Fortran77 type:	(na) not implemented
	Return status. Returns UPS_OK if successfully found.	

ReturnValues

Returns UPS_OK if found.

Discussion

The last name/value pair must be: "", 0

When a name is returned, it is a pointer to the corresponding char* in the name_value_struct. So, if you free the name_value_struct, the name will no longer point to a valid char*.

Examples

C:

```
static UPS_UT_Name_value_struct etypes_nnodes[] =
  {
    { "point",          1  },
    { "bar2",           2  },
    { "bar3",           3  },
    { "quad8",          8  },
    { "hexa8",          8  },
    { "", 0 }
  };
int ierr, value, count;
char *name;
...
ierr = UPS_UT_Get_name_or_value( etypes_nnodes,
                                 "hexa8",
                                 UPS_UT_GET_VALUE,
                                 &value );
...
value = 8;
ierr = UPS_UT_Get_name_or_value( etypes_nnodes,
                                 &value,
                                 UPS_UT_GET_NAME,
                                 &name );
...
ierr = UPS_UT_Get_name_or_value( etypes_nnodes,
                                 NULL,
                                 UPS_UT_GET_COUNT,
                                 &count );
```

This routine can also be used to get names/values of UPS constants.
Code Location:

```
...
ierr = UPS_UT_Get_name_or_value( UPS_AA_Code_location_enum_val_name,
                                 "UPS_IO_LOC_ATTR_READ",
                                 UPS_UT_GET_VALUE,
                                 &value );
...
value = UPS_IO_LOC_ATTR_READ;
ierr = UPS_UT_Get_name_or_value( UPS_AA_Code_location_enum_val_name,
                                 &value,
                                 UPS_UT_GET_NAME,
                                 &name );
```

Datatype:

```
...
ierr = UPS_UT_Get_name_or_value( UPS_DT_Datatype_enum_val_name,
                                 "UPS_DT_DOUBLE",
                                 UPS_UT_GET_VALUE,
                                 &value );
...
value = UPS_DT_DOUBLE;
ierr = UPS_UT_Get_name_or_value( UPS_DT_Datatype_enum_val_name,
                                 &value,
                                 UPS_UT_GET_NAME,
                                            &name );
```

The rule for the name of the UPS_UT_Name_value_struct that has the name/value enums you want is to take the name of the enum and add on "_val_name" to the end.

```
UPS_AA_Code_location_enum --> UPS_AA_Code_location_enum_val_name
```

UPS_UT_Loc_struct_alloc()

Package

ut

Purpose

Allocates space in the private variable `upsp_ut.<d,f,i,l>buf_<in,out>`. The process is as follows:

1. If there is already enough space for the count desired, just return.
2. Otherwise, free the location and set the new location to a place with enough memory.
3. reset the private size of this new buffer.

Negative counts have the effect of freeing the memory for that private variable.

Usage

C `ierr = UPS_UT_Loc_struct_alloc  (datatype, count);`

Arguments

`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The type of the data. Please see UPS_DT_Datatype_enum (section B page 63) for a listing of the possible datatypes.	
`count`	*Intent*:	in
	C type:	const int
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	Number of elements being allocated.	

ReturnValues

Returns UPS_OK if successful. Values less than 0 indicate an error.

Discussion

This function is used in UPS internals but is also provided for the users benefit.

UPS_UT_Loc_structure()

Package

ut

Purpose

Doing an allreduce for a location datatype requires a structure (containing .val and .loc elements) as input and output. Often (as is the case for fortran), you have either 2 arrays (one being val, and the other loc) or a single array ([2*i] = value, [2*i+1] = location: i=0,n).

This routine maps that structure to the arrays.

Usage

C
```
ierr = UPS_UT_Loc_structure  (val, loc, loc_struct,
                              datatype, count, operation,
                              loc_type);
```

Arguments

`val`	*Intent*:	inout
	C type:	void*
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	Either the array corresponding to the .val element in the struct or an array containing both the .val and .loc elements	
`loc`	*Intent*:	inout
	C type:	int*
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	Array of the loc values corresponding `.loc` in the struct	
`loc_struct`	*Intent*:	inout
	C type:	void*
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The array of `.val` `.loc` structs	
`datatype`	*Intent*:	in
	C type:	const UPS_DT_Datatype_enum
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The type of the local data. Please see UPS_DT_Datatype_enum (section B page 63) for a listing of the possible datatypes.	
`count`	*Intent*:	in
	C type:	const int
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	Number of vals you are dealing with	

`operation`	*Intent*:	in
	C type:	const UPS_UT_Loc_structure_op_enum
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The operation (winding or unwinding) Please see UPS_UT_Loc_structure_op_enum (section B page 87) for a listing of the possible operations.	
`loc_type`	*Intent*:	in
	C type:	const UPS_UT_Loc_type_enum
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	What type of loc structure you have (scalar, vector, value) Please see UPS_UT_Loc_type_enum (section B page 87) for a listing of the possible loc types.	

ReturnValues

Returns UPS_OK if successful. Values less than 0 indicate an error.

Discussion

This function is used in UPS internals but the c version is also provided for the users benefit.

UPS_UT_Mem_get_item()

Package

ut

Purpose

Get information about the memory routines. See the argument item_type below for a listing of the different information that can be obtained.

Usage

C	`ierr = UPS_UT_Mem_get_item`	`(item_type, address, item);`
Fortran	`call UPSF_UT_MEM_GET_ITEM`	`(item_type, address, item, ierr)`
Fortran77	`call UPS_UT_MEM_GET_ITEM`	`(item_type, address, item, ierr)`

Arguments

`item_type`	*Intent*:	in
	C type:	UPS_AA_Mem_item_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The type of info requested. Please see UPS_AA_Mem_item_enum (section B page 49) for a listing/explanation of different items.	

`address`	*Intent*:	inout
	C type:	void*
	Fortran type:	UPS_KIND_ADDRESS {0}
	Fortran77 type:	UPS_KIND_ADDRESS {0}
	The address of the memory area gotten by the private macros UPSP_UT_MEM_MALLOC. For general users, this value will probably be NULL since they do not have access to the private macros. For some item_type values, this argument is not used.	
`item`	*Intent*:	out
	C type:	void*
	Fortran type:	user_choice {0::item_type}
	Fortran77 type:	user_choice {0::item_type}
	The output value of the item_type. See item_type above.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

The main purpose for writing memory management is the detection of possible memory problems.

- **Memory Errors Detected**
  - **Near Overwrites**

    When UPS allocates a memory block, additional memory is allocated at the beginning and end of the block. If these "guard bytes" are corrupted, a memory overwrite has occurred. So, simple errors such as writing 1 or 2 elements beyond allocated space can be detected. However, a skip that writes beyond 2 elements can not be detected since the writes will take place beyond the guard bytes. The guard bytes are automatically checked when freed in UPS.
  - **Attempting to Free Memory Never Allocated**

    Information about memory allocated by UPS is stored in a linked list. Unlike the C function free(), UPS will return an error if it tries to free internal memory it has not allocated.
  - **Unfreed Memory Upon Termination**

    UPS_AA_Terminate checks to make sure all memory allocated by UPS has been freed.
- **Additional Information Provided upon Termination**

  Normal memory allocation in UPS is done via the internal macros UPSP_UT_MEM_(C|M|RE)ALLOC/UPSP_UT_MEM_FREE located in the internal header file upsp_ut.h. The code locations (section B page 58) UPS_UT_LOCP_MEM_ALLOC and

UPS_UT_LOCP_MEM_FREE make some data available to the user. When statistics are turned on (see UPS_AA_Statistics), the following info fields are recorded at the end of each alloc/free call:

– Field ID 0: Total size (bytes) of normal memory allocated
– Field ID 1: Total number of normal memory allocations active
– Field ID 2: Size (bytes) of normal memory just allocated/freed

Upon calling UPS_AA_Terminate, the file ups_log.ps will contain the above information. For example, one can see the high water mark for UPS normal memory allocation by looking at the max column of field id 0 of the UPS_UT_LOCP_MEM_ALLOC code location.

- **Performance Penalty**

  UPS code allocates memory by calling the private macro UPSP_UT_MEM_MALLOC and frees memory by calling the
  private macro UPSP_UT_MEM_FREE.

  During an allocation, the following steps are done:

  1. Create memory info struct and place in linked list.
  2. Allocate requested memory plus space for guard bytes before and after memory buffer.
  3. Initialize values of guard bytes.
  4. Call statistics gathering routine (eg records time spent in function).

  During a free, the following steps are done:

  1. Check values of guard bytes.
  2. Free allocated memory.
  3. Free memory info struct.
  4. Call statistics gathering routine (eg records time spent in function).

  This overhead takes additional time. On the SGI's, the time for a UPS malloc/free is approximately 100 times longer than a normal malloc/free. In stead of taking tens of nanoseconds, it takes microseconds.

  To reduce this affect, UPS tries to reuse memory that must be allocated. That is, when the memory requirements are small, we allocate buffers once in init routines (eg UPS_AA_Init or UPS_GS_Setup), reuse them as much as possible, and then free them upon UPS termination.

Notes

Please see the memory management variable section of
UPS_AA_ENVIRONMENT_VARIABLES_enum (section B page 43) for environment variables that affect this call.

SeeAlso

`UPS_AA_Statistics` (page 99)
`UPS_CM_Sm_get_item` (page 120)

UPS_UT_Reduce_op()

Package

ut

Purpose

UPS_DP_Reduce_op performs a specified operation on the input vector. The effect is to reduce count elements into 1 element. NOTE: if count is 0, the identity element for that operation is returned (if one exists - otherwise 0 is assigned to dest). For loc operations with 0 count, the loc will be 0.

Usage

C `ierr = UPS_UT_Reduce_op (src, datatype, operation, count, dest);`

Arguments

`src`	*Intent*:	in
	C type:	const void*
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The input vector of data.	
`datatype`	*Intent*:	in
	C type:	const UPS_DT_Datatype_enum
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The type of the data. Please see UPS_DT_Datatype_enum (section B page 63) for a listing of the possible datatypes.	
`operation`	*Intent*:	in
	C type:	const UPS_AA_Operation_enum
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The operation to perform. Please see UPS_AA_Operation_enum (section B page 56) for a listing of the possible operations.	
`count`	*Intent*:	in
	C type:	const int
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The number of elements in input array src_[ab]	
`dest`	*Intent*:	out
	C type:	void*
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The output vector.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

NOTE: For increased performance, this routine does not check for UPS initialization.

SeeAlso

`UPS_UT_Binary_op` (page 230)
`UPS_UT_Binary_opm` (page 231)
`UPS_UT_Reduce_op` (page 244)
`UPS_UT_Reduce_opm` (page 246)

UPS_UT_Reduce_opm()

Package

ut

Purpose

UPS_UT_Reduce_opm extends the capability of UPS_UT_Reduce_op by adding a masking capability.

Usage

C
```
ierr = UPS_UT_Reduce_opm  (src, mask, datatype,
                           operation, count, dest);
```

Arguments

`src`	*Intent*:	in
	C type:	const void*
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The input vector of data.	
`mask`	*Intent*:	in
	C type:	const int*
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	boolean (int) array specifying which elements of the input vector are to be operated upon.	
`datatype`	*Intent*:	in
	C type:	const UPS_DT_Datatype_enum
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The type of the data. Please see UPS_DT_Datatype_enum (section B page 63) for a listing of the possible datatypes.	
`operation`	*Intent*:	in
	C type:	const UPS_AA_Operation_enum

	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The operation to perform. Please see UPS_AA_Operation_enum (section B page 56) for a listing of the possible operations.	
`count`	*Intent*:	in
	C type:	const int
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The number of elements in input array src_[ab]	
`dest`	*Intent*:	out
	C type:	void*
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	The output vector.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	(na) no equivalent routine
	Fortran77 type:	(na) no equivalent routine
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

NOTE: For increased performance, this routine does not check for UPS initialization.

SeeAlso

`UPS_UT_Binary_op` (page 230)
`UPS_UT_Binary_opm` (page 231)
`UPS_UT_Reduce_op` (page 244)
`UPS_UT_Reduce_opm` (page 246)

UPS_UT_Sleep()

Package

ut

Purpose

Sleep for a time in seconds.

Usage

C	`ierr = UPS_UT_Sleep`	`(sleep_time);`
Fortran	`call UPSF_UT_SLEEP`	`(sleep_time, ierr)`
Fortran77	`call UPS_UT_SLEEP`	`(sleep_time, ierr)`

Arguments

`sleep_time`	*Intent*:	in
	C type:	double
	Fortran type:	UPS_KIND_REAL8 {0}
	Fortran77 type:	UPS_KIND_REAL8 {0}
	Time in seconds to sleep.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

The 'sleep' system call is used for the sleep time portion over 1 second and over. A finer grained sleep call is then made for the rest of the time.

Due to the overhead of the function call and the granularity of the underlying sleep function(s) called, UPS only guarantees that sleep_time is the minimum time spent sleeping.

For example, there is no high resolution sleep call on the TFLOP machine. Sleeping for 1.00001 seconds will result in a sleep of 2 seconds. On the other hand, SGI has a nano-second sleep call. Although UPS uses this nanosleep call, the granularity is not that high. In general a granulatity of 1-10 milliseconds is seen on architectures that support nanosleep.

UPS_UT_Sort_compress()

Package

ut

Purpose

Sorts and compresses (removes duplicates) the input buffer.

Usage

C	`ierr = UPS_UT_Sort_compress`	`(buf, count, datatype);`
Fortran	`call UPSF_UT_SORT_COMPRESS`	`(buf, count, datatype, ierr)`
Fortran77	`call UPS_UT_SORT_COMPRESS`	`(buf, count, datatype, ierr)`

Arguments

`buf`	*Intent*:	inout
	C type:	void*
	Fortran type:	user_choice {0-3}
	Fortran77 type:	user_choice {0-3}
	The buffer of data to sort and compress	
`count`	*Intent*:	inout
	C type:	long long *
	Fortran type:	UPS_KIND_INT8 {0}
	Fortran77 type:	UPS_KIND_INT8 {0}
	On input, the number of elements. On output, the new number of elements.	
`datatype`	*Intent*:	in
	C type:	UPS_DT_Datatype_enum
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	The datatype of the elements. Please see UPS_DT_Datatype_enum (section B page 63)	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

UPS_UT_Square_root()

Package

ut

Purpose

UPS_UT_Square_root returns the square root of the input variable, operating on it in terms of the input UPS datatype.

Usage

C	`ierr = UPS_UT_Square_root`	`(x, datatype, sqrt_x);`
Fortran	`call UPSF_UT_SQUARE_ROOT`	`(x, datatype, sqrt_x, ierr)`
Fortran77	`call UPS_UT_SQUARE_ROOT`	`(x, datatype, sqrt_x, ierr)`

Arguments

`x`
Intent: in
C type: const void*
Fortran type: user_choice {0}
Fortran77 type: user_choice {0}
The number whose square root will be returned.

`datatype`
Intent: in
C type: UPS_DT_Datatype_enum
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
The type of the data.
Please see UPS_DT_Datatype_enum
(section B page 63)
for a listing of the possible datatypes.

`sqrt_x`
Intent: out
C type: void*
Fortran type: user_choice {0}
Fortran77 type: user_choice {0}
The square root of x.

`ierr`
Intent: out
C type: (na) integer return value
Fortran type: UPS_KIND_INT4 {0}
Fortran77 type: UPS_KIND_INT4 {0}
Error return value

ReturnValues

Returns UPS_OK if successful.

Discussion

UPS_UT_Square_root provides coding convenience and readability.

UPS_UT_Time_wall_get()

Package

ut

Purpose

UPS_UT_Time_get returns an opaque handle related to the time this routine is called.

Usage

C	`ierr = UPS_UT_Time_wall_get`	`(time_handle);`
Fortran	`call UPSF_UT_TIME_WALL_GET`	`(time_handle, ierr)`
Fortran77	`call UPS_UT_TIME_WALL_GET`	`(time_handle, ierr)`

Arguments

`time_handle`

Intent:	out
C type:	UPS_DT_TIME_TYPE*
Fortran type:	UPS_KIND_REAL8 {1:UPS_DT_TIME_TYPE_DT}
Fortran77 type:	UPS_KIND_REAL8 {1:UPS_DT_TIME_TYPE_DT}

An opaque handle relating to a time entity.
Fortran users must pass in an array of sufficient
length to hold the time_handle variable.
The number of bytes used for this time_handle can be
obtained with a call to UPSF_DT_SIZEOF() with the
datatype argument being UPS_DT_TIME_TYPE_DT.
This size is probably 16 bytes so an array of
2 elements will be enough.

`ierr`

Intent:	out
C type:	(na) int return value
Fortran type:	UPS_KIND_INT4 {0}
Fortran77 type:	UPS_KIND_INT4 {0}

Return status. Returns UPS_OK if successful.

ReturnValues

Returns UPS_OK if successful.

Discussion

The handles returned are not meant to be viewed directly by the user. A call to UPS_UT_Time_wall_interval is needed.

NOTE: For increased performance and to allow ups init/terminate routines to call the statistics routine, this routine does not check for UPS initialization.

Examples

The following is a fortran example of the timer calls.
Fortran:

```
! ----------------------------------------------
! number of doubles to hold UPS_DT_TIME_TYPE
! ----------------------------------------------
integer(KIND=UPS_KIND_INT4), parameter :: SIZE_FOR_TIME_HANDLE = 2

real(KIND=UPS_KIND_REAL8)   ::  &
    time_start(SIZE_FOR_TIME_HANDLE), &
    time_stop(SIZE_FOR_TIME_HANDLE), &
    time_interval

call UPSF_AA_INIT( ierr )

call UPSF_UT_TIME_WALL_GET( time_start, ierr )

<user code>

CALL UPSF_UT_TIME_WALL_GET( time_stop, ierr )

CALL UPSF_UT_TIME_WALL_INTERVAL( time_start, time_stop, time_interval)

<user code>

CALL UPSF_AA_TERMINATE ( ierr )
```

SeeAlso

UPS_UT_Time_wall_interval (page 252)

UPS_UT_Time_wall_interval()

Package

ut

Purpose

Return the time in seconds given the two time handles.

Usage

C	`ierr = UPS_UT_Time_wall_interval`	`(time_handle_a, time_handle_b, time_sec);`
Fortran	`call UPSF_UT_TIME_WALL_INTERVAL`	`(time_handle_a, time_handle_b, time_sec, ierr)`
Fortran77	`call UPS_UT_TIME_WALL_INTERVAL`	`(time_handle_a, time_handle_b, time_sec, ierr)`

Arguments

`time_handle_a`

Intent:	in
C type:	UPS_DT_TIME_TYPE
Fortran type:	UPS_KIND_REAL8 {1:UPS_DT_TIME_TYPE_DT}
Fortran77 type:	UPS_KIND_REAL8 {1:UPS_DT_TIME_TYPE_DT}

An opaque handle relating to a time entity.
See UPS_UT_Time_wall_get (page 251)
for requirements of the size of the array for Fortran users.

`time_handle_b`	*Intent*:	in
	C type:	UPS_DT_TIME_TYPE
	Fortran type:	UPS_KIND_REAL8 {1:UPS_DT_TIME_TYPE_DT}
	Fortran77 type:	UPS_KIND_REAL8 {1:UPS_DT_TIME_TYPE_DT}
	An opaque handle relating to a time entity. See UPS_UT_Time_wall_get (page 251) for requirements of the size of the array for Fortran users.	
`time_sec`	*Intent*:	out
	C type:	double*
	Fortran type:	UPS_KIND_REAL8 {0}
	Fortran77 type:	UPS_KIND_REAL8 {0}
	The time in seconds between the 2 time handles given.	
`ierr`	*Intent*:	out
	C type:	(na) int return value
	Fortran type:	UPS_KIND_INT4 {0}
	Fortran77 type:	UPS_KIND_INT4 {0}
	Return status. Returns UPS_OK if successful.	

ReturnValues

Returns UPS_OK if successful.

Discussion

NOTE: For increased performance and to allow ups init/terminate routines to call the statistics routine, this routine does not check for UPS initialization.

Examples

See UPS_UT_Time_wall_get (page 251) for a code example.

SeeAlso

`UPS_UT_Time_wall_get` (page 251)

C.10 Reference Pages Index

Listing and reference page number of routines

aa package Routine Name	Reference Page
UPS_AA_Abort	91
UPS_AA_Init	92
UPS_AA_Io_pe_get	95
UPS_AA_Io_pe_set	96
UPS_AA_Opt_get	97
UPS_AA_Opt_set	98
UPS_AA_Statistics	99
UPS_AA_Terminate	100

cm package Routine Name	Reference Page
UPS_CM_Allgather	102
UPS_CM_Allreduce	103
UPS_CM_Barrier	104
UPS_CM_Barrier_idle	105
UPS_CM_Bcast	106
UPS_CM_Context_free	107
UPS_CM_Get_context	108
UPS_CM_Get_numpes	110
UPS_CM_Get_penum	110
UPS_CM_P_group_item	111
UPS_CM_Reduce	115
UPS_CM_Salltoall	117
UPS_CM_Set_context	118
UPS_CM_Sm_free	119
UPS_CM_Sm_get_item	120
UPS_CM_Sm_malloc	122
UPS_CM_Sm_set_item	124

dp package Routine Name	Reference Page
UPS_DP_Combiner	126
UPS_DP_Combinerm	127
UPS_DP_Count_mask	129
UPS_DP_Dot_product	130
UPS_DP_Dot_productm	131
UPS_DP_Number_mask	133
UPS_DP_Sort	134
UPS_DP_Vector_norm	135
UPS_DP_Vector_normm	136

dt package Routine Name	Reference Page
UPS_DT_Sizeof	139

er package Routine Name	Reference Page
UPS_ER_Get_wait_time	140
UPS_ER_Perror	141
UPS_ER_Set_alarm	142
UPS_ER_Set_wait_time	143
UPS_ER_Unset_alarm	144

gs package Routine Name	Reference Page
UPS_GS_Collate	146
UPS_GS_Distribute	147
UPS_GS_Free	148
UPS_GS_Gather	149
UPS_GS_Gather_list	151
UPS_GS_Gather_multi	153
UPS_GS_Get_item	155
UPS_GS_Scatter	156
UPS_GS_Scatter_list	158
UPS_GS_Scatter_multi	163
UPS_GS_Setup	165
UPS_GS_Setup_s_global	167
UPS_GS_Setup_s_local	170
UPS_GS_Setup_study	173

io package Routine Name	Reference Page
UPS_IO_Attr_read	175
UPS_IO_Attr_write	178
UPS_IO_Attr_write_s	180
UPS_IO_Dataset_read	183
UPS_IO_Dataset_write	191
UPS_IO_Ds_r_s	193
UPS_IO_Ds_w_s	195
UPS_IO_File_close	197
UPS_IO_File_open	198
UPS_IO_File_type	205
UPS_IO_Filter_get	206
UPS_IO_Filter_set	208
UPS_IO_Group_close	209
UPS_IO_Group_open	210
UPS_IO_Info_count	212
UPS_IO_Info_create	214
UPS_IO_Info_create_self	217
UPS_IO_Info_free	219
UPS_IO_Info_item_get	220
UPS_IO_Info_item_set	222
UPS_IO_Loc_item_get	224
UPS_IO_Loc_item_set	225
UPS_IO_Rm	227

`ut package Routine Name`	`Reference Page`
UPS_UT_Binary_op	230
UPS_UT_Binary_opm	231
UPS_UT_Checksum_get	233
UPS_UT_Convert	235
UPS_UT_Dt_change	236
UPS_UT_Get_name_or_value	237
UPS_UT_Loc_struct_alloc	240
UPS_UT_Loc_structure	241
UPS_UT_Mem_get_item	242
UPS_UT_Reduce_op	244
UPS_UT_Reduce_opm	246
UPS_UT_Sleep	248
UPS_UT_Sort_compress	249
UPS_UT_Square_root	250
UPS_UT_Time_wall_get	251
UPS_UT_Time_wall_interval	252