POSIX Utilities Package 2.2 Reference Manual

Vilas K. Chitrakaran

Sat Aug 26 17:12:56 2006

Contents

Chapter 1

POSIX Utilities Package 2.2 Hierarchical Index

1.1 POSIX Utilities Package 2.2 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

POSIX Utilities Package 2.2 Class Index

2.1 POSIX Utilities Package 2.2 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

POSIX Utilities Package 2.2 Class Documentation

3.1 ErrnoException Class Reference

A mechanism for basic run-time exception handling.

`#include <ErrnoException.hpp>`

Public Member Functions

- ErrnoException ()
- ErrnoException (int error, const char *desc=NULL)
- ErrnoException (const ErrnoException &e)
- ErrnoException & operator= (const ErrnoException &e)
- void setError (int error, const char *desc=NULL)
- int getErrorCode () const
- const char * getErrorDesc () const

3.1.1 Detailed Description

A mechanism for basic run-time exception handling.

Example Program:

```
//===========================================================================
// ErrnoException.t.cpp - Example program for ErrnoException class.
//
// Author        : Vilas Kumar Chitrakaran
//===========================================================================

//===========================================================================
// NOTE: For older versions of gcc ( <= 2.96 ) make sure that you compile with
//       exceptions enabled (-fexceptions switch) to avoid SIGABRT on exception.
//===========================================================================

#include "ErrnoException.hpp"
#include <iostream>
#include <string>
```

```
using namespace std;

// Just an example function that shows how
// to throw an exception
void enterNumberBelowFive(int number)
{
 // throw an exception for invalid argument
 if(number >= 5)
  throw ErrnoException(EINVAL,"[enterNumberBelowFive]");
}


// The main function
int main()
{

 // Try something that may cause error
 try
 {
  cout << "First call to enterNumberBelowFive() with arg = 2 ... ";
  enterNumberBelowFive(2);    // this will go through
  cout << "worked." << endl;

  cout << "Second call to enterNumberBelowFive() with arg = 10 ... ";
  enterNumberBelowFive(10);   // this will throw exception
  cout << "worked." << endl;  // this line should not print
 }

 // catch the first error that was thrown from within try block....
 catch(ErrnoException ex)
 {
  cout << "caught exception: " << ex.getErrorDesc() << ": "
       << strerror(ex.getErrorCode()) << endl;

  // put error recovery code here, based on type of exception!
 }

 return 0;
}
```

3.1.2 Constructor & Destructor Documentation

3.1.2.1 ErrnoException::ErrnoException () [inline]

Standard constructor sets error to 0 (no error)

3.1.2.2 ErrnoException::ErrnoException (int *error*, const char * *desc* = NULL) [inline]

This constructor allows initialization

Parameters:

error set integer error code. (0 reserved for no error)

desc set a short description [less than 40 chars], possibly just the object that set the error.

3.1.2.3 ErrnoException::ErrnoException (const ErrnoException & *e*) [inline]

Copy constructor

3.1.3 Member Function Documentation

3.1.3.1 int ErrnoException::getErrorCode () const [inline]

Returns:
latest error code. (0 means no error).

3.1.3.2 const char* ErrnoException::getErrorDesc () const [inline]

Returns:
any descriptive message that was set with the error.

3.1.3.3 ErrnoException& ErrnoException::operator= (const ErrnoException & *e*) [inline]

Assignment operation

3.1.3.4 void ErrnoException::setError (int *error*, const char * *desc* = NULL) [inline]

Destructor does nothing Set an error

Parameters:
error set integer error code. (0 reserved for no error)
desc set a short description [less than 40 chars], possibly just the object that set the error.

The documentation for this class was generated from the following file:

- ErrnoException.hpp

3.2 MessageQueue Class Reference

Inter-process messaging mechanism.

`#include <MessageQueue.hpp>`

Public Member Functions

- MessageQueue ()
- ~MessageQueue ()
- int create (const char *name, int maxNumMsgs, int maxMsgLen=1024)
- int open (const char *name)
- int close ()
- int unlink ()
- int trySend (const char *msgBuffer, int msgSize)
- int send (const char *msgBuffer, int msgSize)
- int tryReceive (char *msgBuffer, int bufSize)
- int receive (char *msgBuffer, int bufSize)
- int notify (const struct sigevent *notification)
- int getMaxNumMsgs () const
- int getMaxMsgLength () const
- int getErrnoError () const

3.2.1 Detailed Description

Inter-process messaging mechanism.

This class provides a wrapper for non-blocking message queues. Messages are sent at the priority of the sending process, and received highest priority first. Messages of equal priority are received on a first-come-first-serve basis. When developing a client-server system using a MessageQueue object, run the server at maximum possible priority as it waits for messages.

This is an efficient mechanism only for sending small messages (because messages get 'copied' from sender to OS, and then OS to receiver). Use SharedMemory for passing large amounts of data between processes.

Example Program:

```
//==========================================================================
// MessageQueue.t.cpp - Example program for MessageQueue class.
//
// Author        : Vilas Kumar Chitrakaran
//==========================================================================

#include "MessageQueue.hpp"
#include <iostream>

using namespace std;

//==========================================================================
// receiver thread
// - Creates a queue that can hold 2 messages at a time, each 1 byte long
// - Waits for, receives and prints 3 messages before exiting
//==========================================================================
void *receiver(void *arg)
{
 arg = arg;
```

```
 MessageQueue rq;
 char rbuf[10];

 int numMsgs = 2;
 int msgSize = 1 * sizeof(char);

 int m = 0;

 // create receive queue
 if( rq.create("/rq", numMsgs, msgSize) == -1)
 {
  cout << "receiver: " << strerror(rq.getErrnoError()) << endl << flush;
  return NULL;
 }

 // stats
 cout << "receiver: queue created for " << rq.getMaxNumMsgs() << " messages, "
      << rq.getMaxMsgLength() << " bytes long." << endl << flush;

 while(m < 3)
 {
  // receive a message from someone
  strncpy(rbuf, "\0", 10);
  if( rq.receive(rbuf, msgSize) == -1)
   cout << "receiver: " << strerror(rq.getErrnoError()) << endl << flush;
  else
   cout << "receiver: received msg: " << rbuf << endl << flush;
  m++;
 }

 // exit
 if( rq.unlink() == -1)
   cout << "receiver: " << strerror(rq.getErrnoError()) << endl << flush;
 cout << "receiver: exiting" << endl << flush;
 return NULL;
}


//===========================================================================
// sender thread
// - opens queue created by receiver
// - sends messages from user without blocking
//===========================================================================
int sender()
{
 MessageQueue sq;
 char sbuf;

 // open queue
 if( sq.open("/rq") == -1)
 {
  cout << "sender: " << strerror(sq.getErrnoError()) << endl << flush;
  return -1;
 }

 // stats
 cout << "sender: queue opened for " << sq.getMaxNumMsgs() << " messages, "
      << sq.getMaxMsgLength() << " bytes long." << endl << flush;

 while(1)
 {
  // send message
  cout << endl << "sender: Enter message: " << endl << flush;
  cin >> sbuf;

  if ( sq.trySend(&sbuf, sizeof(char)) == -1 )
   cout << "sender: " << strerror(sq.getErrnoError())
```

```
        << ". Queue is full." << endl << flush;
 }
 return 0;
}


//==========================================================================
// main function
//==========================================================================
int main()
{
 pthread_t threadId;
 pthread_create(&threadId, NULL, &receiver, NULL);
 sleep(1);
 sender();
 return 0;
}
```

3.2.2 Constructor & Destructor Documentation

3.2.2.1 MessageQueue::MessageQueue ()

Default constructor does nothing.

3.2.2.2 MessageQueue::~MessageQueue ()

Default destructor deletes the message queue if it was created by the object.

3.2.3 Member Function Documentation

3.2.3.1 int MessageQueue::close ()

Release access to the message queue. Note that the queue and any messages it may contain are not deleted, and can be opened again.

Returns:

0 on sucess, -1 on error, 'errno' is set and can be retrieved by a call to getErrnoError().

3.2.3.2 int MessageQueue::create (const char * *name*, int *maxNumMsgs*, int *maxMsgLen* = 1024)

Creates a message queue object with read/write access. Note that if a message queue with the same name already exists, this function will exit with an error (errno set).

Parameters:

name Name of the message queue. For portability, the name should begin with a leading "/" and contain no other "/" characters.

maxNumMsgs Number of messages the queue must hold

maxMsgLen Maximum possible length (bytes) of each message (default 1kB)

Returns:

0 on sucess, -1 on error, 'errno' is set and can be retrieved by a call to getErrnoError().

3.2.3.3 int MessageQueue::getErrnoError () const [inline]

Returns:
Error code from the last error that happened before this function was called, else 0. Error codes are defined in the standard header errno.h

3.2.3.4 int MessageQueue::getMaxMsgLength () const [inline]

Returns:
The maximum possible length (in bytes) for a message in queue.

3.2.3.5 int MessageQueue::getMaxNumMsgs () const [inline]

Returns:
The maximum number of messages the queue can hold.

3.2.3.6 int MessageQueue::notify (const struct sigevent * *notification*)

Notify the calling process asynchronously if a message appeared in the queue. This is useful if you don't want to keep polling the queue to find whether a new message has arrived, for instance in handling emergency messages. (See pp. 107, Programming for the Real World, POSIX.4 for an example.)

Parameters:
notification NULL or a pointer to sigevent structure that describes how you want to be notified.

Returns:
0 on sucess, -1 on error, 'errno' is set and can be retrieved by a call to getErrnoError().

3.2.3.7 int MessageQueue::open (const char * *name*)

Opens an existing message queue for read/write access. Note that if the queue does not exist, this function will return with an error (errno set).

Parameters:
name Name of the queue to establish connection with.

Returns:
0 on sucess, -1 on error, 'errno' is set and can be retrieved by a call to getErrnoError().

3.2.3.8 int MessageQueue::receive (char * *msgBuffer*, int *bufSize*)

Removes a message from the head of the queue. If the queue is empty, it will block until there is something to read.

Parameters:
msgBuffer Buffer for received message.

bufSize Size of the buffer (must be atleast as large as the maximum message size for the queue).

Returns:
Number of bytes in the received message, -1 on error. If the queue is empty, this function will return -1 with errno set to EAGAIN (which can be retrieved by a call to getErrnoError()).

3.2.3.9 int MessageQueue::send (const char * *msgBuffer*, int *msgSize*)

Send a message. If the queue is full, this function blocks until the message queue empties and the message can be placed in the queue

Parameters:
msgBuffer Buffer containing message to be sent.

msgSize Size of the message.

Returns:
0 on sucess, -1 on error, 'errno' is set and can be retrieved by a call to getErrnoError().

3.2.3.10 int MessageQueue::tryReceive (char * *msgBuffer*, int *bufSize*)

Removes a message from the head of the queue. If the queue is empty, returns immediately without blocking.

Parameters:
msgBuffer Buffer for received message.

bufSize Size of the buffer (must be atleast as large as the maximum message size for the queue).

Returns:
Number of bytes in the received message, -1 on error. If the queue is empty, this function will return -1 with errno set to EAGAIN (which can be retrieved by a call to getErrnoError()).

3.2.3.11 int MessageQueue::trySend (const char * *msgBuffer*, int *msgSize*)

Send a message without blocking.

Parameters:
msgBuffer Buffer containing message to be sent.

msgSize Size of the message.

Returns:
0 on sucess, -1 on error. If the queue is full, this function will return -1 with errno set to EAGAIN (which can be retrieved by a call to getErrnoError()).

3.2.3.12 int MessageQueue::unlink ()

If the process calling this function created the message queue, this function calls close() and marks the message queue for deletion. Message queues are persistant, i.e., if there are processes that have the queue open when this function is called then the destruction of the queue is delayed until all processes have closed their access to the queue (by calling close()).

Returns:

0 on sucess, -1 on error, 'errno' is set and can be retrieved by a call to getErrnoError(). This function will return -1 if it is called by an object that did not create the message queue.

The documentation for this class was generated from the following file:

- MessageQueue.hpp

3.3 PtBarrier Class Reference

The pthread barrier synchronization object.

`#include <PtBarrier.hpp>`

Public Member Functions

- PtBarrier (int n)
- ~PtBarrier ()
- void wait ()

3.3.1 Detailed Description

The pthread barrier synchronization object.

- A barrier can be created and used to synchronize a bunch of threads. A barrier object common to multiple threads blocks each of the threads until all of them have reached a certain point in their code, at which point they are all released.
- This class will throw an exception of type ErrnoException in case of errors.

Example Program:

```
//============================================================================
// PtBarrier.t.cpp - Example program for PtBarrier class.
//
// Author        : Vilas Kumar Chitrakaran
//============================================================================

#include "PtBarrier.hpp"
#include <iostream>
#include <time.h>

using namespace std;

PtBarrier barrier(2);
// The synchronization object for two threads

//============================================================================
// work
// - does some work
// - waits at the barrier for the other thread to finish its work cycle
// - loop again
//============================================================================
void *work(void *arg)
{
 struct timespec napTime;
 napTime.tv_sec = (int)arg;
 napTime.tv_nsec = 0;
 while (1)
 {
  nanosleep( &napTime, NULL ); // work!!
  cout << "Thread " << pthread_self() << " worked "
       << napTime.tv_sec << " secs."
       << endl << flush;

  // wait for the other thread
  barrier.wait();
```

```
  cout << "Thread " << pthread_self() << " sync."
       << endl << flush;
 }
 return NULL;
}


//===========================================================================
// main function
// - creates two synchronized threads that wait for each other
//   at the end of each cycle.
//===========================================================================
int main()
{
 pthread_t threadId;
 int arg;
 arg = 1;
 pthread_create(&threadId, NULL, work, (void *)arg);
 arg = 2;
 work((void *)arg);
 return 0;
}
```

3.3.2 Constructor & Destructor Documentation

3.3.2.1 PtBarrier::PtBarrier (int *n*) [inline]

Initialize a barrier object.

Parameters:

n The number of threads that must call wait() before any of them successfully returns from the call. This value must be greater than 0.

3.3.2.2 PtBarrier::~PtBarrier () [inline]

Destroy the barrier object

3.3.3 Member Function Documentation

3.3.3.1 void PtBarrier::wait () [inline]

Synchronize participating threads at the barrier. NOTE: For 'n' cooperating threads as specified in the constructor:

- This function blocks until 'n-1' other participating threads have called wait() on the same barrier.
- You can't unblock this function by calling wait() 'n' times from the same thread.

The documentation for this class was generated from the following file:

- PtBarrier.hpp

3.4 RecursiveMutex Class Reference

A wrapper for pthread mutex, with the added functionality that it is recursive.

`#include <RecursiveMutex.hpp>`

Public Member Functions

- RecursiveMutex ()
- ~RecursiveMutex ()
- void lock ()
- void unlock ()
- int tryLock ()

3.4.1 Detailed Description

A wrapper for pthread mutex, with the added functionality that it is recursive.

- A recursive mutex can be locked more than once by a thread without causing a deadlock.
- The thread must call the unlock routine on the mutex the same number of times that it called the lock routine before another thread can lock the same mutex.
- This class is useful if the thread is already in a mutex protected section of the code and needs to call another routine that locks the same mutex again.
- This class will throw an exception of type ErrnoException in case of errors.

The error checking code snippet used here is similar to John Nagle's mutexlock.h.

Example Program:

```
//============================================================================
// RecursiveMutex.t.cpp - Example program for RecursiveMutex class.
//
// Author        : Vilas Kumar Chitrakaran
//============================================================================


#include "RecursiveMutex.hpp"
#include <iostream>
#include <string.h>
#include <time.h>
#include <stdlib.h>

//============================================================================
// This program demostrates how access to a resource shared between two
// threads is controlled using a RecursiveMutex object.
//============================================================================

using namespace std;

RecursiveMutex mutex; // mutex object
static int counter;     // shared resource

//============================================================================
// Consumer thread
//============================================================================
int consume()
```

```
{
 struct timespec delay;
 delay.tv_sec = 0;
 delay.tv_nsec = (long int)5e8;
 while(1)
 {
  if(mutex.tryLock() == -1)
   cout << "CONSUME: missed" << endl << flush;
  else
  {
   cout << "CONSUME : " << --counter << endl << flush;
   mutex.unlock();
  }
  nanosleep(&delay, NULL);
 }
 return 0;
}

//===========================================================================
// Producer thread
// NOTE that the mutex is recursively locked twice
//===========================================================================
void *produce(void *arg)
{
 arg=arg;
 struct timespec delay;
 delay.tv_sec = 0;
 delay.tv_nsec = (long int)1e8;
 while(1)
 {
  mutex.lock();
  mutex.lock(); // This is valid!
  cout << "PRODUCE: " << ++counter << endl << flush;
  mutex.unlock();
  nanosleep(&delay, NULL); // sleep a little before releasing second mutex
  mutex.unlock();
  nanosleep(&delay, NULL);
 }
 return NULL;
}



//===========================================================================
// main function
//===========================================================================
int main()
{
 pthread_t threadId;
 pthread_create(&threadId, NULL, &produce, NULL);
 consume();
 return 0;
}
```

3.4.2 Constructor & Destructor Documentation

3.4.2.1 RecursiveMutex::RecursiveMutex () [inline]

Constructs a recursive mutex

3.4.2.2 RecursiveMutex::~RecursiveMutex () [inline]

Deletes a recursive mutex

3.4.3 Member Function Documentation

3.4.3.1 void RecursiveMutex::lock () [inline]

Locks a recursive mutex. If the mutex is locked by another thread, this thread is blocked until the mutex gets unlocked.

3.4.3.2 int RecursiveMutex::tryLock () [inline]

Returns 0 and locks the mutex if it is not already locked by another thread, else returns -1.

3.4.3.3 void RecursiveMutex::unlock () [inline]

Unlocks a recursive mutex.

The documentation for this class was generated from the following file:

- RecursiveMutex.hpp

3.5 RWLock Class Reference

The pthread reader-writer lock.

`#include <RWLock.hpp>`

Public Member Functions

- RWLock ()
- ~RWLock ()
- void readLock ()
- int tryReadLock ()
- void writeLock ()
- int tryWriteLock ()
- void unlock ()

3.5.1 Detailed Description

The pthread reader-writer lock.

- A Reader-Writer lock allows concurrent access to multiple processes for reading shared data, but restricts writing to shared data only when no readers are present.
- Conversely, when a writer has access to shared data, all other writers and readers are blocked until the writer is done.
- This class will throw an exception of type ErrnoException in case of errors.

Example Program:

```
//============================================================================
// RWLock.t.cpp - Example program for RWLock class.
//
// Author        : Vilas Kumar Chitrakaran
//============================================================================

#include "RWLock.hpp"
#include <iostream>
#include <math.h>
#include <stdlib.h>

using namespace std;

// A structure for checking accounts
typedef struct
{
 double balance;
 RWLock key;
}bank_account;


// My checking account
bank_account myAccnt;


//========================================================================
// balance
//========================================================================
```

```
void *balance(void *)
{
 while(1)
 {
  if( myAccnt.key.tryReadLock() == -1 )
   cout << "Account info: Busy updating" << endl;
  else
  {
   cout << "Account info: $ " << myAccnt.balance << endl;
   myAccnt.key.unlock();
  }
 }
 return NULL;
}


//=========================================================================
// credit
//=========================================================================
void credit(double amount)
{
 myAccnt.key.writeLock();
 myAccnt.balance += amount;
 myAccnt.key.unlock();
}


//=========================================================================
// main function
//=========================================================================
int main()
{
 double amnt;
 pthread_t threadId;
 pthread_create(&threadId, NULL, balance, NULL);
 while(1)
 {
  amnt = 10.0 * (rand()/(double)RAND_MAX - 0.5);
  credit(amnt);
 }
 return 0;
}
```

3.5.2 Constructor & Destructor Documentation

3.5.2.1 RWLock::RWLock () [inline]

Constructor initializes the lock.

3.5.2.2 RWLock::~RWLock () [inline]

Destroys the lock.

3.5.3 Member Function Documentation

3.5.3.1 void RWLock::readLock () [inline]

Acquire the shared lock for read access. If the lock is not available, block until it is.

3.5.3.2 int RWLock::tryReadLock () [inline]

Try to acquire the shared lock for read access. If the lock is not available, return immediately.

Returns:
0 on successful acquisition of lock, else -1

3.5.3.3 int RWLock::tryWriteLock () [inline]

Try to acquire the shared lock for exclusive write access. If the lock is not available, return immediately.

Returns:
0 on successful acquisition of lock, else -1

3.5.3.4 void RWLock::unlock () [inline]

Unlock the shared lock. If the calling thread doesn't own the lock, the behavior of this function is undefined.

3.5.3.5 void RWLock::writeLock () [inline]

Acquire the shared lock for exclusive write access. If the lock is not available, block until it is.

The documentation for this class was generated from the following file:

- RWLock.hpp

3.6 ShMem Class Reference

Shared memory objects.

`#include <ShMem.hpp>`

Public Member Functions

- ShMem ()
- ~ShMem ()
- void * create (const char *name, int size)
- void * open (const char *name, int size)
- int close ()
- int unlink ()
- int getErrnoError () const

3.6.1 Detailed Description

Shared memory objects.

This class provides just the basic shared memory functionality. The user must provide the facility for synchronization of access to the shared object between multiple processes (using memory based semaphores, etc - see pp. 143, Programming for the Real World, POSIX.4).

Example Program:

```
//============================================================================
// ShMem.t.cpp - Example program for creating and writing
//                            into shared memory.
//
// Author        : Vilas Kumar Chitrakaran
//============================================================================

#include <stdio.h>
#include <iostream>
#include <errno.h>
#include <pthread.h>
#include <unistd.h>
#include "ShMem.hpp"

using namespace std;

//============================================================================
// writer thread
// - Creates shared memory
// - modifies contents continuously
//============================================================================
void *writer(void *arg)
{
 arg = arg;
 ShMem shm;
 double *counter;

 // create shared memory
 counter = (double *)shm.create( "/shm0", sizeof(double) );
 if( counter == NULL )
 {
  cout << "writer: " << strerror(shm.getErrnoError()) << endl;
  return NULL;
 }
```

```
 // change shared memory
 *counter = 0;
 while(*counter < 10)
 {
  cout << "writer: " << ++(*counter) << endl;;
  sleep(1);
 }

 // unlink
 shm.unlink();

 return NULL;
}

//============================================================================
// reader thread
// - opens shared memory created by writer
// - reads shared memory continuously
//============================================================================
int reader()
{
 ShMem shm;
 double *counter;

 // open existing shared memory
 counter = (double *)shm.open( "/shm0", sizeof(double) );
 if( counter == NULL )
 {
  cout << "reader: " << strerror(shm.getErrnoError()) << endl;
  return -1;
 }

 // read shared memory
 while(*counter < 10)
 {
  cout << "reader: " << *counter << endl;
  sleep(1);
 }

 // close
 shm.close();

 return 0;
}


//============================================================================
// main function
//============================================================================
int main()
{
 pthread_t threadId;
 pthread_create(&threadId, NULL, &writer, NULL);
 sleep(1);
 reader();
 return 0;
}
```

3.6.2 Constructor & Destructor Documentation

3.6.2.1 ShMem::ShMem ()

Default constructor does nothing

3.6.2.2 ShMem::~ShMem ()

Default destructor deletes shared memory region if it was created by the object.

3.6.3 Member Function Documentation

3.6.3.1 int ShMem::close ()

Unmaps the shared memory from process address space and closes the memory region to further access. The shared memory region and its contents are however not deleted and can be opened again (similar to a file open and close operation).

Returns:
0 on sucess, -1 on error, 'errno' is set and can be retrieved by a call to getErrnoError().

3.6.3.2 void* ShMem::create (const char * *name*, int *size*)

Creates a shared memory object with read/write access and maps it to your process address space. Note that if the shared memory object already exists this function will exit with an error (errno set).

Parameters:
name Name of the shared memory object. For portability, the name should begin with a leading "/" and contain no other "/" characters.
size Size (number of bytes) of the shared memory object

Returns:
If successful, a pointer to the starting memory location of shared memory, else NULL, 'errno' is set and can be retrieved by a call to getErrnoError().

3.6.3.3 int ShMem::getErrnoError () const [inline]

Returns:
Error code from the last error that happened before this function was called, else 0. Error codes are defined in the standard header errno.h

3.6.3.4 void* ShMem::open (const char * *name*, int *size*)

Opens a shared memory object for read/write and maps it to your process address space. Note that if the shared memory object doesn't exist this function will exit with an error (errno set).

Parameters:
name Name of the shared memory object. For portability, the name should begin with a leading "/" and contain no other "/" characters.
size Size (number of bytes) of the shared memory object

Returns:
If successful, a pointer to the starting memory location of shared memory, else NULL, 'errno' is set and can be retrieved by a call to getErrnoError().

3.6.3.5 int ShMem::unlink ()

If the process calling this function created the shared memory, this function calls close() and marks the shared memory for deletion. Shared memory objects are persistant, i.e., if there are processes that have the object open when this function is called then the destruction of the object is delayed until all processes have closed their access to the object (by calling close()).

Returns:

0 on sucess, -1 on error, 'errno' is set and can be retrieved by a call to getErrnoError(). NOTE: This function will return -1 if it is called by an object that did not create the shared memory.

The documentation for this class was generated from the following file:

- ShMem.hpp

3.7 StatusReport Class Reference

An object for storing status messages containing a integer code, a message and a timestamp.

`#include <StatusReport.hpp>`

Public Member Functions

- StatusReport (int maxMsgLen=80, int maxNumMsgs=1, SR_buffer_type type=SR_-CIRCULAR)
- ~StatusReport ()
- void setReport (int code, const char *message=NULL)
- const char * getReportMessage (unsigned int reportNum=1) const
- int getReportCode (unsigned int reportNum=1) const
- timespec getReportTimestamp (unsigned int reportNum=1) const
- void clearReports ()
- unsigned int getNumReports () const
- unsigned int getNumReportsOverflow () const

3.7.1 Detailed Description

An object for storing status messages containing a integer code, a message and a timestamp.

Objects of this class can be embedded in any software module that requires status reporting capabilities to users or other connected subsystems. Notes about usability to your application:

- User must specify the maximum number of reports to store and maximum possible length of reports apriori.
- Reports longer than maximum specified report length will get truncated.
- If the buffer is initialized as circular, once the buffer is full new messages will overwrite oldest messages from the beginning of the buffer so that if the buffer size is 'n', you will always have the last 'n' messages available to you.
- The class constructor does dynamic memory allocation. Create objects of this class outside realtime code.

Example Program:

```
//============================================================================
// StatusReport.t.cpp - Example program for StatusReport class.
//
// Author        : Vilas Kumar Chitrakaran
//============================================================================

#include "StatusReport.hpp"
#include <iostream>
#include <unistd.h>

using namespace std;

// use message
void usage(char *argv[])
{
 cout << endl
```

```
      << "Usage: " << argv[0] << " -[cl]" << endl
      << " where:" << endl
      << "  -c          use circular buffer" << endl
      << "  -l          use linear buffer" << endl << endl;
}

//==========================================================================
// main function
//==========================================================================
int main(int argc, char *argv[])
{
 StatusReport *buffer;
 char timeBuf[26];
 time_t time;
 unsigned int i;
 int opt;
 SR_buffer_type type = SR_CIRCULAR;

 // check command line arguments
 if(argc <= 1)
 {
  usage(argv);
  return 0;
 }

 // parse command line option for type of buffer
 while( (opt = getopt(argc, argv, "cl")) != -1)
 {
  switch(opt)
  {
   case 'c': // circular buffer
    type = SR_CIRCULAR;
    break;
   case 'l': // linear buffer
    type = SR_LINEAR;
    break;
   default:
    usage(argv);
    return 0;
    break;
  }
 }

 // create buffer for 2 messages 80 chars long
 buffer = new StatusReport(80,2,type);

 // Add two error messages
 buffer->setReport(0x100);
 buffer->setReport(0x0, "Second message");

 // Add another message - this will overflow
 // for linear buffer and replace first message
 // for a circular buffer
 buffer->setReport(0x300, "Third message");

 // Print the reports
 cout << "num reports: " << buffer->getNumReports() << endl
      << "num overflow: " << buffer->getNumReportsOverflow() << endl;
 for(i = 1; i <= buffer->getNumReports(); i++)
 {
  time = (time_t)buffer->getReportTimestamp(i).tv_sec;
  ctime_r(&time, timeBuf);
  timeBuf[24] = '\0';
  cout << "[Report: " << (dec) << i
       << "] [mesg: " << buffer->getReportMessage(i)
       << "] [code: " << "0x" << (hex) << buffer->getReportCode(i)
       << "] [time: " << timeBuf << "]" << endl;
```

```
 }

 delete buffer;

 return 0;
}
```

3.7.2 Constructor & Destructor Documentation

3.7.2.1 StatusReport::StatusReport (int *maxMsgLen* = 80, int *maxNumMsgs* = 1, SR_buffer_type *type* = SR_CIRCULAR)

The constructor. Creates required buffers and initializes.

Parameters:

maxMsgLen Maximum length of messages in the buffer (default = 80 chars).

maxNumMsgs Number of messages the buffer can hold (default = 1).

type The type of buffer, either circular (SR_CIRCULAR) or linear (SR_LINEAR). If the buffer is initialized as circular, once the buffer is full new messages will overwrite oldest messages from the beginning of the buffer so that if the buffer size is 'n', you will always have the last 'n' messages available to you (default SR_CIRCULAR).

3.7.2.2 StatusReport::~StatusReport ()

The destructor. Frees allocated memory.

3.7.3 Member Function Documentation

3.7.3.1 void StatusReport::clearReports ()

Clear all reports.

3.7.3.2 unsigned int StatusReport::getNumReports () const

Return the number of reports available to the user from the reports buffer.

3.7.3.3 unsigned int StatusReport::getNumReportsOverflow () const

Return the number of reports that were not recorded because of insufficient buffer space. If the buffer is circular this function will return number of messages lost because they were overwritten.

3.7.3.4 int StatusReport::getReportCode (unsigned int *reportNum* = 1) const

Return a report code.

Parameters:

reportNum The desired report number for which you want the message. Note that reports are collected first-in last out. Hence if no report number is specified, the most recent report is returned; i.e. report number 1 is most recent.

3.7.3.5 const char* StatusReport::getReportMessage (unsigned int *reportNum* = 1) const

Return a report message.

Parameters:

reportNum The desired report number for which you want the message. Note that reports are collected first-in last out. Hence if no report number is specified, the most recent report is returned; i.e. report number 1 is most recent.

3.7.3.6 struct timespec StatusReport::getReportTimestamp (unsigned int *reportNum* = 1) const

Return the timestamp when a report was received.

Parameters:

reportNum The desired report number for which you want the message. Note that reports are collected first-in last out. Hence if no report number is specified, the most recent report is returned; i.e. report number 1 is most recent.

3.7.3.7 void StatusReport::setReport (int *code*, const char * *message* = NULL)

Add a report message and code to the buffer.

Parameters:

code An integer code.

message The message to add to the report.

The documentation for this class was generated from the following file:

- StatusReport.hpp

3.8 TCPClient Class Reference

This is the client part of the TCPServer/TCPClient pair.

`#include <TCPClientServer.hpp>`

Public Member Functions

- TCPClient ()
- TCPClient (const char *serverIp, int port, struct timeval &timeout, int bdp=0)
- ~TCPClient ()
- int init (const char *serverIp, int port, struct timeval &timeout, int bdp=0)
- int sendAndReceive (char *outMsgBuf, int outMsgLen, char *inMsgBuf, int inBufLen, int *inMsgLen)
- int getStatusCode () const
- const char * getStatusMessage () const
- int enableIgnoreSigPipe ()
- int disableIgnoreSigPipe ()

3.8.1 Detailed Description

This is the client part of the TCPServer/TCPClient pair.

An object of this class can establish connection with an object of class (or derived from) TCPServer over a TCP/IP network. This implementation does not do endian conversions to the data being sent/received over the network. Hence you will have jumbled data when communicating between little endian and big endian devices and vice-versa (no problems if both ends use same byte order for data). Note that this implementation provides a signal handler to ignore SIGPIPE. This will allow client to keep running even after send/recv data on illegal socket resulting from an unexpected server termination.

Use TCPClient/TCPServer when you want to reliably transfer data at slow speeds. Use UDPClient/UDPServer when your primary requirement is speed.

Example Program: See example for TCPServer

3.8.2 Constructor & Destructor Documentation

3.8.2.1 TCPClient::TCPClient ()

The default constructor. Does nothing.

3.8.2.2 TCPClient::TCPClient (const char * *serverIp*, int *port*, struct timeval & *timeout*, int *bdp* = 0)

This constructor initializes parameters for a connection to remote server BUT doesn't connect until sendAndReceive() is called.

Parameters:

serverIp IP name of the remote server.

port Port address on which the remote server is listening for client connections.

timeout The sendAndReceive() function sends messages and waits for replies from the server. This parameter sets the timeout period in waiting for a reply. If a reply is not received within this timeout period, sendAndReceive() will exit with error.

bdp This is an advanced option. It allows the user to suggest the bandwidth-delay product in kilo bytes so that socket buffers of optimal sizes can be created. Suppose you are going to connect to a machine whose round-trip time (delay between sending a packet and receiving acknowledgement) is 50ms, and the link bandwidth is 100 Mbits per sec. Then your BDP is 100e6 * 50e-3 / 8 = 625 kilo bytes. You can use the 'ping' utility to get an approx. measure for the round-trip time. Set this to 0 to use system defaults.

3.8.2.3 TCPClient::~TCPClient ()

The destructor. Cleans up.

3.8.3 Member Function Documentation

3.8.3.1 int TCPClient::disableIgnoreSigPipe ()

Call this function to disable SIG_PIPE handling. The client will terminate if server terminates

Returns:
0 if no error, else -1

3.8.3.2 int TCPClient::enableIgnoreSigPipe ()

Call this function to ignore SIG_PIPE, and hence save client from terminating due to server termination

Returns:
0 if no error, else -1

3.8.3.3 int TCPClient::getStatusCode () const

Returns:
Latest status code.

3.8.3.4 const char* TCPClient::getStatusMessage () const

Returns:
Latest error status report.

3.8.3.5 int TCPClient::init (const char * *serverIp*, int *port*, struct timeval & *timeout*, int *bdp* = 0)

Establish connection with a remote server.

Parameters:

serverIp IP name of the remote server.

port Port address on which the remote server is listening for client connections.

timeout The sendAndReceive() function sends messages and waits for replies from the server. This parameter sets the timeout period in waiting for a reply. If a reply is not received within this timeout period, sendAndReceive() will exit with error.

bdp This is an advanced option. It allows the user to suggest the bandwidth-delay product in kilo bytes so that socket buffers of optimal sizes can be created. Suppose you are going to connect to a machine whose round-trip time (delay between sending a packet and receiving acknowledgement) is 50ms, and the link bandwidth is 100 Mbits per sec. Then your BDP is 100e6 * 50e-3 / 8 = 625 kilo bytes. You can use the 'ping' utility to get an approx. measure for the round-trip time. Set this to 0 to use system defaults.

Returns:

0 on success, -1 on error.

3.8.3.6 int TCPClient::sendAndReceive (char * *outMsgBuf*, int *outMsgLen*, char * *inMsgBuf*, int *inBufLen*, int * *inMsgLen*)

Send a message to the server, and receive a reply.

- This function will normally block waiting for reply from server unless you set *inMsgBuf* to NULL.
- The message from server is discarded if the receive buffer *inMsgBuf* is not large enough.

Parameters:

outMsgBuf Pointer to buffer containing your message to server.

outMsgLen The length of your message above.

inMsgBuf A pointer to buffer provided by you to store the reply message from the server. Set this to NULL if you aren't interested in reply from server.

inBufLen The size (bytes) of the above buffer.

inMsgLen The actual length (bytes) of message received from the server.

Returns:

0 on success, -1 on error. Call getStatus....() for the error.

The documentation for this class was generated from the following file:

- TCPClientServer.hpp

3.9 TCPServer Class Reference

This is the server part of the TCPServer/TCPClient pair.

`#include <TCPClientServer.hpp>`

Public Member Functions

- TCPServer ()
- TCPServer (int port, int maxMsgSize=1024, int bdp=0)
- virtual ~TCPServer ()
- int init (int port, int maxMsgSize, int bdp=0)
- void doMessageCycle ()
- int getStatusCode () const
- const char * getStatusMessage () const
- int enableIgnoreSigPipe ()
- int disableIgnoreSigPipe ()

Protected Member Functions

- virtual const char * receiveAndReply (const char *inMsgBuf, int inMsgLen, int *outMsgLen)

3.9.1 Detailed Description

This is the server part of the TCPServer/TCPClient pair.

This implementation is the base class that provides server functionality in a client-server relationship over a TCP/IP network. The user must reimplement atleast the receiveAndReply() function in a derived class in order to have a functional server. This server can listen for upto 20 waiting client connections. This implementation does not do endian conversions to the data being sent/received over the network. Hence you will have jumbled data when communicating between little endian and big endian devices and vice-versa (no problems if both ends use same byte order for data). Note also that this implementation provides a signal handler to ignore SIGPIPE. This will allow server to keep running even after send/recv data on illegal socket resulting from an unexpected client termination.

Use TCPClient/TCPServer when you want to reliably transfer data at slow speeds. Use UDPClient/UDPServer when your primary requirement is speed.

Example Program:

```
//============================================================================
// TCPClientServer.t.cpp - Example program for TCPClient/TCPServer
//
// Author        : Vilas Kumar Chitrakaran
//============================================================================

#include "TCPClientServer.hpp"
#include <iostream>
#include <string>
#include <pthread.h>

using namespace std;

//============================================================================
// class MyServer
```

```
//============================================================================
class MyServer : public TCPServer
{
 public:
  MyServer(int port, int maxLen, int bdp) : TCPServer(port, maxLen, bdp){};
  ~MyServer() {};
  virtual const char *receiveAndReply(const char *inMsgBuf, int inMsgLen, int *outMsgLen);
 private:
  char d_outMsgBuf[80];
};


const char *MyServer::receiveAndReply(const char *inMsgBuf, int inMsgLen, int *outMsgLen)
{
 cout << "MyServer: client said: ";
 for(int i = 0; i < inMsgLen; i++) cout << inMsgBuf[i];
 cout << endl;
 snprintf(d_outMsgBuf, 80, "%s", "Hi there client!!");
 *outMsgLen = strlen(d_outMsgBuf);
 return d_outMsgBuf;
}

//============================================================================
// server
// - handles messages less than 8 bytes long from clients
//============================================================================
void *server(void *arg)
{
 arg=arg;
 MyServer server(3000, 8, 100);

 server.enableIgnoreSigPipe();

 // check for errors
 if(server.getStatusCode())
  cout << "server: " << server.getStatusMessage() << endl;

 // serve clients
 server.doMessageCycle();

 // check for error
 if(server.getStatusCode())
  cout << "server: " << server.getStatusMessage() << endl;

 return NULL;
}


//============================================================================
// client
// - alternately sends valid and invalid messages to server
//============================================================================
int client()
{
 char outMsgBuf[80];
 char inMsgBuf[80];
 int outMsgLen = 0;
 int inMsgLen;
 bool flip = false;

 struct timeval timeout;
 timeout.tv_sec = 0;
 timeout.tv_usec = 5000; // 5 ms

 // initialize a client and connect to server
 TCPClient client("127.0.0.1", 3000, timeout, 100);
 if(client.getStatusCode())
```

```
 {
  cout << "client: " << client.getStatusMessage() << endl;
  return -1;
 }

 client.enableIgnoreSigPipe();
 int msgNum = 0;
 while(1)
 {
  if(msgNum > 100)
   return 0;

  // a message
  if(flip)
  {
   // improper message (longer than what server can handle)
   snprintf(outMsgBuf, 80, "%s %d", "Hello server", msgNum);
   outMsgLen = strlen(outMsgBuf);
   flip = !flip;
   cout << endl << "client  : Sending invalid (long) client msg. - " << outMsgBuf << endl;
  }
  else
  {
   // proper message
   snprintf(outMsgBuf, 80, "%s %d", "Hello", msgNum);
   outMsgLen = strlen(outMsgBuf);
   //flip = !flip;
   cout << endl << "client  : Sending valid client msg. - " << outMsgBuf << endl;
  }
  msgNum++;

  // Send a message and receive reply
  if( client.sendAndReceive(outMsgBuf, outMsgLen, inMsgBuf, 80, &inMsgLen) == -1)
  {
   cout << "client  : " << client.getStatusMessage() << endl;
   continue;
  }

  // reply received
  cout << "client  : server replied: ";
  for(int i = 0; i < inMsgLen; i++) cout << inMsgBuf[i];
  cout << endl;
 }

 // bye
 if(client.getStatusCode())
  cout << "client  : " << client.getStatusMessage() << endl;

 return 0;
}


//========================================================================
// main function
//========================================================================
int main()
{
 pthread_t threadId;
 pthread_create(&threadId, NULL, &server, NULL);
 sleep(1);
 client();
 return 0;
}
```

3.9.2 Constructor & Destructor Documentation

3.9.2.1 TCPServer::TCPServer ()

The default constructor. Does nothing.

3.9.2.2 TCPServer::TCPServer (int *port*, int *maxMsgSize* = 1024, int *bdp* = 0)

Initializes the sever.

Parameters:

port The port on which the server will wait for clients.

maxMsgSize Maximum size (bytes) of the receive buffer. Client messages larger than this size are discarded.

bdp This is an advanced option. It allows the user to suggest the bandwidth-delay product in kilo bytes so that socket buffers of optimal sizes can be created. Suppose you are going to receive connections from a machine whose round-trip time (delay between sending a packet and receiving acknowledgement) is 50ms, and the link bandwidth is 100 Mbits per sec. Then your BDP is 100e6 * 50e-3 / 8 = 625 kilo bytes. You can use the 'ping' utility to get an approx. measure for the round-trip time. Set this to 0 to use system defaults.

3.9.2.3 virtual TCPServer::~TCPServer () [virtual]

The destructor frees resources.

3.9.3 Member Function Documentation

3.9.3.1 int TCPServer::disableIgnoreSigPipe ()

Call this function to disable SIG_PIPE handling. The server will terminate if client terminates

Returns:

0 if no error, else -1

3.9.3.2 void TCPServer::doMessageCycle ()

This function never returns, unless server initialization failed. It constantly checks for any waiting clients. When connected to a client it copies the message from the client into the message buffer and calls receiveAndReply(). Upon return from user implemented receiveAndReply() this function will reply back to the client if required (see receiveAndReply()).

3.9.3.3 int TCPServer::enableIgnoreSigPipe ()

Call this function to ignore SIG_PIPE, and hence save server from terminating due to client termination

Returns:

0 if no error, else -1

3.9.3.4 int TCPServer::getStatusCode () const

Returns:
0 on no error, else latest status code. See errno.h for codes.

3.9.3.5 const char* TCPServer::getStatusMessage () const

Returns:
Latest error status report

3.9.3.6 int TCPServer::init (int *port*, int *maxMsgSize*, int *bdp* = 0)

Initialize the server.

Parameters:
port The port number used by the server in listening for clients.

maxMsgSize Maximum size (bytes) of the receive buffer. Client messages larger than this size are discarded.

bdp Estimated BDP. See constructor for details.

Returns:
0 on success, -1 on failure.

3.9.3.7 virtual const char* TCPServer::receiveAndReply (const char * *inMsgBuf*, int *inMsgLen*, int * *outMsgLen*) [protected, virtual]

Re-implement this function in your derived class. This function is called by doMessageCycle() everytime it receives a message from a client.

- If return value is set to NULL, the server will not attempt to reply back to the client.
- The message from client is discarded if the receive buffer size (set in the constructor) is not large enough.

Parameters:
inMsgBuf Pointer to buffer containing message from client.

inMsgLen Length of the message (bytes) in the above buffer.

outMsgLen The length (bytes) of the reply buffer.

Returns:
NULL, or a pointer to reply buffer provided by you containing reply message for the client.

The documentation for this class was generated from the following file:

- TCPClientServer.hpp

3.10 Thread Class Reference

A mechanism to execute code in a separate thread.

`#include <Thread.hpp>`

Public Member Functions

- Thread ()
- virtual ~Thread ()
- int run (void *arg=NULL)
- bool isThreadRunning ()
- int cancel ()
- int join ()
- pthread_t getThreadId ()

Protected Member Functions

- virtual void enterThread (void *arg)=0
- virtual int executeInThread (void *arg)=0
- virtual void exitThread (void *arg)=0

3.10.1 Detailed Description

A mechanism to execute code in a separate thread.

This is a pure virtual base class for threads. Users must reimplement atleast the enterThread(), executeInThread() and exitThread() functions in a derived class to use this wrapper over POSIX threads.

This class is based partly on the idea presented by Ryan Teixeira in http://www.geocities.com/SiliconValley/Heights/6038/dthreads.html

Example Program:

```
//============================================================================
// Thread.t.cpp - Example program for Thread class.
//
// Author        : Vilas Kumar Chitrakaran
//============================================================================

#include "Thread.hpp"
#include "RWLock.hpp"
#include <iostream>
#include <string>
#include <time.h>

using namespace std;

//============================================================================
// MyThread class
//============================================================================
class MyThread : public Thread
{
 public:
  MyThread();
  ~MyThread();
```

```
  void setVal(int p);
  int getVal();
 protected:
  virtual void enterThread(void *arg);
  virtual int executeInThread(void *arg);
  virtual void exitThread(void *arg);
 private:
  int d_val;
  RWLock d_lock;
};

//---------------------------------------------------------------------------
// MyThread::MyThread
//---------------------------------------------------------------------------
MyThread::MyThread()
{
 d_val = 0;
}

//---------------------------------------------------------------------------
// MyThread::~MyThread
//---------------------------------------------------------------------------
MyThread::~MyThread()
{
 cancel();
}

//---------------------------------------------------------------------------
// MyThread::enterThread
//---------------------------------------------------------------------------
void MyThread::enterThread(void *arg)
{
 arg=arg;
 cout << "thread: in entry routine" << endl;
}

//---------------------------------------------------------------------------
// MyThread::executeInThread
//---------------------------------------------------------------------------
int MyThread::executeInThread(void *arg)
{
 int i = 0;
 struct timespec napTime;
 napTime.tv_sec = 0;
 napTime.tv_nsec = (long int)5e8;

 while(1)//i < 2)
 {
  i = ((MyThread *)arg)->getVal();
  cout << "thread: reading value = " << i << endl;
  nanosleep(&napTime, NULL);
  pthread_testcancel();
 }

 return i;
}

//---------------------------------------------------------------------------
// MyThread::setVal
//---------------------------------------------------------------------------
void MyThread::setVal(int p)
{
 d_lock.writeLock();
 d_val = p;
 d_lock.unlock();
}
```

```
//-----------------------------------------------------------------------------
// MyThread::getVal
//-----------------------------------------------------------------------------
int MyThread::getVal()
{
 int i;
 d_lock.readLock();
 i = d_val;
 d_lock.unlock();
 return i;
};

//-----------------------------------------------------------------------------
// MyThread::cleanupInThread
//-----------------------------------------------------------------------------
void MyThread::exitThread(void *arg)
{
 arg=arg;
 cout << "thread: in cleanup routine" << endl;
}


//=============================================================================
// main function
//=============================================================================
int main()
{
 MyThread thread;
 struct timespec napTime;
 napTime.tv_sec = 1;
 napTime.tv_nsec = 0;
 int err;
 int i = 0;

 // Start the thread
 err = thread.run(&thread);
 if( err != 0)
 {
  cout << "parent: Thread creation failed: " << strerror(err) << endl;
  return -1;
 }

 // Yield to allow the thread to run
 sched_yield();

 // Change data in parent thread
 while(i < 3)
 {
  cout << "parent: setting value = " << i << endl;
  thread.setVal(i);
  nanosleep(&napTime,NULL);
  i++;
 }

 // wait for thread to finish
 //err = thread.join();
 if( err != 0)
 {
  cout << "parent: thread retuned: " << err << endl;
 }
 cout << "parent: exiting" << endl;
 return 0;
}
```

3.10.2 Constructor & Destructor Documentation

3.10.2.1 Thread::Thread ()

The constructor. Does some initializations.

3.10.2.2 virtual Thread::~Thread () [virtual]

This destructor does nothing. NOTE: If the derived class does not have a shutdown routine that calls cancel() and join() to wait for the thread to exit cleanly after executing the clean up routine (see exitThread()), the destructor of the derived class should call cancel() and join().

3.10.3 Member Function Documentation

3.10.3.1 int Thread::cancel ()

Request cancellation of execution of the thread. Cancel requests are held pending until a cancellation point in the thread is reached (see man pages for pthread_setcanceltype() and pthread_testcancel()).

Returns:
0 on success, and errno code on error (ESRCH if thread is already cancelled).

3.10.3.2 virtual void Thread::enterThread (void * *arg*) [protected, pure virtual]

As soon as the run() function is called and the thread is instantiated, this function gets called in the separate thread. Override this function in the derived class to do thread setup operations such as setting thread priority, scheduling policy and so on.

Parameters:
arg Arguments passsed by the call to run().

3.10.3.3 virtual int Thread::executeInThread (void * *arg*) [protected, pure virtual]

This is the main function that gets executed in a separate thread. Override this function in your derived class.

Parameters:
arg Arguments passsed by the call to run().

Returns:
Your choice of return value.

3.10.3.4 virtual void Thread::exitThread (void * *arg*) [protected, pure virtual]

This is a routine that gets executed in the thread just before it terminates (due to a call to cancel(), pthread_exit() and so on). Put your cleanup code here by overriding this function in your derived class.

Parameters:
arg Arguments passsed by the call to run().

3.10.3.5 pthread_t Thread::getThreadId ()

Returns:
Thread ID if the thread is already running, else 0.

3.10.3.6 bool Thread::isThreadRunning ()

Returns:
true if thread is running, else false.

3.10.3.7 int Thread::join ()

Wait until thread finishes execution.

Returns:
return value from the thread, or -1 if thread already exited (most probably due to a call to cancel).

3.10.3.8 int Thread::run (void * *arg* = NULL)

Start the thread with arguments 'arg'.

- This function blocks until enterThread() finishes executing in the thread.

Parameters:
arg A pointer to arguments passed to the new thread.

Returns:
0 on success, and errno code on error (EPERM if thread is already running).

The documentation for this class was generated from the following file:

- Thread.hpp

3.11 UDPClient Class Reference

This is the client part of the UDPServer/UDPClient pair.

`#include <UDPClientServer.hpp>`

Public Member Functions

- UDPClient ()
- UDPClient (const char *serverIp, int port, struct timeval &timeout, int bdp=0)
- ~UDPClient ()
- int init (const char *serverIp, int port, struct timeval &timeout, int bdp=0)
- int getStatusCode () const
- const char * getStatusMessage () const
- int sendAndReceive (char *outMsgBuf, int outMsgLen, char *inMsgBuf, int inBufLen, int *inMsgLen)

3.11.1 Detailed Description

This is the client part of the UDPServer/UDPClient pair.

UDPServer/UDPClient uses the User Datagram Protocol (UDP, IETF RFC768) for fast, 'unreliable' data transfer between two devices over the ethernet. The protocol is unreliable because there is no guarantee that data packets will reach their destination, or that they will reach the destination in the right sequence. UDP prioritizes speed over reliability. Use (the much slower) TCPServer/TCPClient if reliability and data integrity is more important in your application.

Example Program: See example for UDPServer

3.11.2 Constructor & Destructor Documentation

3.11.2.1 UDPClient::UDPClient ()

The default constructor. Does nothing.

3.11.2.2 UDPClient::UDPClient (const char * *serverIp*, int *port*, struct timeval & *timeout*, int *bdp* = 0)

This constructor initializes parameters for a connection to remote server.

Parameters:

serverIp IP name of the remote server.

port Port address on which the remote server is listening for client connections.

timeout The sendAndReceive() function sends messages and waits for replies from the server. This parameter sets the timeout period in waiting for a reply. If a reply is not received within this timeout period, sendAndReceive() will exit with error.

bdp This is an advanced option. It allows the user to suggest the bandwidth-delay product in kilo bytes so that socket buffers of optimal sizes can be created. Suppose you are going to connect to a machine whose round-trip time (delay between sending a packet and receiving acknowledgement) is 50ms, and the link bandwidth is 100 Mbits per sec. Then your BDP is 100e6 * 50e-3 / 8 = 625 kilo bytes. You can use the 'ping' utility

to get an approx. measure for the round-trip time. Set this value to 0 to use system defaults.

3.11.2.3 UDPClient::~UDPClient ()

The destructor. Cleans up.

3.11.3 Member Function Documentation

3.11.3.1 int UDPClient::getStatusCode () const

Returns:
Latest status code.

3.11.3.2 const char* UDPClient::getStatusMessage () const

Returns:
Latest error status report.

3.11.3.3 int UDPClient::init (const char * *serverIp*, int *port*, struct timeval & *timeout*, int *bdp* = 0)

Establish connection with a remote server.

Parameters:

serverIp IP name of the remote server.

port Port address on which the remote server is listening for client connections.

timeout The sendAndReceive() function sends messages and waits for replies from the server. This parameter sets the timeout period in waiting for a reply. If a reply is not received within this timeout period, sendAndReceive() will exit with error.

bdp This is an advanced option. It allows the user to suggest the bandwidth-delay product in kilo bytes so that socket buffers of optimal sizes can be created. Suppose you are going to connect to a machine whose round-trip time (delay between sending a packet and receiving acknowledgement) is 50ms, and the link bandwidth is 100 Mbits per sec. Then your BDP is 100e6 * 50e-3 / 8 = 625 kilo bytes. You can use the 'ping' utility to get an approx. measure for the round-trip time. Set this value to 0 to use system defaults.

Returns:
0 on success, -1 on error.

3.11.3.4 int UDPClient::sendAndReceive (char * *outMsgBuf*, int *outMsgLen*, char * *inMsgBuf*, int *inBufLen*, int * *inMsgLen*)

Send a message to the server, and receive a reply.

- If *inMsgBuf* is set to NULL, the function will return immediately after sending a message. It won't wait for any reply.

- If *inMsgBuf* is set, this function will block until a reply is received or until timeout (set in init()).
- If the data packet is too long to fit into the receive buffer *inMsgBuf*, the excess message from client is discarded.

Parameters:

outMsgBuf Pointer to buffer containing your message to server.

outMsgLen The length of your message above.

inMsgBuf A pointer to buffer provided by you to store the reply message from the server. Set this to NULL if you aren't interested in reply from server.

inBufLen The size (bytes) of the above buffer.

inMsgLen The actual length (bytes) of message received from the server.

Returns:

0 on success, -1 on error. Call getStatus....() for the error.

The documentation for this class was generated from the following file:

- UDPClientServer.hpp

3.12 UDPServer Class Reference

This is the server part of the UDPServer/UDPClient pair.

`#include <UDPClientServer.hpp>`

Public Member Functions

- UDPServer ()
- UDPServer (int port, int maxMsgSize, int bdp=0)
- virtual ~UDPServer ()
- int init (int port, int maxMsgSize, int bdp=0)
- void doMessageCycle ()
- int getStatusCode () const
- const char * getStatusMessage () const

Protected Member Functions

- virtual const char * receiveAndReply (const char *inMsgBuf, int inMsgLen, int *outMsgLen)

3.12.1 Detailed Description

This is the server part of the UDPServer/UDPClient pair.

UDPServer/UDPClient uses the User Datagram Protocol (UDP, IETF RFC768) for fast, 'unreliable' data transfer between two devices over the ethernet. The protocol is unreliable because there is no guarantee that data packets will reach their destination, or that they will reach the destination in the right sequence. UDP prioritizes speed over reliability. Use (the much slower) TCPServer/TCPClient if reliability and data integrity is more important in your application.

Example Program:

```
//============================================================================
// UDPClientServer.t.cpp - Example program for UDPClient/UDPServer
//
// Author        : Vilas Kumar Chitrakaran
//============================================================================

#include "UDPClientServer.hpp"
#include <iostream>
#include <string>
#include <pthread.h>

using namespace std;

//============================================================================
// class MyServer
//============================================================================
class MyServer : public UDPServer
{
 public:
  MyServer(int port, int maxLen, int bdp) : UDPServer(port, maxLen, bdp){};
  ~MyServer() {};
 protected:
  virtual const char *receiveAndReply(const char *inMsgBuf, int inMsgLen, int *outMsgLen);
 private:
  char d_outMsgBuf[80];
```

```
};


const char *MyServer::receiveAndReply(const char *inMsgBuf, int inMsgLen, int *outMsgLen)
{
 cout << "MyServer: client said: ";
 for(int i = 0; i < inMsgLen; i++) cout << inMsgBuf[i];
 cout << endl;
 snprintf(d_outMsgBuf, 80, "%s", "Hi there client!!");
 *outMsgLen = strlen(d_outMsgBuf);
 return d_outMsgBuf;
}

//============================================================================
// server
// - handles messages from clients
//============================================================================
void *server(void *arg)
{
 arg=arg;
 MyServer server(3000, 80, 625);

 // check for errors
 if(server.getStatusCode())
  cout << "server: " << server.getStatusMessage() << endl;

 // serve clients
 server.doMessageCycle();

 // check for error
 if(server.getStatusCode())
  cout << "server: " << server.getStatusMessage() << endl;

 return NULL;
}


//============================================================================
// client
//============================================================================
int client()
{
 char outMsgBuf[80];
 char inMsgBuf[80];
 int outMsgLen = 0;
 int inMsgLen;

 struct timeval timeout;
 timeout.tv_sec = 0;
 timeout.tv_usec = 50000; // 50 ms

 // initialize a client and connect to server
 UDPClient client("127.0.0.1", 3000, timeout, 625);
 if(client.getStatusCode())
 {
  cout << "client: " << client.getStatusMessage() << endl;
  return -1;
 }

 int msgNum = 0;
 while(1)
 {
  if(msgNum > 100)
   break;

  // a message
  snprintf(outMsgBuf, 80, "%s %d", "Hello server", msgNum);
```

```
  outMsgLen = strlen(outMsgBuf);
  cout << endl << "client  : Sending msg. - " << outMsgBuf << endl;
  msgNum++;

  // Send the message and receive reply
  if( client.sendAndReceive(outMsgBuf, outMsgLen, inMsgBuf, 80, &inMsgLen) == -1)
  {
   cout << "client  : " << client.getStatusMessage() << endl;
   break;
  }

  // reply received
  cout << "client  : server replied: ";
  for(int i = 0; i < inMsgLen; i++) cout << inMsgBuf[i];
  cout << endl;
 }

 // bye
 if(client.getStatusCode())
  cout << "client  : " << client.getStatusMessage() << endl;

 return 0;
}


//========================================================================
// main function
//========================================================================
int main()
{
 pthread_t threadId;
 pthread_create(&threadId, NULL, &server, NULL);
 sleep(1);
 client();
 return 0;
}
```

3.12.2 Constructor & Destructor Documentation

3.12.2.1 UDPServer::UDPServer ()

The default constructor. Does nothing.

3.12.2.2 UDPServer::UDPServer (int *port*, int *maxMsgSize*, int *bdp* = 0)

Initializes the sever.

Parameters:

port The port on which the server will listen for data packets from clients.

maxMsgSize Maximum size (bytes) of the receive buffer. Client messages larger than this size are discarded.

bdp This is an advanced option. It allows the user to suggest the bandwidth-delay product in kilo bytes so that socket buffers of optimal sizes can be created. Suppose you are going to receive connections from a machine whose round-trip time (delay between sending a packet and receiving acknowledgement) is 50ms, and the link bandwidth is 100 Mbits per sec. Then your BDP is 100e6 * 50e-3 / 8 = 625 kilo bytes. You can use the 'ping' utility to get an approx. measure for the round-trip time. Set this value to 0 to use system defaults.

3.12.2.3 virtual UDPServer::~UDPServer () [virtual]

The destructor frees resources.

3.12.3 Member Function Documentation

3.12.3.1 void UDPServer::doMessageCycle ()

This function never returns, waiting for any data packets in an infinite loop. When a data packet is received, it copies the data into the message buffer and calls the user implemented function receiveAndReply().

3.12.3.2 int UDPServer::getStatusCode () const

Returns:
0 on no error, else latest status code. See errno.h for codes.

3.12.3.3 const char* UDPServer::getStatusMessage () const

Returns:
Latest error status report

3.12.3.4 int UDPServer::init (int *port*, int *maxMsgSize*, int *bdp* = 0)

Initialize the server.

Parameters:

port The port number used by the server in listening for clients.

maxMsgSize Maximum size (bytes) of the receive buffer. Client messages larger than this size are discarded.

bdp This is an advanced option. It allows the user to suggest the bandwidth-delay product in kilo bytes so that socket buffers of optimal sizes can be created. Suppose you are going to receive connections from a machine whose round-trip time (delay between sending a packet and receiving acknowledgement) is 50ms, and the link bandwidth is 100 Mbits per sec. Then your BDP is 100e6 * 50e-3 / 8 = 625 kilo bytes. You can use the 'ping' utility to get an approx. measure for the round-trip time. Set this value to 0 to use system defaults.

Returns:
0 on success, -1 on failure.

3.12.3.5 virtual const char* UDPServer::receiveAndReply (const char * *inMsgBuf*, int *inMsgLen*, int * *outMsgLen*) [protected, virtual]

Re-implement this function in your derived class. This function is called by doMessageCycle() everytime it receives a data packet from a client.

- If return value is set to NULL, the server will not attempt to reply back to the client.

- If the data packet is too long to fit into the receive buffer (whose length is set in the constructor), the excess message from client is discarded.

Parameters:

inMsgBuf Pointer to buffer containing message from client.

inMsgLen Length of the message (bytes) in the above buffer.

outMsgLen The length (bytes) of the reply buffer.

Returns:

NULL, or a pointer to reply buffer provided by you containing reply message for the client.

The documentation for this class was generated from the following file:

- UDPClientServer.hpp

Index