NAME

ost::Thread −

Every thread of execution in an application is created by instantiating an object of a class derived from the **Thread** class.

SYNOPSIS

```
#include <thread.h>
```

Inherited by **ost::PosixThread**, **ost::SerialService**, **ost::SocketService**, **ost::TCPSession**, **ost::ThreadQueue**, **ost::TTYSession**, and **ost::UnixSession**.

Public Types

enum **Throw** { **throwNothing**, **throwObject**, **throwException** }
How to raise error.

enum **Cancel** { **cancelInitial** = 0, **cancelDeferred** = 1, **cancelImmediate**, **cancelDisabled**, **cancelManual**, **cancelDefault** = cancelDeferred }
How work cancellation.

enum **Suspend** { **suspendEnable**, **suspendDisable** }
How work suspend.

typedef enum **ost::Thread::Throw Throw**
How to raise error.

typedef enum **ost::Thread::Cancel Cancel**
How work cancellation.

typedef enum **ost::Thread::Suspend Suspend**
How work suspend.

Public Member Functions

Thread (bool isMain)
This is actually a special constructor that is used to create a thread 'object' for the current execution context when that context is not created via an instance of a derived **Thread** *object itself.*

Thread (int pri=0, size_t stack=0)
When a thread object is contructed, a new thread of execution context is created.

Thread (const **Thread** &th)
A thread of execution can also be specified by cloning an existing thread.

virtual **~Thread** ()
The thread destructor should clear up any resources that have been allocated by the thread.

int **start** (**Semaphore** *start=0)
When a new thread is created, it does not begin immediate execution.

int **detach** (**Semaphore** *start=0)
Start a new thread as 'detached'.

Thread * **getParent** (void)
Gets the pointer to the **Thread** *class which created the current thread object.*

void **suspend** (void)
Suspends execution of the selected thread.

void **resume** (void)
Resumes execution of the selected thread.

Cancel getCancel (void)
Used to retrieve the cancellation mode in effect for the selected thread.

bool **isRunning** (void) const
Verifies if the thread is still running or has already been terminated but not yet deleted.

bool **isDetached** (void) const
Check if this thread is detached.

void **join** (void)
Blocking call which unlocks when thread terminates.

bool **isThread** (void) const
Tests to see if the current execution context is the same as the specified thread object.

cctid_t getId (void) const
Get system thread numeric identifier.

const char * **getName** (void) const

Get the name string for this thread, to use in debug messages.

Static Public Member Functions

static **Thread** * **get** (void)

static void **setStack** (size_t size=0)

Set base stack limit before manual stack sizes have effect.

static void **sleep** (**timeout_t** msec)

A thread-safe sleep call.

static void **yield** (void)

Yields the current thread's CPU time slice to allow another thread to begin immediate execution.

static **Throw getException** (void)

Get exception mode of the current thread.

static void **setException** (**Throw** mode)

Set exception mode of the current thread.

static **Cancel enterCancel** (void)

This is used to help build wrapper functions in libraries around system calls that should behave as cancellation points but don't.

static void **exitCancel** (**Cancel** cancel)

This is used to restore a cancel block.

Protected Member Functions

void **setName** (const char *text)

Set the name of the current thread.

virtual void **run** (void)=0

All threads execute by deriving the Run method of **Thread**.

virtual void **final** (void)

A thread that is self terminating, either by invoking **exit()** *or leaving it's* **run()**, *will have this method called.*

virtual void **initial** (void)

The initial method is called by a newly created thread when it starts execution.

virtual void * **getExtended** (void)

Since **getParent()** *and* **getThread()** *only refer to an object of the* **Thread** *'base' type, this virtual method can be replaced in a derived class with something that returns data specific to the derived class that can still be accessed through the pointer returned by* **getParent()** *and* **getThread()**.

virtual void **notify** (**Thread** *)

When a thread terminates, it now sends a notification message to the parent thread which created it.

void **exit** (void)

Used to properly exit from a **Thread** *derived* **run()** *or* **initial()** *method.*

void **sync** (void)

Used to wait for a join or cancel, in place of explicit exit.

bool **testCancel** (void)

test a cancellation point for deferred thread cancellation.

void **setCancel** (**Cancel** mode)

Sets thread cancellation mode.

void **setSuspend** (**Suspend** mode)

Sets the thread's ability to be suspended from execution.

void **terminate** (void)

Used by another thread to terminate the current thread.

void **clrParent** (void)

clear parent thread relationship.

Friends

class **PosixThread**

class **DummyThread**

class **Cancellation**

class **postream_type**

class **Slog**

class **ThreadImpl**

void **operator++** (**Thread** &th)

Signal the semaphore that the specified thread is waiting for before beginning execution.

void **operator--** (**Thread** &th)

Detailed Description

Every thread of execution in an application is created by instantiating an object of a class derived from the **Thread** class.

Classes derived from **Thread** must implement the **run()** method, which specifies the code of the thread. The base **Thread** class supports encapsulation of the generic threading methods implemented on various target operating systems. This includes the ability to start and stop threads in a synchronized and controllable manner, the ability to specify thread execution priority, and thread specific 'system call' wrappers, such as for sleep and yield. A thread exception is thrown if the thread cannot be created. Threading was the first part of Common C++ I wrote, back when it was still the APE library. My goal for Common C++ threading has been to make threading as natural and easy to use in C++ application development as threading is in Java. With this said, one does not need to use threading at all to take advantage of Common C++. However, all Common C++ classes are designed at least to be thread-aware/thread-safe as appropriate and necessary.

Common C++ threading is currently built either from the Posix 'pthread' library or using the win32 SDK. In that the Posix 'pthread' draft has gone through many revisions, and many system implementations are only marginally compliant, and even then usually in different ways, I wrote a large series of autoconf macros found in ost_pthread.m4 which handle the task of identifying which pthread features and capabilities your target platform supports. In the process I learned much about what autoconf can and cannot do for you..

Currently the GNU Portable **Thread** library (GNU pth) is not directly supported in Common C++. While GNU 'Pth' doesn't offer direct native threading support or benefit from SMP hardware, many of the design advantages of threading can be gained from it's use, and the Pth pthread 'emulation' library should be usable with Common C++. In the future, Common C++ will directly support Pth, as well as OS/2 and BeOS native threading API's.

Common C++ itself defines a fairly 'neutral' threading model that is not tied to any specific API such as pthread, win32, etc. This neutral thread model is contained in a series of classes which handle threading and synchronization and which may be used together to build reliable threaded applications.

Common C++ defines application specific threads as objects which are derived from the Common C++ 'Thread' base class. At minimum the 'Run' method must be implemented, and this method essentially is the 'thread', for it is executed within the execution context of the thread, and when the Run method terminates the thread is assumed to have terminated.

Common C++ allows one to specify the running priority of a newly created thread relative to the 'parent' thread which is the thread that is executing when the constructor is called. Since most newer C++ implementations do not allow one to call virtual constructors or virtual methods from constructors, the thread must be 'started' after the constructor returns. This is done either by defining a 'starting' semaphore object that one or more newly created thread objects can wait upon, or by invoking an explicit 'start' member function.

Threads can be 'suspended' and 'resumed'. As this behavior is not defined in the Posix 'pthread' specification, it is often emulated through signals. Typically SIGUSR1 will be used for this purpose in Common C++ applications, depending in the target platform. On Linux, since threads are indeed processes, SIGSTP and SIGCONT can be used. On solaris, the Solaris thread library supports suspend and resume directly.

Threads can be canceled. Not all platforms support the concept of externally cancelable threads. On those platforms and API implementations that do not, threads are typically canceled through the action of a signal handler.

As noted earlier, threads are considered running until the 'Run' method returns, or until a cancellation request is made. Common C++ threads can control how they respond to cancellation, using setCancellation(). **Cancellation** requests can be ignored, set to occur only when a cancellation 'point' has been reached in the code, or occur immediately. Threads can also exit by returning from Run() or by invoking the Exit() method.

Generally it is a good practice to initialize any resources the thread may require within the constructor

of your derived thread class, and to purge or restore any allocated resources in the destructor. In most cases, the destructor will be executed after the thread has terminated, and hence will execute within the context of the thread that requested a join rather than in the context of the thread that is being terminated. Most destructors in derived thread classes should first call Terminate() to make sure the thread has stopped running before releasing resources.

A Common C++ thread is normally canceled by deleting the thread object. The process of deletion invokes the thread's destructor, and the destructor will then perform a 'join' against the thread using the Terminate() function. This behavior is not always desirable since the thread may block itself from cancellation and block the current 'delete' operation from completing. One can alternately invoke Terminate() directly before deleting a thread object.

When a given Common C++ thread exits on it's own through it's Run() method, a 'Final' method will be called. This Final method will be called while the thread is 'detached'. If a thread object is constructed through a 'new' operator, it's final method can be used to 'self delete' when done, and allows an independent thread to construct and remove itself autonomously.

A special global function, **getThread()**, is provided to identify the thread object that represents the current execution context you are running under. This is sometimes needed to deliver signals to the correct thread. Since all thread manipulation should be done through the Common C++ (base) thread class itself, this provides the same functionality as things like 'pthread_self' for Common C++.

All Common C++ threads have an exception 'mode' which determines their behavior when an exception is thrown by another Common C++ class. Extensions to Common C++ should respect the current exception mode and use **getException()** to determine what to do when they are about to throw an object. The default exception mode (defined in the **Thread()** constructor) is throwObject, which causes a pointer to an instance of the class where the error occured to be thrown. Other exception modes are throwException, which causes a class-specific exception class to be thrown, and throwNothing, which causes errors to be ignored.

As an example, you could try to call the **Socket** class with an invalid address that the system could not bind to. This would cause an object of type **Socket** * to be thrown by default, as the default exception mode is throwObject. If you call setException(throwException) before the bad call to the **Socket** constructor, an object of type **SockException** (the exception class for class **Socket**) will be thrown instead.

To determine what exception class is thrown by a given Common C++ class when the exception mode is set to throwException, search the source files for the class you are interested in for a class which inherits directly or indirectly from class **Exception**. This is the exception class which would be thrown when the exception mode is set to throwException.

The advantage of using throwException versus throwObject is that more information is available to the programmer from the thrown object. All class-specific exceptions inherit from class **Exception**, which provides a getString() method which can be called to get a human-readable error string.

Common C++ threads are often aggregated into other classes to provide services that are 'managed' from or operate within the context of a thread, even within the Common C++ framework itself. A good example of this is the **TCPSession** class, which essentially is a combination of a TCP client connection and a separate thread the user can define by deriving a class with a Run() method to handle the connected service. This aggregation logically connects the successful allocation of a given resource with the construction of a thread to manage and perform operations for said resource.

Threads are also used in 'service pools'. In Common C++, a service pool is one or more threads that are used to manage a set of resources. While Common C++ does not provide a direct 'pool' class, it does provide a model for their implementation, usually by constructing an array of thread 'service' objects, each of which can then be assigned the next new instance of a given resource in turn or algorithmically.

Threads have signal handlers associated with them. Several signal types are 'predefined' and have special meaning. All signal handlers are defined as virtual member functions of the **Thread** class which are called when a specific signal is received for a given thread. The 'SIGPIPE' event is defined as a 'Disconnect' event since it's normally associated with a socket disconnecting or broken fifo. The Hangup() method is associated with the SIGHUP signal. All other signals are handled through the more generic Signal().

Incidently, unlike Posix, the win32 API has no concept of signals, and certainly no means to define or deliver signals on a per-thread basis. For this reason, no signal handling is supported or emulated in the win32 implementation of Common C++ at this time.

In addition to **TCPStream**, there is a **TCPSession** class which combines a thread with a **TCPStream** object. The assumption made by **TCPSession** is that one will service each TCP connection with a separate thread, and this makes sense for systems where extended connections may be maintained and complex protocols are being used over TCP.

Author:

David Sugar <dyfet AT ostel DOT com> base class used to derive all threads of execution.

Examples:

bug1.cpp, **bug2.cpp**, **tcpservice.cpp**, **tcpstr1.cpp**, **thread1.cpp**, and **thread2.cpp**.

Member Typedef Documentation

typedef enum ost::Thread::Cancel ost::Thread::Cancel

How work cancellation.

typedef enum ost::Thread::Suspend ost::Thread::Suspend

How work suspend.

typedef enum ost::Thread::Throw ost::Thread::Throw

How to raise error.

Member Enumeration Documentation

enum ost::Thread::Cancel

How work cancellation.

Enumerator:

cancelInitial
: used internally, do not use

cancelDeferred
: exit thread on cancellation pointsuch as yield

cancelImmediate
: exit befor cancellation

cancelDisabled
: ignore cancellation

cancelManual
: unimplemented (working in progress)

cancelDefault
: default you should use this for compatibility instead of deferred

enum ost::Thread::Suspend

How work suspend.

Enumerator:

suspendEnable
: suspend enabled

suspendDisable
: suspend disabled, Suspend do nothing

enum ost::Thread::Throw

How to raise error.

Enumerator:

throwNothing
: continue without throwing error

throwObject
: throw object that cause error (throw this)

throwException
throw an object relative to error

Constructor & Destructor Documentation

ost::Thread::Thread (bool isMain)
This is actually a special constructor that is used to create a thread 'object' for the current execution context when that context is not created via an instance of a derived **Thread** object itself. This constructor does not support First.

Parameters:
isMain bool used if the main 'thread' of the application.

ost::Thread::Thread (int pri = `0`, size_t stack = `0`)
When a thread object is contructed, a new thread of execution context is created. This constructor allows basic properties of that context (thread priority, stack space, etc) to be defined. The starting condition is also specified for whether the thread is to wait on a semaphore before begining execution or wait until it's start method is called.

Parameters:
pri thread base priority relative to it's parent.
stack space as needed in some implementations.

ost::Thread::Thread (const Thread & th)
A thread of execution can also be specified by cloning an existing thread. The existing thread's properties (cancel mode, priority, etc), are also duplicated.

Parameters:
th currently executing thread object to clone.

virtual ost::Thread::~Thread () `[virtual]`
The thread destructor should clear up any resources that have been allocated by the thread. The desctructor of a derived thread should begin with Terminate() and is presumed to then execute within the context of the thread causing terminaton.

Member Function Documentation

void ost::Thread::clrParent (void) `[inline, protected]`
clear parent thread relationship.

int ost::Thread::detach (Semaphore * start = `0`)
Start a new thread as 'detached'. This is an alternative **start()** method that resolves some issues with later glibc implimentations which incorrectly impliment self-detach.

Returns:
error code if execution fails.

Parameters:
start optional starting semaphore to alternately use.

Examples:
thread2.cpp.

static Cancel ost::Thread::enterCancel (void) `[static]`
This is used to help build wrapper functions in libraries around system calls that should behave as cancellation points but don't. **Returns:**
saved cancel type.

void ost::Thread::exit (void) `[protected]`
Used to properly exit from a **Thread** derived **run()** or **initial()** method. Terminates execution of the current thread and calls the derived classes **final()** method.

Examples:
bug2.cpp, and **tcpservice.cpp**.

static void ost::Thread::exitCancel (Cancel cancel) `[static]`
This is used to restore a cancel block. **Parameters:**
cancel type that was saved.

virtual void ost::Thread::final (void) `[protected, virtual]`

A thread that is self terminating, either by invoking **exit()** or leaving it's **run()**, will have this method called. It can be used to self delete the current object assuming the object was created with new on the heap rather than stack local, hence one may often see final defined as 'delete this' in a derived thread class. A final method, while running, cannot be terminated or cancelled by another thread. Final is called for all cancellation type (even immediate).

You can safe delete thread ('delete this') class on final, but you should exit ASAP (or do not try to call CommonC++ methods...)

Note:

A thread cannot delete its own context or join itself. To make a thread that is a self running object that self-deletes, one has to detach the thread by using **detach()** instead of **start()**.

See also:

exit

run

Reimplemented in **ost::ThreadQueue**.

Examples:

tcpthread.cpp.

static Thread* ost::Thread::get (void) `[static]`

Referenced by ost::getThread().

Cancel ost::Thread::getCancel (void) `[inline]`

Used to retrieve the cancellation mode in effect for the selected thread. **Returns:**

cancellation mode constant.

static Throw ost::Thread::getException (void) `[static]`

Get exception mode of the current thread. **Returns:**

exception mode.

virtual void* ost::Thread::getExtended (void) `[protected, virtual]`

Since **getParent()** and **getThread()** only refer to an object of the **Thread** 'base' type, this virtual method can be replaced in a derived class with something that returns data specific to the derived class that can still be accessed through the pointer returned by **getParent()** and **getThread()**. **Returns:**

pointer to derived class specific data.

cctid_t ost::Thread::getId (void) const

Get system thread numeric identifier. **Returns:**

numeric identifier of this thread.

const char* ost::Thread::getName (void) const `[inline]`

Get the name string for this thread, to use in debug messages. **Returns:**

debug name.

Thread* ost::Thread::getParent (void) `[inline]`

Gets the pointer to the **Thread** class which created the current thread object. **Returns:**

a **Thread** *, or '(Thread *)this' if no parent.

virtual void ost::Thread::initial (void) `[protected, virtual]`

The initial method is called by a newly created thread when it starts execution. This method is ran with deferred cancellation disabled by default. The Initial method is given a separate handler so that it can create temporary objects on it's own stack frame, rather than having objects created on **run()** that are only needed by startup and yet continue to consume stack space.

See also:

run

final

Reimplemented in **ost::TCPSession**, and **ost::UnixSession**.

bool ost::Thread::isDetached (void) const

Check if this thread is detached. **Returns:**

true if the thread is detached.

bool ost::Thread::isRunning (void) const
Verifies if the thread is still running or has already been terminated but not yet deleted. **Returns:**
true if the thread is still executing.

bool ost::Thread::isThread (void) const
Tests to see if the current execution context is the same as the specified thread object. **Returns:**
true if the current context is this object.

void ost::Thread::join (void)
Blocking call which unlocks when thread terminates.

virtual void ost::Thread::notify (Thread *) `[protected, virtual]`
When a thread terminates, it now sends a notification message to the parent thread which created it. The actual use of this notification is left to be defined in a derived class.

Parameters:
- the thread that has terminated.

void ost::Thread::resume (void)
Resumes execution of the selected thread.

virtual void ost::Thread::run (void) `[protected, pure virtual]`
All threads execute by deriving the Run method of **Thread**. This method is called after Initial to begin normal operation of the thread. If the method terminates, then the thread will also terminate after notifying it's parent and calling it's Final() method.

See also:
Initial

Examples:
bug1.cpp, **bug2.cpp**, **tcpservice.cpp**, **tcpstr1.cpp**, **tcpthread.cpp**, **thread1.cpp**, and **thread2.cpp**.

void ost::Thread::setCancel (Cancel mode) `[protected]`
Sets thread cancellation mode. Threads can either be set immune to termination (cancelDisabled), can be set to terminate when reaching specific 'thread cancellation points' (cancelDeferred) or immediately when Terminate is requested (cancelImmediate).

Parameters:
mode for cancellation of the current thread.

static void ost::Thread::setException (Throw mode) `[static]`
Set exception mode of the current thread. **Returns:**
exception mode.

void ost::Thread::setName (const char * text) `[protected]`
Set the name of the current thread. If the name is passed as NULL, then the default name is set (usually object pointer).

Parameters:
text name to use.

static void ost::Thread::setStack (size_t size = `0`**)** `[inline, static]`
Set base stack limit before manual stack sizes have effect. **Parameters:**
size stack size to set, or use 0 to clear autostack.

void ost::Thread::setSuspend (Suspend mode) `[protected]`
Sets the thread's ability to be suspended from execution. The thread may either have suspend enabled (suspendEnable) or disabled (suspendDisable).

Parameters:
mode for suspend.

static void ost::Thread::sleep (timeout_t msec) `[static]`
A thread-safe sleep call. On most Posix systems, 'sleep()' is implimented with SIGALRM making it unusable from multipe threads. Pthread libraries often define an alternate 'sleep' handler such as usleep(), nanosleep(), or nap(), that is thread safe, and also offers a higher timer resolution.

Parameters:

msec timeout in milliseconds.

int ost::Thread::start (Semaphore * start = 0)

When a new thread is created, it does not begin immediate execution. This is because the derived class virtual tables are not properly loaded at the time the C++ object is created within the constructor itself, at least in some compiler/system combinations. The thread can either be told to wait for an external semaphore, or it can be started directly after the constructor completes by calling the **start()** method.

Returns:

error code if execution fails.

Parameters:

start optional starting semaphore to alternately use.

Examples:

tcpservice.cpp, and **tcpstr1.cpp**.

void ost::Thread::suspend (void)

Suspends execution of the selected thread. Pthreads do not normally support suspendable threads, so the behavior is simulated with signals. On systems such as Linux that define threads as processes, SIGSTOP and SIGCONT may be used.

void ost::Thread::sync (void) [protected]

Used to wait for a join or cancel, in place of explicit exit.

void ost::Thread::terminate (void) [protected]

Used by another thread to terminate the current thread. Termination actually occurs based on the current **setCancel()** mode. When the current thread does terminate, control is returned to the requesting thread. **terminate()** should always be called at the start of any destructor of a class derived from **Thread** to assure the remaining part of the destructor is called without the thread still executing.

bool ost::Thread::testCancel (void) [protected]

test a cancellation point for deferred thread cancellation.

static void ost::Thread::yield (void) [static]

Yields the current thread's CPU time slice to allow another thread to begin immediate execution.

Friends And Related Function Documentation

friend class Cancellation [friend]

friend class DummyThread [friend]

void operator++ (Thread & th) [friend]

Signal the semaphore that the specified thread is waiting for before beginning execution. **Parameters:**

th specified thread.

void operator-- (Thread & th) [friend]

friend class PosixThread [friend]

friend class postream_type [friend]

friend class Slog [friend]

friend class ThreadImpl [friend]

Reimplemented in **ost::PosixThread**.

Author

Generated automatically by Doxygen for GNU CommonC++ from the source code.