



TS Thread Library Reference

Version 6.1, December 2003

IONA, IONA Technologies, the IONA logo, Orbix, Orbix/E, Orbacus, Artix, Orchestrator, Mobile Orchestrator, Enterprise Integrator, Adaptive Runtime Technology, Transparent Enterprise Deployment, and Total Business Integration are trademarks or registered trademarks of IONA Technologies PLC and/or its subsidiaries.

Java and J2EE are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

CORBA is a trademark or registered trademark of the Object Management Group, Inc. in the United States and other countries. All other trademarks that appear herein are the property of their respective owners.

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA Technologies PLC shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

COPYRIGHT NOTICE

No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, photocopying, recording or otherwise, without prior written consent of IONA Technologies PLC. No third party intellectual property right liability is assumed with respect to the use of the information contained herein. IONA Technologies PLC assumes no responsibility for errors or omissions contained in this book. This publication and features described herein are subject to change without notice.

Copyright © 2001–2003 IONA Technologies PLC. All rights reserved.

All products or services mentioned in this manual are covered by the trademarks, service marks, or product names as designated by the companies who market those products.

Updated: 24-Oct-2003

M 3 1 4 7

Contents

Threading and Synchronization Toolkit Overview	1
Timeouts	2
Execution Modes	2
Wrapper Classes	3
Inlined Classes	3
Setting an Execution Mode	3
Errors and Exceptions	4
IT_Condition Class	7
IT_Condition::broadcast()	7
IT_Condition::IT_Condition() Constructor	8
IT_Condition::~~IT_Condition() Destructor	8
IT_Condition::signal()	8
IT_Condition::wait()	9
IT_CurrentThread Class	11
IT_CurrentThread::cleanup()	11
IT_CurrentThread::id()	11
IT_CurrentThread::is_main_thread()	12
IT_CurrentThread::self()	12
IT_CurrentThread::sleep()	12
IT_CurrentThread::yield()	13
IT_DefaultTSErrorHandler Class	15
IT_DefaultTSErrorHandler::handle()	15
IT_DefaultTSErrorHandler::~~IT_DefaultTSErrorHandler() Destructor	15
IT_Gateway Class	17
IT_Gateway::close()	17
IT_Gateway::IT_Gateway() Constructor	18
IT_Gateway::~~IT_Gateway() Destructor	18
IT_Gateway::open()	18
IT_Gateway::wait()	19

Table of Contents

IT_Locker Template Class	21
IT_Locker::cancel()	23
IT_Locker::is_locked()	23
IT_Locker::IT_Locker()	23
IT_Locker::~IT_Locker()	25
IT_Locker::lock()	25
IT_Locker::mutex()	25
IT_Locker::trylock()	26
IT_Mutex Class	27
IT_Mutex::IT_Mutex() Constructor	28
IT_Mutex::~IT_Mutex() Destructor	28
IT_Mutex::lock()	28
IT_Mutex::trylock()	29
IT_Mutex::unlock()	29
IT_PODMutex Structure	31
IT_PODMutex::lock()	31
IT_PODMutex::m_index Data Type	32
IT_PODMutex::trylock()	32
IT_PODMutex::unlock()	33
IT_RecursiveMutex Class	35
IT_RecursiveMutex::IT_RecursiveMutex() Constructor	36
IT_RecursiveMutex::~IT_RecursiveMutex() Destructor	36
IT_RecursiveMutex::lock()	36
IT_RecursiveMutex::trylock()	37
IT_RecursiveMutex::unlock()	37
IT_RecursiveMutexLocker Class	39
IT_RecursiveMutexLocker::cancel()	41
IT_RecursiveMutexLocker::IT_RecursiveMutexLocker() Constructors	41
IT_RecursiveMutexLocker::~IT_RecursiveMutexLocker() Destructor	42
IT_RecursiveMutexLocker::lock()	43
IT_RecursiveMutexLocker::lock_count()	43
IT_RecursiveMutexLocker::mutex()	43
IT_RecursiveMutexLocker::trylock()	43

IT_RecursiveMutexLocker::unlock()	44
IT_Semaphore Class	45
IT_Semaphore::IT_Semaphore() Constructor	45
IT_Semaphore::~IT_Semaphore() Destructor	46
IT_Semaphore::post()	46
IT_Semaphore::trywait()	46
IT_Semaphore::wait()	47
IT_TerminationHandler Class	49
IT_TerminationHandler()	50
~IT_TerminationHandler()	50
IT_Thread Class	51
IT_Thread::id()	52
IT_Thread::is_null()	52
IT_Thread::IT_Thread() Constructors	52
IT_Thread::~IT_Thread() Destructor	53
IT_Thread::join()	53
IT_Thread::operator=()	53
IT_Thread::operator==(())	54
IT_Thread::operator!=(())	54
IT_Thread::thread_failed Constant	55
IT_ThreadBody Class	57
IT_ThreadBody::~IT_ThreadBody() Destructor	57
IT_ThreadBody::run()	57
IT_ThreadFactory Class	59
IT_ThreadFactory::DetachState Enumeration	60
IT_ThreadFactory::IT_ThreadFactory() Constructor	60
IT_ThreadFactory::~IT_ThreadFactory() Destructor	60
IT_ThreadFactory::smf_start()	61
IT_ThreadFactory::start()	61
IT_TimedCountByNSemaphore Class	63
IT_TimedCountByNSemaphore::infinite_size Constant	64

Table of Contents

IT_TimedCountByNSemaphore::infinite_timeout Constant	64
IT_TimedCountByNSemaphore::IT_TimedCountByNSemaphore() Constructor	64
IT_TimedCountByNSemaphore::~~IT_TimedCountByNSemaphore() Destructor	65
IT_TimedCountByNSemaphore::post()	65
IT_TimedCountByNSemaphore::trywait()	65
IT_TimedCountByNSemaphore::wait()	66
IT_TimedOneshot Class	67
IT_TimedOneshot::infinite_timeout Constant	68
IT_TimedOneshot::IT_TimedOneshot() Constructor	68
IT_TimedOneshot::~~IT_TimedOneshot() Destructor	68
IT_TimedOneshot::reset()	69
IT_TimedOneshot::signal()	69
IT_TimedOneshot::trywait()	70
IT_TimedOneshot::wait()	70
IT_TimedSemaphore Class	73
IT_TimedSemaphore::infinite_timeout Constant	74
IT_TimedSemaphore::IT_TimedSemaphore() Constructor	74
IT_TimedSemaphore::~~IT_TimedSemaphore() Destructor	74
IT_TimedSemaphore::post()	74
IT_TimedSemaphore::trywait()	75
IT_TimedSemaphore::wait()	75
IT_TSBadAlloc Error Class	77
IT_TSError Error Class	79
IT_TSError::IT_TSError() Constructors	79
IT_TSError::~~IT_TSError() Destructor	80
IT_TSError::OS_error_number()	80
IT_TSError::raise()	80
IT_TSError::TS_error_code()	80
IT_TSError::what()	81
IT_TSErrorHandler Class	83
IT_TSErrorHandler::handle()	83
IT_TSErrorHandler::~~IT_TSErrorHandler() Destructor	83

IT_TSLogic Error Class	85
IT_TSRuntime Error Class	87
IT_TSVoidStar Class	89
IT_TSVoidStar::IT_TSVoidStar() Constructor	89
IT_TSVoidStar::~~IT_TSVoidStar() Destructor	90
IT_TSVoidStar::get()	91
IT_TSVoidStar::set()	91
Index	93

Table of Contents

Threading and Synchronization Toolkit Overview

The Threading and Synchronization (TS) toolkit provides an object-oriented and platform-neutral abstraction that hides the diverse, lower-level, thread packages. [Table 1](#) shows the threading and synchronization (TS) classes organized into some useful groups.

Table 1: *TS Thread Classes*

Thread Management	<u>IT_CurrentThread</u> <u>IT_Thread</u> <u>IT_ThreadBody</u> <u>IT_ThreadFactory</u> <u>IT_TerminationHandler</u> <u>IT_TSVoidStar</u>
Thread Errors and Exceptions	<u>IT_TSBadAlloc</u> <u>IT_DefaultTSErrorHandler</u> <u>IT_TSError</u> <u>IT_TSErrorHandler</u> <u>IT_TSLogic</u> <u>IT_TSRuntime</u>
Mutex Locks	<u>IT_Locker</u> <u>IT_Mutex</u> <u>IT_PODMutex</u> <u>IT_RecursiveMutex</u> <u>IT_RecursiveMutexLocker</u>
Thread Synchronization	<u>IT_Condition</u> <u>IT_Gateway</u> <u>IT_Semaphore</u> <u>IT_TimedCountByNSemaphore</u> <u>IT_TimedOneshot</u> <u>IT_TimedSemaphore</u>

The rest of this overview covers these topics:

-
- [“Timeouts”](#)
 - [“Execution Modes”](#)
 - [“Errors and Exceptions”](#)

Timeouts

Timeouts are expressed in milliseconds. They represent the time period from the invocation of the timed method until the expiration of the timer. This time-out period is approximate because it is affected by the number and kind of interrupts received and by the changes external sources may make to the system’s time.

Execution Modes

The TS classes are designed to be efficient and to help you write code that is correct and portable across various platforms. You can build TS applications in either of the following modes:

Unchecked	This is the normal production mode. Inexpensive checks, such as checking values returned by the API, are performed, but a minimum of memory, locking, and system calls are used to implement TS features.
Checked	In this mode, extra-checking is performed to detect erroneous or non-portable situations. On platforms that support exceptions, exceptions are raised to report such errors. This mode may be less time or space efficient than the unchecked mode.

The effect of a program that runs correctly (the program does not create any TS error object) in the checked mode is identical to that of the unchecked mode.

TS provides two kinds of classes in different sets of header files. These include wrapper and inline classes.

Wrapper Classes

Wrapper classes are the recommended classes to use because you can switch between checked and unchecked modes by simply re-linking without recompiling your application. These clean, platform-neutral wrapper classes simply delegate to the appropriate inlined classes for whichever mode you are using.

The wrapper classes are in header files ending in `.h`.

Inlined Classes

To minimize the delegation overhead of wrapper classes, the TS toolkit also provides C++ classes with only inlined member methods and pre-preprocessor directives. These inline classes accommodate the differences between the underlying thread packages.

Delegation overhead for a normal method call is generally negligible, but you can save on this overhead by using these inlined classes directly. However by using these header files, you will need to recompile your application whenever you want to switch between checked and unchecked modes, and each time even minor improvements are made to the TS implementation.

The inline classes are in header files ending in `_i.h`.

Setting an Execution Mode

[Table 2](#) shows the default settings for each platform.

Table 2: *Default Thread Settings*

Platform	Thread Primitives	Default Mode
HPUX 11 Solaris 2.6	Posix	unchecked
HPUX 10.20 Other Solaris	DCE UI	unchecked
Win32	Win32	unchecked

To set a different mode, you reset the library by inserting the preferred `lib` subdirectory at the beginning of your `LD_LIBRARY_PATH` or `SHLIB_PATH`. For example, to reset to the checked mode, do the following for your respective platform:

Solaris	Put the following at the beginning of your <code>LD_LIBRARY_PATH</code> : <code>/vob/common/ts/lib/posix/checked</code>
HPUX 10.20	Put the following at the beginning of your <code>SHLIB_PATH</code> : <code>/vob/common/ts/lib/dce/checked</code>
HPUX 11.00	Put the following at the beginning of your <code>SHLIB_PATH</code> : <code>/vob/common/ts/lib/posix/checked</code>
NT	Put the following at the beginning of your <code>PATH</code> : <code>/common/ts/lib/win32/checked</code>

Errors and Exceptions

[Table 3](#) summarizes the TS error classes:

Table 3: *Error and Exception Classes*

Control	Exceptions
IT_DefaultTSErrorHandler	IT_TSBadAlloc
IT_TSError	IT_TSLogic
IT_TSErrorHandler	IT_TSRuntime

The TS API allows you to use either error parameters or exceptions. The last parameter of almost every TS method is a reference to an error handler object of the class [IT_TSErrorHandler](#). When a TS method detects an error, it creates an [IT_TSError](#) object and passes it to [IT_TSErrorHandler::handle\(\)](#).

TS errors form the hierarchy shown in [Figure 1](#). An [IT_TSRuntime](#) error generally signals an error detected by the operating system or the underlying thread package. An [IT_TSLogic](#) error reports a logic error in your program, for example, when a thread tries to release a lock it does not own. Logic

errors are either detected by the underlying thread package, or by extra checking code in checked mode. An [IT_TSBadAlloc](#) error signals that the `new` operator failed.

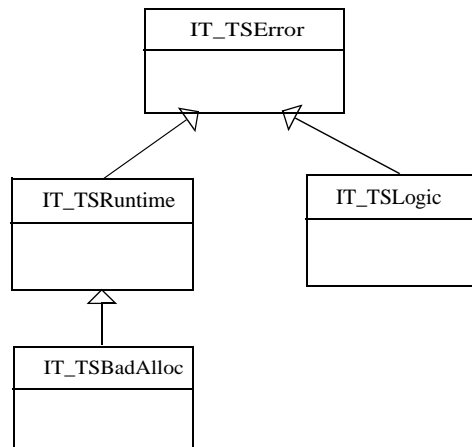


Figure 1: *The TS Error Class Hierarchy*

The TS API provides a default, static, and stateless error handler named [IT_DefaultTSErrorHandler](#). If you use exceptions, this error handler throws [IT_TSError](#) objects. In environments that do not use exceptions this handler aborts the process.

For most applications, the default error handler object provides the desired behavior. In this situation, instead of passing an [IT_DefaultTSErrorHandler](#) object each time you call a TS method, you can define in your build command the environment variable `IT_TS_DEFAULTED`. This will instruct the TS API to use the default error handler object for the error handler parameter. For example:

```

#ifndef IT_TS_DEFAULT_ERROR_HANDLER
#ifdef IT_TS_DEFAULTED
#define IT_TS_DEFAULT_ERROR_HANDLER = IT_DefaultTSErrorHandler
#else
#define IT_TS_DEFAULT_ERROR_HANDLER
#endif
#endif

```

C++ destructors do not have parameters, and as result, cannot be given an error handler object parameter. In the checked mode, the TS API reports errors in destructors to the default error handler object. In the unchecked mode, the TS API does not report errors that occur in destructors.

Because default parameters are not part of the function-type in C++, the TS library can be built with or without defining `IT_TS_DEFAULTED`. Also, the same library can be used by modules that use the defaulted parameter and by modules built without defining `IT_TS_DEFAULTED`.

If you intend to use your own error handler objects in your application, it is strongly recommended that you do not define `IT_TS_DEFAULTED` to avoid using the default error handler object by mistake. If you want to consistently use the same error handler object, you can define

`IT_TS_DEFAULT_ERROR_HANDLER` in your command or in a non-exported file.

For example:

```
#define IT_TS_DEFAULT_ERROR_HANDLER = myErrorHandler;
```

IT_Condition Class

The `IT_Condition` class provides a signalling mechanism that events use to synchronize when sharing a mutex. In one atomic operation, a condition wait both releases the mutex and waits until another thread signals or broadcasts a change of state for the condition.

```
class IT_Condition {
public:
    IT\_Condition(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );
    ~IT\_Condition();
    void wait(
        IT_Mutex& app_mutex,
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );
    void wait(
        IT_MutexLocker& locker,
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );
    void signal(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );
    void broadcast(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );
};
```

`IT_Condition::broadcast()`

```
void broadcast(
    IT\_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
);
```

Wakes up all waiting threads. One thread acquires the mutex and resumes with the associated mutex lock. The rest of the threads continue waiting.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

See Also [IT_Mutex](#)

IT_Condition::IT_Condition() Constructor

```
IT_Condition(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

The constructor for an `IT_Condition` object.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

IT_Condition::~~IT_Condition() Destructor

```
~IT_Condition();
```

The destructor for an `IT_Condition` object.

Enhancement Orbix enhancement.

IT_Condition::signal()

```
void signal(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Wakes up a single waiting thread. The thread resumes with the associated mutex locked.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

IT_Condition::wait()

```
void wait(  
    IT\_Mutex& app_mutex,  
    IT\_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);  
  
void wait(  
    IT_MutexLocker& locker,  
    IT\_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Atomically releases the mutex, and waits until another thread calls `signal()` or `broadcast()`.

Parameters

`app_mutex` Use the mutex `app_mutex`.
`locker` Use the mutex in `locker`.
`eh`

The mutex must always be locked when `wait()` is called. When a condition wakes up from a wait, it resumes with the mutex locked.

Enhancement Orbix enhancement.

IT_CurrentThread Class

The `IT_CurrentThread` class gives access to the current thread. It has only static member methods.

```
class IT_TS_API IT_CurrentThread {
public:
    static IT_Thread self(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    static int is\_main\_thread();

    static void cleanup();

    static void yield();

    static void sleep(
        unsigned long milliseconds
    );

    static long id();
};
```

IT_CurrentThread::cleanup()

```
static void cleanup();
```

Cleans up thread-specific data. A thread typically calls `cleanup()` before exiting. Threads created with an [IT_ThreadFactory](#) do this automatically.

Enhancement Orbix enhancement.

IT_CurrentThread::id()

```
static long id();
```

Returns a unique identifier for the current thread.

Enhancement Orbix enhancement.

IT_CurrentThread::is_main_thread()

```
static int is_main_thread();
```

Returns 1 if the caller is the main thread, but returns 0 if it is not.

Enhancement Orbix enhancement.

IT_CurrentThread::self()

```
static IT_Thread self(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Returns an [IT_Thread](#) object for the thread that calls this method.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

IT_CurrentThread::sleep()

```
static void sleep(  
    unsigned long milliseconds  
);
```

Suspends the current thread for the approximate number of milliseconds input.

Parameters

milliseconds The length of time in milliseconds to suspend the thread.

Enhancement Orbix enhancement.

IT_CurrentThread::yield()

```
static void yield();
```

Yields the CPU to another thread of equal priority, if one is available.

Enhancement Orbix enhancement.

IT_DefaultTSErrHandler Class

The `IT_DefaultTSErrHandler` class is the default TS error handler. If you use exceptions, this error handler throws [IT_TSErr](#) objects. In environments that do not use exceptions this handler aborts the process.

```
class IT_DefaultTSErrHandler : public IT_TSErrHandler{
public:
    virtual ~IT\_DefaultTSErrHandler\(\)
    virtual void handle(
        const IT_TSErr& this_err
    );
};
```

[See page 4](#) for more on error handling.

IT_DefaultTSErrHandler::handle()

```
void handle(
    const IT\_TSErr& this_err
);
```

Do appropriate processing for the given error.

Parameters

`this_err` A reference to an error object.

Enhancement Orbix enhancement.

IT_DefaultTSErrHandler::~~IT_DefaultTSErrHandler() Destructor

```
~IT_DefaultTSErrHandler()
```

The destructor for the error handler object.

Enhancement Orbix enhancement.

IT_Gateway Class

The `IT_Gateway` class provides a gate where a set of threads can only do work if the gate is open.

```
class IT_Gateway {
public:
    IT\_Gateway(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );
    ~IT\_Gateway();

    void open(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    void close(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    void wait(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

private:
    ...
}
```

`IT_Gateway::close()`

```
void close(
    IT\_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
);
```

Close the gateway so no threads can do any work.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

IT_Gateway::IT_Gateway() Constructor

```
IT_Gateway(  
    IT\_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

The gateway constructor.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

IT_Gateway::~~IT_Gateway() Destructor

```
~IT_Gateway();
```

The destructor.

Enhancement Orbix enhancement.

IT_Gateway::open()

```
void open(  
    IT\_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Open the gateway to allow threads to work.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

IT_Gateway::wait()

```
void wait(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Wait for a thread to finish.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

IT_Locker Template Class

`IT_Locker` is a helper class for locking and unlocking non-recursive mutexes, including [IT_Mutex](#) and [IT_PODMutex](#) objects. Typically a locker locks a mutex in its constructor and releases it in its destructor. This is particularly useful for writing clean code that behaves properly when an exception is raised.

An `IT_Locker` object must be created on the stack of a particular thread, and must never be shared by more than one thread.

The `IT_Locker` method definitions are inlined directly in the class declaration, because these methods call each other. If a definition calls a method that is not previously declared inlined, this method is generated out of line, regardless of its definition (which can be provided later in the translation unit with the inline keyword).

```
template<class T> class IT_Locker {
public:
    IT\_Locker(
        T& mutex,
        IT_TErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    ) :
        m_mutex(mutex),
        m_locked(0),
        m_error_handler(eh)
    {
        lock();
    }

    IT\_Locker(
        T& mutex,
        int wait,
        IT_TErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    ) :
        m_mutex(mutex),
        m_locked(0),
        m_error_handler(eh)
    {
        if (wait)
```

```
        {
            lock();
        }
        else
        {
            trylock();
        }
    }

~IT_Locker()
{
    cancel();
}

void cancel()
{
    if (m_locked)
    {
        m_mutex.unlock(m_error_handler);
        m_locked = 0;
    }
}

int is_locked()
{
    return m_locked;
}

void lock()
{
    m_mutex.lock(m_error_handler);
    m_locked = 1;
}

int trylock()
{
    return (m_locked = m_mutex.trylock(m_error_handler));
}

T& mutex()
{
    return m_mutex;
}
```

```
private:
...
```

IT_Locker::cancel()

```
void cancel() {
    if (m_locked)
    {
        m_mutex.unlock(m_error_handler);
        m_locked = 0;
    }
}
```

Releases the mutex only if it is locked by this locker. You can call `cancel()` safely even when the mutex is not locked.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSRuntime](#)
[IT_TSLogic](#)

IT_Locker::is_locked()

```
int is_locked() {
    return m_locked;
}
```

returns 1 if this mutex locker has the lock and returns 0 if it does not.

Enhancement Orbix enhancement.

IT_Locker::IT_Locker()

```
IT_Locker(
    T& mutex,
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
) :
    m_mutex(mutex),
    m_locked(0),
```

```
        m_error_handler(eh)
    {
        lock();
    }
```

A constructor for a locker object that locks the given mutex.

```
IT_Locker(
    T& mutex,
    int wait,
    IT\_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
) :
    m_mutex(mutex),
    m_locked(0),
    m_error_handler(eh)
{
    if (wait)
    {
        lock();
    }
    else
    {
        trylock();
    }
}
```

A constructor for a locker object.

Parameters

mutex	The mutex to which the locker applies.
wait	If wait has a value of 1, this constructor waits to acquire the lock. If wait has a value of 0, the constructor only tries to lock the mutex.
eh	A reference to an error handler object.

Enhancement Orbix enhancement.

See Also [IT_Locker::trylock\(\)](#)

IT_Locker::~~IT_Locker()

```
~IT_Locker()  
{  
    cancel();  
}
```

The destructor releases the mutex if it is locked by this locker.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSLogic](#)
[IT_TSRuntime](#)

IT_Locker::lock()

```
void lock()  
{  
    m_mutex.lock(m_error_handler);  
    m_locked = 1;  
}
```

Locks the mutex associated with the locker.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSLogic](#)
[IT_TSRuntime](#)

IT_Locker::mutex()

```
T& mutex()  
{  
    return m_mutex;  
}
```

Returns direct access to the locker's mutex.

Enhancement Orbix enhancement.

IT_Locker::trylock()

```
int trylock()
{
    return (m_locked = m_mutex.trylock(m_error_handler));
}
```

Tries to lock the mutex. Returns 1 if the mutex is successfully locked or 0 if it is not locked.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSLogic](#)

[IT_TSRuntime](#)

IT_Mutex Class

An `IT_Mutex` object is a synchronization primitive for mutual exclusion locks.

When a thread has successfully locked, it is said to own the `IT_Mutex`. `IT_Mutex` objects have scope only within a single process (they are not shared by several processes) and they are not recursive. When a thread that owns an `IT_Mutex` attempts to lock it again, a deadlock occurs.

You use an `IT_Mutex` in conjunction with an [IT_Locker](#) object to lock and unlock your mutexes.

```
class IT_Mutex {
public:
    IT\_Mutex(
        IT_TErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    ~IT\_Mutex();

    void lock(
        IT_TErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    void unlock(
        IT_TErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    int trylock(
        IT_TErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

private:
    // ...
};
```

See Also

[IT_Locker](#)
[IT_RecursiveMutex](#)

IT_Mutex::IT_Mutex() Constructor

```
IT_Mutex(I
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
);
```

Constructs an IT_Mutex object. It is initially unlocked.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions The [IT_TSRuntime](#) error can be reported.

IT_Mutex::~~IT_Mutex() Destructor

```
IT_Mutex();
```

The destructor for the mutex.

Enhancement Orbix enhancement.

IT_Mutex::lock()

```
void lock(
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
);
```

Blocks until the IT_Mutex can be acquired.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSLogic](#)
[IT_TSRuntime](#)

IT_Mutex::trylock()

```
int trylock(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Tries to acquire the lock. If successful, the method returns a 1 immediately, otherwise it returns a 0 and does not block.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSLogic](#)
[IT_TSRuntime](#)

IT_Mutex::unlock()

```
void unlock(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Releases this `IT_Mutex`. Only the owner thread of an `IT_Mutex` is allowed to release an `IT_Mutex`.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSLogic](#)
[IT_TSRuntime](#)

IT_PODMutex Structure

An `IT_PODMutex` is a mutex for a “plain old data” (POD) structure. Just as with a standard C++ PODS, an `IT_PODMutex` can be fully initialized at compile time without the overhead of an explicit constructor call. This is particularly useful for static objects. Likewise, the object can be destroyed without an explicit destructor call (in a manner similar to the C language).

You can use the built-in definition `IT_POD_MUTEX_INIT` to easily initialize an `IT_PODMutex` to zero. For example:

```
static IT_PODMutex my_global_mutex = IT_POD_MUTEX_INIT;
```

You use an `IT_PODMutex` in conjunction with an [IT_Locker](#) object to lock and unlock your mutexes. The structure members for an `IT_PODMutex` include the following:

```
struct IT_TS_API IT_PODMutex {
    void lock(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );
    int trylock(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );
    void unlock(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );
    // DO NOT USE and DO NOT MAKE PRIVATE
    unsigned char m_index;
};
```

See Also

[IT_Locker](#)
[IT_Mutex](#)

IT_PODMutex::lock()

```
void lock(
    IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
);
```

Blocks until the mutex can be acquired.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSLogic](#)

[IT_TSRuntime](#)

IT_PODMutex::m_index Data Type

```
unsigned char m_index;
```

Note: For internal use only.

IT_PODMutex::trylock()

```
int trylock(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Tries to acquire the mutex lock. If `trylock()` succeeds, it returns a 1 immediately. Otherwise it returns 0.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSLogic](#)

[IT_TSRuntime](#)

IT_PODMutex::unlock()

```
void unlock(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Releases the mutex lock. Only the owner of a mutex is allowed to release it.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSLogic](#)
[IT_TSRuntime](#)

IT_RecursiveMutex Class

An `IT_RecursiveMutex` object is a synchronization primitive for mutual exclusion. In general do not use it directly.

Note: It is strongly recommended that you use the [IT_RecursiveMutexLocker](#) to lock and unlock your recursive mutexes.

In most respects an `IT_RecursiveMutex` object is similar to an [IT_Mutex](#) object. However, it can be locked recursively, which means that a thread that already owns a recursive mutex object can lock it again in a deeper scope without creating a deadlock condition.

When a thread has successfully locked a recursive mutex, it is said to own it. Recursive mutex objects have process-scope which means that they are not shared by several processes.

To release an `IT_RecursiveMutex`, its owner thread must call `unlock()` the same number of times that it called `lock()`.

```
class IT_RecursiveMutex {
public:
    IT\_RecursiveMutex(
        IT_TErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    ~IT\_RecursiveMutex();

    void lock(
        IT_TErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    void unlock(
        IT_TErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    int trylock(
        IT_TErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
```

```
    );  
private:  
    ...
```

See Also

[IT_Mutex](#)
[IT_RecursiveMutexLocker](#)

IT_RecursiveMutex::IT_RecursiveMutex() Constructor

```
IT_RecursiveMutex(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Constructs an `IT_RecursiveMutex` object. It is initially unlocked.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions The [IT_TSRuntime](#) error can be reported.

IT_RecursiveMutex::~~IT_RecursiveMutex() Destructor

```
~IT_RecursiveMutex();
```

Destructor for an `IT_RecursiveMutex` object.

Enhancement Orbix enhancement.

IT_RecursiveMutex::lock()

```
void lock(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Blocks until the recursive mutex can be acquired.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions The [IT_TSRuntime](#) error can be reported.

IT_RecursiveMutex::trylock()

```
int trylock(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Tries to acquire the recursive mutex. If it succeeds, returns 1 immediately; otherwise returns 0.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions The [IT_TSRuntime](#) error can be reported.

IT_RecursiveMutex::unlock()

```
void unlock(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Releases this recursive mutex (one count). Only the owner of a mutex is allowed to release it.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSRuntime](#)
[IT_TSLogic](#)

IT_RecursiveMutexLocker Class

The `IT_RecursiveMutexLocker` is a locker for recursive mutexes. The `IT_RecursiveMutexLocker` methods are defined as inline in the class declaration, because these methods call each other.

```
class IT_RecursiveMutexLocker {
public:
    IT\_RecursiveMutexLocker(
        IT_RecursiveMutex& m,
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    ) :
        m_recursive_mutex(m),
        m_lock_count(0),
        m_error_handler(eh)
    {
        lock();
    }

    IT\_RecursiveMutexLocker(
        IT_RecursiveMutex& m,
        int wait,
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    ) :
        m_recursive_mutex(m),
        m_lock_count(0),
        m_error_handler(eh)
    {
        if (wait)
        {
            lock();
        }
        else
        {
            trylock();
        }
    }

    ~IT\_RecursiveMutexLocker()
}
```

```
{
    cancel();
}

void cancel()
{
    while (m_lock_count > 0)
    {
        m_recursive_mutex.unlock(m_error_handler);
        m_lock_count--;
    }
}

void lock()
{
    m_recursive_mutex.lock(m_error_handler);
    m_lock_count++;
}

unsigned int lock_count()
{
    return m_lock_count;
}

int trylock()
{
    if (m_recursive_mutex.trylock(m_error_handler) == 1)
    {
        m_lock_count++;
        return 1;
    }
    else
    {
        return 0;
    }
}

void unlock()
{
    m_recursive_mutex.unlock(m_error_handler);
    m_lock_count--;
}
```

```
IT_RecursiveMutex& mutex()  
{  
    return m_recursive_mutex;  
}
```

```
Private:  
...
```

IT_RecursiveMutexLocker::cancel()

```
void cancel() {  
    while (m_lock_count > 0)  
    {  
        m_recursive_mutex.unlock(m_error_handler);  
        m_lock_count--;  
    }  
}
```

Releases all locks held by this recursive mutex locker. The `cancel()` method can be called safely even when the recursive mutex is not locked.

Enhancement Orbix enhancement.

IT_RecursiveMutexLocker::IT_RecursiveMutexLocker() Constructors

```
IT_RecursiveMutexLocker(  
    IT\_RecursiveMutex& m,  
    IT\_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
) :  
    m_recursive_mutex(m),  
    m_lock_count(0),  
    m_error_handler(eh)  
{  
    lock();  
}
```

Constructs a recursive mutex locker object. This constructor locks the given recursive mutex.

```
IT_RecursiveMutexLocker(  
    IT\_RecursiveMutex& m,
```

```

        int wait,
        IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    ) :
        m_recursive_mutex(m),
        m_lock_count(0),
        m_error_handler(eh)
    {
        if (wait)
        {
            lock();
        }
        else
        {
            trylock();
        }
    }

```

Constructs a recursive mutex locker object.

Parameters

<code>m</code>	The mutex to which the locker applies.
<code>wait</code>	If <code>wait</code> has a value of 1, this constructor waits to acquire the lock. If <code>wait</code> has a value of 0, it only tries to lock the recursive mutex.
<code>eh</code>	A reference to an error handler object.

Enhancement Orbix enhancement.

IT_RecursiveMutexLocker::~~IT_RecursiveMutexLocker() **Destructor**

```

~IT_RecursiveMutexLocker()
{
    cancel();
}

```

The destructor releases all locks held by this recursive mutex locker.

Enhancement Orbix enhancement.

IT_RecursiveMutexLocker::lock()

```
void lock()
{
    m_recursive_mutex.lock(m_error_handler);
    m_lock_count++;
}
```

Acquires the lock.

Enhancement Orbix enhancement.

IT_RecursiveMutexLocker::lock_count()

```
unsigned int lock_count()
{
    return m_lock_count;
}
```

Returns the number of locks held by this recursive mutex locker.

Enhancement Orbix enhancement.

IT_RecursiveMutexLocker::mutex()

```
IT_RecursiveMutex& mutex()
{
    return m_recursive_mutex;
}
```

Returns direct access to the locker's recursive mutex.

Enhancement Orbix enhancement.

IT_RecursiveMutexLocker::trylock()

```
int trylock()
{
    if (m_recursive_mutex.trylock(m_error_handler) == 1)
    {
        m_lock_count++;
    }
}
```

```
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Tries to acquire one lock for the recursive mutex. Returns 1 if the mutex lock is successfully acquired or 0 if it is not.

Enhancement Orbix enhancement.

IT_RecursiveMutexLocker::unlock()

```
void unlock()
{
    m_recursive_mutex.unlock(m_error_handler);
    m_lock_count--;
}
```

Releases one lock held by this recursive mutex.

Enhancement Orbix enhancement.

IT_Semaphore Class

A semaphore is a non-negative counter, typically used to coordinate access to some resources.

```
class IT_Semaphore {
public:
    IT\_Semaphore(
        size_t initialCount,
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    ~IT\_Semaphore();

    void post(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    void wait(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    int trywait(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

private:
    // ...
};
```

IT_Semaphore::IT_Semaphore() Constructor

```
IT_Semaphore(
    size_t initialCount,
    IT\_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
);
```

A semaphore constructor that initializes the semaphore's counter with the value `initialCount`.

Parameters

`initialCount` A positive integer value.
`eh` A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions The [IT_TSRuntime](#) error can be reported.

IT_Semaphore::~~IT_Semaphore() Destructor

```
~IT_Semaphore();
```

Destroys the semaphore.

Enhancement Orbix enhancement.

IT_Semaphore::post()

```
void post(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Posts a resource thread with the semaphore. This method increments the semaphore's counter and wakes up a thread that might be blocked on [wait\(\)](#).

Parameters

`eh` A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions The [IT_TSRuntime](#) error can be reported.

IT_Semaphore::trywait()

```
int trywait(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Tries to get a resource thread. The method returns 1 if it succeeds, and 0 if it fails.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions An error that can be reported is [IT_TSRuntime](#).

IT_Semaphore::wait()

```
void wait(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Waits for one resource. The `wait()` method blocks if the semaphore's counter value is 0 and decrements the counter if the counter's value is greater than 0.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSLogic](#)
[IT_TSRuntime](#)

See Also [IT_TimedSemaphore](#)
[IT_TimedCountByNSemaphore](#)

IT_TerminationHandler Class

The `IT_TerminationHandler` class enables server applications to handle delivery of `CTRL_C` and similar events in a portable manner. On UNIX, the termination handler handles the following signals:

```
SIGINT  
SIGTERM  
SIGQUIT
```

On Windows, the termination handler is a wrapper around `SetConsoleCtrlHandler`, which handles delivery of the following control events:

```
CTRL_C_EVENT  
CTRL_BREAK_EVENT  
CTRL_SHUTDOWN_EVENT  
CTRL_LOGOFF_EVENT  
CTRL_CLOSE_EVENT
```

You can create only one termination handler object in a program.

```
#include <it_ts/ts_error.h>  
  
typedef void (*IT_TerminationHandlerFunctionPtr)(long);  
  
class IT_IFC_API IT_TerminationHandler  
{  
public:  
  
    IT\_TerminationHandler(  
        IT_TerminationHandlerFunctionPtr f,  
        IT_ExceptionHandler& eh = IT_EXCEPTION_HANDLER  
    );  
  
    ~IT\_TerminationHandler();  
};
```

IT_TerminationHandler()

```
IT_TerminationHandler(  
    IT_TerminationHandlerFunctionPtr f,  
    IT_ExceptionHandler& eh = IT_EXCEPTION_HANDLER  
);
```

Creates a termination handler object on the stack. On POSIX platforms, it is critical to create this object in the main thread before creation of any other thread, and especially before ORB initialization.

Parameters

- f The callback function registered by the application. The callback function takes a single `long` argument:
- On UNIX, the signal number on Unix/POSIX
 - On Windows, the type of event caught

~IT_TerminationHandler()

```
~IT_TerminationHandler();
```

Deregisters the callback, in order to avoid calling it during static destruction.

IT_Thread Class

An `IT_Thread` object represents a thread of control. An `IT_Thread` object can be associated with a running thread, associated with a thread that has already terminated, or it can be null, which means it is not associated with any thread.

The important class members are as follows:

```
class IT_Thread {
public:
    IT\_Thread\(\);

    ~IT\_Thread\(\);

    IT\_Thread(
        const IT_Thread& other
    );

    IT_Thread& operator=(
        const IT_Thread& other
    );

    int operator==(
        const IT_Thread& x
    ) const;

    int operator!=(
        const IT_Thread& x
    ) const
    {
        return ! operator==(x);
    }

    int is\_null() const;

    static void* const thread\_failed;

    void* join(
        IT_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
```

```
        ) const;  
  
        long id() const;  
        ...  
};
```

IT_Thread::id()

```
long id() const;
```

Returns a unique thread identifier. This method is useful for debugging.

Enhancement Orbix enhancement.

IT_Thread::is_null()

```
int is_null() const;
```

Tests if this is a null `IT_Thread` object.

Enhancement Orbix enhancement.

IT_Thread::IT_Thread() Constructors

```
IT_Thread(  
    IT_Thread_i* t=0  
);
```

Constructs a null `IT_Thread` object.

```
IT_Thread (  
    const IT_Thread& other  
);
```

Copies the `IT_Thread` object. This constructor does not start a new thread.

Parameters

`other` The original thread to copy.

Enhancement Orbix enhancement.

IT_Thread::~~IT_Thread() Destructor

```
~IT_Thread();
```

Destructor for an `IT_Thread` object.

Enhancement Orbix enhancement.

IT_Thread::join()

```
void* join(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
    ) const;
```

Waits until the thread has terminated and returns its exit status. At most one thread can successfully join a given thread, and only `Attached` threads can be joined. Note that even in the checked mode, `join()` does not always detect that you tried to join a `Detached` thread, or that you joined the same thread several times.

Parameters

`eh` A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSLogic](#)
[IT_TSRuntime](#)

See Also

[IT_CurrentThread](#)
[IT_ThreadBody](#)

IT_Thread::operator=()

```
IT_Thread& operator=(  
    const IT_Thread& other  
    );
```

Assignment operator that copies the `IT_Thread` object. This does not start a new thread.

Parameters

`other` The original thread that is copied.

Enhancement Orbix enhancement.

IT_Thread::operator==()

```
int operator==(
    const IT_Thread& x
) const;
```

Operator that checks if two `IT_Thread` objects refer to the same thread. Returns 1 if the two objects refer to the same thread or it returns 0 if they do not refer to the same thread.

Parameters

`x` The thread to compare to this thread.

Enhancement Orbix enhancement.

IT_Thread::operator!=()

```
int operator!=(
    const IT_Thread& x
) const
```

Operator that checks if two `IT_Thread` objects refer to different threads. Returns 1 if the two objects refer to different threads or it returns 0 if they refer to the same thread.

Parameters

`x` The thread to compare to this thread.

Enhancement Orbix enhancement.

IT_Thread::thread_failed Constant

```
static void* const thread_failed;
```

The constant `thread_failed` is the return status of a thread to report a failure. It is neither `NULL` nor does it denote a valid address.

Enhancement Orbix enhancement.

IT_ThreadBody Class

IT_ThreadBody is the base class for thread execution methods. To start a thread, derive a class from IT_ThreadBody, add any data members needed by the thread, and provide a run() method which does the thread's work. Then use an [IT_ThreadFactory](#) object to start a thread that will execute the run() method of your IT_ThreadBody object.

If a derived IT_ThreadBody contains data, then it must not be destroyed while threads are using it. One way to manage this is to allocate the IT_ThreadBody with the new() operator and have the IT_ThreadBody delete itself at the end of run(). Also, if multiple threads run the same IT_ThreadBody, it is up to you to provide synchronization on shared data.

```
class IT_ThreadBody {
public:
    virtual ~IT\_ThreadBody() {}

    virtual void* run() =0;
};
```

IT_ThreadBody::~~IT_ThreadBody() Destructor

```
virtual ~IT\_ThreadBody();
```

The destructor for the IT_ThreadBody object.

IT_ThreadBody::run()

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

Does the work and returns a status, which is typically NULL or the address of a static object.

Exceptions

On platforms that support exceptions, if run() throws an exception while used by an attached thread, this thread's exit status will be [IT_Thread::thread_failed](#).

IT_ThreadFactory Class

An `IT_ThreadFactory` object starts threads that share some common properties. You can derive your own class from `IT_ThreadFactory` to control other aspect of thread creation, such as the exact method used to create or start the thread, or the priority of threads when they are created.

```
class IT_ThreadFactory {
public:
    enum DetachState { Detached, Attached };

    IT\_ThreadFactory(
        DetachState detachState,
        size_t stackSize =0
    );

    virtual ~IT\_ThreadFactory();

    virtual IT_Thread start(
        IT_ThreadBody& body,
        IT_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    static IT_Thread smf\_start(
        IT_ThreadBody& body,
        DetachState detach_state,
        size_t stack_size,
        IT_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

protected:
    ...
};
```

IT_ThreadFactory::DetachState Enumeration

```
enum DetachState { Detached, Attached };
```

A thread can be started in a detached or attached state. If a thread is detached, you cannot join it (retrieve its exit status). If a thread is attached you must join it to tell the operating system to forget about it.

Enhancement Orbix enhancement.

IT_ThreadFactory::IT_ThreadFactory() Constructor

```
IT_ThreadFactory(  
    DetachState detachState,  
    size_t stackSize = 0  
);
```

Constructor for an `IT_ThreadFactory` object.

Parameters

<code>detachState</code>	Specify whether the manufactured threads are <code>Detached</code> or <code>Attached</code> .
<code>stackSize</code>	As an option, you can specify the stack size of your threads (expressed in bytes). A value of 0 (the default) means that the operating system will use a default.

Enhancement Orbix enhancement.

See Also [IT_Thread::join\(\)](#)

IT_ThreadFactory::~~IT_ThreadFactory() Destructor

```
virtual ~IT_ThreadFactory();
```

The destructor for a thread factory object.

Enhancement Orbix enhancement.

IT_ThreadFactory::smf_start()

```
static IT_Thread smf_start(  
    IT\_ThreadBody& body,  
    DetachState detach_state,  
    size_t stack_size,  
    IT\_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

A static member method (smf) that starts a thread without creating a thread factory explicitly. This method is useful for simple examples and prototyping but is not as flexible for robust applications.

Enhancement Orbix enhancement.

See Also [IT_ThreadFactory::start\(\)](#)

IT_ThreadFactory::start()

```
virtual IT\_Thread start(  
    IT\_ThreadBody& body,  
    IT\_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Starts a thread. This method creates an operating system thread that runs the given `body`. The method returns an [IT_Thread](#) object that represents this thread.

Parameters

<code>body</code>	The thread body to run.
<code>eh</code>	A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions An error that can be reported includes [IT_TSRuntime](#).

See Also [IT_Thread](#)
[IT_ThreadBody](#)

IT_TimedCountByNSemaphore Class

This semaphore is a non-negative counter typically used to coordinate access to a set of resources. Several resources can be posted or waited for atomically. For example, if there are 5 resources available, a thread that asks for 7 resources would wait but another thread that later asks for 3 resources would succeed, taking 3 resources.

```
class IT_TimedCountByNSemaphore {
public:
    enum { infinite_timeout = -1 };
    enum { infinite_size = 0 };

    IT\_TimedCountByNSemaphore(
        size_t initial_count,
        size_t max_size,
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    ~IT\_TimedCountByNSemaphore();

    void post(
        size_t n,
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    void wait(
        size_t n,
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    int wait(
        size_t n,
        long timeout,
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    int trywait(
        size_t n,
```

```
        IT_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

private:
    ...
};
```

IT_TimedCountByNSemaphore::infinite_size Constant

```
enum { infinite_size = 0 };
```

A constant used to indicate an infinite sized semaphore.

See Also [IT_TimedCountByNSemaphore::wait\(\)](#)

IT_TimedCountByNSemaphore::infinite_timeout Constant

```
enum { infinite_timeout = -1 };
```

A constant used to indicate there is no time-out period for the semaphore.

See Also [IT_TimedCountByNSemaphore::wait\(\)](#)

IT_TimedCountByNSemaphore:: IT_TimedCountByNSemaphore() Constructor

```
IT_TimedCountByNSemaphore(
    size_t initial_count,
    size_t max_size,
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
);
```

Initializes the semaphore with `initial_count` and sets its maximum size to `max_size`.

Enhancement Orbix enhancement.

Exceptions An error that can be reported is [IT_TSRuntime](#).

IT_TimedCountByNSemaphore:: ~IT_TimedCountByNSemaphore() Destructor

```
~IT_TimedCountByNSemaphore();
```

The destructor for the semaphore.

Enhancement Orbix enhancement.

IT_TimedCountByNSemaphore::post()

```
void post(  
    size_t n,  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Posts the number of resources managed.

Parameters

n The number of resources. If the value of **n** plus the previous number of resources is greater than `max_size`, then the number of resources remains unchanged and an [IT_TSLogic](#) error is reported. Calling the method using a value of 0 does nothing.

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSRuntime](#)

[IT_TSLogic](#)

IT_TimedCountByNSemaphore::trywait()

```
int trywait(  
    size_t n,  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Equivalent to a `wait(n, 0, eh)`.

-
- Enhancement** Orbix enhancement.
- Exceptions** An error that can be reported is [IT_TSRuntime](#).
- See Also** [IT_TimedCountByNSemaphore::wait\(\)](#)

IT_TimedCountByNSemaphore::wait()

```
void wait(  
    size_t n,  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Attempts to take a set of resources atomically.

```
int wait(  
    size_t n,  
    long timeout,  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Attempts to take a set of resources (n) atomically. Returns 1 upon success or 0 when the operation times out. Calling wait(0, timeout, eh) returns 1 immediately.

Parameters

- n The number of resources attempted. A value of 0 causes the methods to return immediately.
- timeout The number of milliseconds before the call gives up. You can use the constant [infinite_timeout](#).
- eh A reference to an error handler object.

[IT_Semaphore](#) and [IT_TimedSemaphore](#) can be more efficient than [IT_TimedCountByNSemaphore](#) when resources are posted and waited for one by one.

- Enhancement** Orbix enhancement.
- Exceptions** An error that can be reported is [IT_TSRuntime](#).
- See Also** [IT_Semaphore](#)
[IT_TimedSemaphore](#)

IT_TimedOneshot Class

An `IT_TimedOneshot` class is a synchronization policy typically used to establish a rendezvous between two threads. It can have three states:

- RESET
- SIGNALED
- WAIT

The key class members are as follows:

```
class IT_TimedOneshot {
public:
    enum { infinite\_timeout = -1 };

    IT\_TimedOneshot(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    ~IT\_TimedOneshot();

    void signal(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    void reset(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    void wait(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );
    int wait(
        long timeout,
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    int trywait(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );
};
```

```
...  
};
```

IT_TimedOneshot::infinite_timeout Constant

```
enum { infinite_timeout = -1 };
```

The `IT_TimedOneshot` class includes the symbolic constant `infinite_timeout`. This constant has a value of `-1`.

Enhancement Orbix enhancement.

See Also [IT_TimedOneshot::wait\(\)](#)

IT_TimedOneshot::IT_TimedOneshot() Constructor

```
IT_TimedOneshot(  
    IT_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Initializes the one-shot to the `RESET` state.

Parameters

`eh` A reference to an error handler object.

Enhancement Orbix enhancement.

IT_TimedOneshot::~~IT_TimedOneshot() Destructor

```
~IT_TimedOneshot();
```

Destroys the one-shot object.

Parameters

`eh` A reference to an error handler object.

Enhancement Orbix enhancement.

IT_TimedOneshot::reset()

```
void reset(  
    IT\_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Resets the one-shot object.

- Resetting a one-shot while in the `SIGNALED` state changes its state to `RESET`.
- Resetting a one-shot while in the `RESET` state has no effect.
- Resetting a one-shot in the `WAIT` state is an error. Note that this error is not always detected, even in the checked mode.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

IT_TimedOneshot::signal()

```
void signal(  
    IT\_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Signals the one-shot.

- Signaling a one-shot while in the `RESET` state changes its state to `SIGNALED`.
- Signaling a one-shot while in the `WAIT` state atomically releases the waiting thread and changes the one-shot state to `RESET`.
- Signaling a one-shot while in the `SIGNALED` state is an error.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

IT_TimedOneshot::trywait()

```
int trywait(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Equivalent to a call to `wait(0, eh)`.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

See Also [IT_TimedOneshot::wait\(\)](#)

IT_TimedOneshot::wait()

```
void wait(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);  
  
int wait(  
    long timeout,  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Waits for the one-shot.

- Waiting for a one-shot while in the `RESET` state changes its state to `WAIT`. the second method returns 1 when another thread signals the one-shot within the time-out period. Otherwise it returns 0 and changes the state back to `RESET`.
- Waiting for a one-shot while in the `SIGNALED` state changes its state to `RESET`. The first method returns immediately and the second method returns 1 immediately.
- Waiting for a one-shot while in the `WAIT` state is an error.

Parameters

timeout The number of milliseconds before the call gives up. You can use the constant [infinite_timeout](#).

eh A reference to an error handler object.

Enhancement Orbix enhancement.

See Also [IT_Semaphore](#)
[IT_TimedSemaphore](#)

IT_TimedSemaphore Class

The `IT_TimedSemaphore` object is a counter with a timer for coordinating access to some resources.

```
class IT_TS_API IT_TimedSemaphore
{
public:
    enum { infinite\_timeout = -1 };

    IT\_TimedSemaphore(
        size_t initial_count,
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    ~IT\_TimedSemaphore();

    void post(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    void wait(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );
    int wait(
        long timeout,
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    int trywait(
        IT_TSErrHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );
private:
    ...
};
```

IT_TimedSemaphore::infinite_timeout Constant

```
enum { infinite_timeout = -1 };
```

The `IT_TimedSemaphore` class includes the symbolic constant `infinite_timeout`. This constant has a value of `-1`.

Enhancement Orbix enhancement.

See Also [IT_TimedSemaphore::wait\(\)](#)

IT_TimedSemaphore::IT_TimedSemaphore() Constructor

```
IT_TimedSemaphore(  
    size_t initial_count,  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

A semaphore constructor.

Parameters

`initial_count` Initializes the semaphore's counter with this value.

`eh` A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions An error that can be reported is [IT_TSRuntime](#).

IT_TimedSemaphore::~~IT_TimedSemaphore() Destructor

```
~IT_TimedSemaphore();
```

The destructor.

Enhancement Orbix enhancement.

IT_TimedSemaphore::post()

```
void post(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions An error that can be reported is [IT_TSRuntime](#).

IT_TimedSemaphore::trywait()

```
int trywait(  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Returns 1 if a resource has been obtained, 0 otherwise.

Parameters

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions An error that can be reported is [IT_TSRuntime](#).

IT_TimedSemaphore::wait()

```
void wait()  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);  
  
int wait(  
    long timeout,  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Waits for one resource. The `wait()` method blocks if the semaphore's counter value is 0 and decrements the counter if the counter's value is greater than 0.

Parameters

timeout The number of milliseconds before the call gives up. You can also use the constant [infinite_timeout](#).

eh A reference to an error handler object.

Enhancement Orbix enhancement.

Exceptions Errors that can be reported include:

[IT_TSRuntime](#)

[IT_TSLogic](#)

IT_TSBadAlloc Error Class

When `new()` returns 0 an `IT_TSBadAlloc` exception is reported.

```
class IT_TS_API IT_TSBadAlloc : public IT_TSRuntime
public:
    IT_TSBadAlloc();
    virtual ~IT_TSBadAlloc();
    virtual void raise() const;
};
```

See Also

[IT_TSRuntime](#)
[IT_TSError](#)

IT_TSError Error Class

All errors reported by the TS package are `IT_TSError` objects. The key members of the class are as follows:

```
class IT_TS_API IT_TSError {
public:
    IT\_TSError(
        unsigned long TS_errcode,
        long OS_errno = 0
    );
    IT\_TSError(
        const IT_TSError& other
    );

    virtual ~IT\_TSError();

    unsigned long TS\_error\_code() const;
    long OS\_error\_number() const;
    const char* what() const;
    virtual void raise() const;

protected:
    ...
};
```

See Also

[IT_DefaultTSErrorHandler](#)

IT_TSError::IT_TSError() Constructors

```
IT_TSError(
    unsigned long TS_errcode,
    long OS_errno = 0
);

IT_TSError(
    const IT_TSError& other
);
```

Constructs an error with this TS error code and optionally an error number given by the operating system. The second method is the copy constructor.

Enhancement Orbix enhancement.

IT_TSError::~~IT_TSError() Destructor

```
virtual ~IT_TSError();
```

The destructor.

Enhancement Orbix enhancement.

IT_TSError::OS_error_number()

```
long OS_error_number() const;
```

Returns the operating system error number that represent the error. Returns 0 if the error is not reported by the operating system.

Enhancement Orbix enhancement.

IT_TSError::raise()

```
virtual void raise() const;
```

When exceptions are supported, this method throws `*this`, a pointer to this `IT_TSError` object. If exceptions are not supported, it calls `::abort()`.

Enhancement Orbix enhancement.

IT_TSError::TS_error_code()

```
unsigned long TS_error_code() const;
```

Returns the TS error code that represents the error.

Enhancement Orbix enhancement.

IT_TSError::what()

```
const char* what();
```

Returns a string describing the error. The caller must not de-allocate the returned string.

Enhancement Orbix enhancement.

See Also [IT_TSLogic](#)
[IT_TSRuntime](#)
[IT_TSBadAlloc](#)

IT_TSErrHandler Class

The last parameter of almost every TS method is a reference to an object of the class `IT_TSErrHandler`. When a TS method detects an error, it creates an [IT_TSErr](#) object and passes it to [IT_TSErrHandler::handle\(\)](#).

```
class IT_TS_API IT_TSErrHandler {
public:
    virtual ~IT\_TSErrHandler\(\);

    virtual void handle(
        const IT_TSErr& thisError
    ) = 0;
};
```

See Also [IT_DefaultTSErrHandler](#)

IT_TSErrHandler::handle()

```
virtual void handle(
    const IT_TSErr& thisError
) = 0;
```

Handles the given TS error.

Parameters

`thisError` The error raised.

Enhancement Orbix enhancement.

IT_TSErrHandler::~~IT_TSErrHandler() Destructor

```
virtual ~IT\_TSErrHandler\(\);
```

The destructor for the error handler object.

Enhancement Orbix enhancement.

IT_TSLogic Error Class

An `IT_TSLogic` error signals an error in the application's logic, for example when a thread attempts to join itself.

```
class IT_TS_API IT_TSLogic : public IT_TSError {
public:
    IT_TSLogic(
        unsigned long code,
        long fromOS =0
    );

    virtual ~IT_TSLogic();

    virtual void raise() const;

private:
    // ...
};
```

See Also

[IT_TSError](#)
[IT_TSRuntime](#)

IT_TSRuntime Error Class

An `IT_TSRuntime` error is an error detected by the operating system or by the underlying thread package.

```
class IT_TS_API IT_TSRuntime : public IT_TSError {
public:
    IT_TSRuntime(
        unsigned long code,
        long fromOS =0
    );

    virtual ~IT_TSRuntime();

    virtual void raise() const;

private:
    ...
};
```

See Also

[IT_TSError](#)
[IT_TSRuntime](#)

IT_TSVoidStar Class

An `IT_TSVoidStar` object is a data entry point that can be shared by multiple threads. Each thread can use this entry point to get and set a `void*` pointer that refers to thread-specific (private) data.

```
class IT_TSVoidStar {
public:
    IT\_TSVoidStar(
        void (*destructor)(void*) df,
        IT_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

    ~IT\_TSVoidStar();

    void* get(
        IT_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    ) const;

    void set(
        void* newValue,
        IT_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
    );

private:
    ...
};
```

`IT_TSVoidStar::IT_TSVoidStar()` Constructor

```
IT_TSVoidStar(
    void (*destructor)(void*) df,
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER
);
```

Constructs an `IT_TSVoidStar` object. Initially, all thread-specific pointers are `NULL`.

Parameters

- df You can optionally associate a non-NULL destructor method with an `IT_TSVoidStar` object. Before exiting, a thread will call this destructor with its specific pointer value only when its specific pointer value is not `NULL`.
- eh A reference to an error handler object.

On some platforms, when threads are not started using an [IT_ThreadBody](#), the application might have to call explicitly [IT_CurrentThread::cleanup\(\)](#) upon thread exit to perform this cleanup.

Enhancement Orbix enhancement.

Exceptions An error that can be reported is [IT_TSRuntime](#).

See Also [IT_TSVoidStar::~~IT_TSVoidStar\(\)](#)
[IT_CurrentThread::cleanup\(\)](#)

IT_TSVoidStar::~~IT_TSVoidStar() Destructor

```
~IT_TSVoidStar();
```

The destructor for an `IT_TSVoidStar` object.

If a non-NULL destructor method is associated with this `IT_TSVoidStar` object (by way of the `IT_TSVoidStar()` constructor), and the thread-specific value of this object is not `NULL`, the non-NULL destructor method is called with the thread-specific value.

WARNING: If the `IT_TSVoidStar` object has a non-NULL destructor, do not destroy the object while any other threads have a non-NULL thread-specific pointer. This is because on some platforms, a newly allocated `IT_TSVoidStar` object might *reincarnate* the destroyed `IT_TSVoidStar` object and its thread-specific values. This can lead to unexpected results.

Enhancement Orbix enhancement.

See Also [IT_TSVoidStar::IT_TSVoidStar\(\)](#)

IT_TSVoidStar::get()

```
void* get(  
    IT_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
) const;
```

Gets the pointer associated with the calling thread. Returns `NULL` when the calling thread did not explicitly set this value.

Exceptions An error that can be reported is [IT_TSRuntime](#).

Enhancement Orbix enhancement.

IT_TSVoidStar::set()

```
void set(  
    void* newValue,  
    IT\_TSErrorHandler& eh IT_TS_DEFAULT_ERROR_HANDLER  
);
```

Sets the pointer associated with the calling thread to `newValue`.

Exceptions An error that can be reported is [IT_TSRuntime](#).

Enhancement Orbix enhancement.

Index

B

broadcast() 7

C

cancel() 23, 41
cleanup() 11
close() 17

D

DetachState enumeration 60

G

get() 91

H

handle() 15, 83

I

id() 11, 52
infinite_size constant 64
infinite_timeout constant 64, 68, 74
is_locked() 23
is_main_thread() 12
is_null() 52
IT_Condition class 7
~IT_Condition() 8
IT_Condition() constructor 8
IT_CurrentThread class 11
IT_DefaultTSErrorHandler class 15
~IT_DefaultTSErrorHandler() 15
IT_Gateway class 17
~IT_Gateway() 18
IT_Gateway() constructor 18
IT_Locker Template class 21
~IT_Locker() 25
IT_Locker() 23
IT_Mutex class 27
~IT_Mutex() 28
IT_Mutex() constructor 28
IT_PODMutex Structure 31
IT_RecursiveMutex class 35
~IT_RecursiveMutex() 36

IT_RecursiveMutex() constructor 36
IT_RecursiveMutexLocker class 39
~IT_RecursiveMutexLocker() 42
IT_RecursiveMutexLocker() constructors 41
IT_Semaphore class 45
~IT_Semaphore() 46
IT_Semaphore() constructor 45
IT_TerminationHandler class 49
IT_Thread class 51
~IT_Thread() 53
IT_Thread() constructors 52
IT_ThreadBody class 57
~IT_ThreadBody() 57
IT_ThreadFactory class 59
~IT_ThreadFactory() 60
IT_ThreadFactory() constructor 60
IT_TimedCountByNSemaphore class 63
~IT_TimedCountByNSemaphore() 65
IT_TimedCountByNSemaphore() constructor 64
IT_TimedOneshot class 67
~IT_TimedOneshot() 68
IT_TimedOneshot() constructor 68
IT_TimedSemaphore class 73
~IT_TimedSemaphore() 74
IT_TimedSemaphore() constructor 74
IT_TSBadAlloc error class 77
IT_TSError error class 79
~IT_TSError() 80
IT_TSError() constructors 79
IT_TSErrorHandler class 83
~IT_TSErrorHandler() 83
IT_TSLogic error class 85
IT_TSRuntime error class 87
IT_TSVoidStar class 89
~IT_TSVoidStar() 90
IT_TSVoidStar() constructor 89

J

join() 53

L

lock() 25, 28, 31, 36, 43
lock_count() 43

Index

M

m_index data type 32
mutex() 25, 43

O

open() 18
operator!=(()) 54
operator=() 53
operator==(()) 54
OS_error_number() 80

P

post() 46, 65, 74

R

raise() 80
reset() 69
run() 57

S

self() 12
set() 91
signal() 8, 69
sleep() 12
smf_start() 61
start() 61
synchronization toolkit 1

T

thread
 errors and exceptions 4
 execution modes 2
 Inlined classes 3
 setting an execution mode 3
 Timeouts 2
 wrapper classes 3
thread_failed constant 55
threading toolkit 1
trylock() 26, 29, 32, 37, 43
trywait() 46, 65, 70, 75
TS, threading and synchronization 1
TS_error_code() 80

U

unlock() 29, 33, 37, 44

W

wait() 9, 19, 47, 66, 70, 75
what() 81

Y

yield() 13