Testing Multithreaded Applications

In this section:

Prerequisites

To enable testing multi-threaded applications, add the following option to Build Settings' Compiler Options (described in Setting Project and File Options):

```
"-DCPPTEST_THREADS=1"
```

This option will activate multi-thread support in the C++test's Runtime library, as well as disable use of Safe Stubs for thread-related routines.

Considerations

- C++test's test executable must be linked with the appropriate runtime library (i.e. the multithreaded version of C Runtime Library for Visual C++ compilers or "pthread" library on UNIX).
- C++test's Runtime does not control threads' life-time - we recommended you implement test cases so that all threads are terminated when the test case completes.

Known Problems and Limitations

Known problems and limitations:

- If multi-thread support is enabled (using -DCPPTEST_THREADS=1)—but no appropriate library is used during linking (or a multi- and single-threaded C Runtime is mixed)—then the test executable may produce corrupted test results, or may terminate unexpectedly.
- Unexpected behavior in a non-main thread (signal, unhandled exception, time-out) will cause the test executable to terminate.

Building the C++test Runtime Library

C++test's default Runtime Library has built-in support for testing multithreaded applications. No additional action is required unless you are building a custom runtime library with multithreaded support (e.g., for embedded testing). If you need to build a custom Runtime Library with the multi-thread support, add `"- DCPPTEST_THREADS_ENABLED=1"` to the compiler command line.

Supported Thread APIs

The implementation of Runtime Library can use the following types of thread APIs:

- Windows Threads
- POSIX Threads
- VxWorks 6.x

Using Other Thread APIs

If you need to use another thread API (or a custom one), the following types and routines must be implemented.

Note that the CppTestThread.c file in the C++test Runtime sources contains definitions of thread support routines; it can be used as an example or as a base for changes.

TIs - Thread Local Storage

```
typedef IMPLEMENTATION_DEPENDENT_TYPE CppTestThreadKey;
/**
* Creates key for thread local storage data * Returns 0 on success
*/
extern int localThreadKeyCreate(CppTestThreadKey* key);
/**
* Deletes key for thread local storage data * Returns 0 on success
*/
extern int localThreadKeyDelete(CppTestThreadKey key);
/**
* Returns a thread specific value associate with a key */
extern void* localThreadGetSpecific(CppTestThreadKey key);
/**
* Associate a thread specific value with a key * Returns 0 on success
*/
extern int localThreadSetSpecific(CppTestThreadKey key, void* value);
```

Mutex

Note: C++test's Runtime assumes that a mutex can be statically initialized.

```
typedef IMPLEMENTATION_DEPENDENT_TYPE CppTestThreadMutex;
#define CPPTEST_THREADS_MUTEX_STATIC_INIT <IMPLEMENTATION_DEPENDENT_STATIC_INITIALIZER>
/**
* Initializes mutex
 * @return 0 on success
 */
extern int localThreadMutexInit(CppTestThreadMutex* mutex);
/**
* Destroy and release resources used by mutex * @return 0 on success
*/
extern int localThreadMutexDestroy(CppTestThreadMutex* mutex);
/**
* Lock mutext and return when calling thread becames is owner of it. * @return 0 on success
*/
extern int localThreadMutexLock(CppTestThreadMutex* mutex);
/**
 * Releases mutex owned by calling thread.
 * @return 0 on success
 */
extern int localThreadMutexUnlock(CppTestThreadMutex* mutex);
```

Miscellaneous

```
/**
 * Exits calling thread
 * (never returns)
 */
extern void localThreadExit();
/**
* @return non-zero if threads already finished execution */
extern int localThreadFinished(CppTestThread* thread);
/**
* @return non-zero if threads are supported in current build
* (proper macros, libraries, compiler options were used).
*/
extern int localThreadsSupported(void);
/**
 * Initializes given thread structure
 */
extern void localThreadInit(CppTestThread* thread);
```