TIBCO SmartSockets™

Utilities

Software Release 6.8
July 2006

Important Information

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE *TIBCO SMARTSOCKETS INSTALLATION GUIDE*). USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document contains confidential information that is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIB, TIBCO, Information Bus, The Power of Now, TIBCO Adapter, RTclient, RTserver, RTworks, SmartSockets, and Talarian are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and / or other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND / OR CHANGES IN THE PRODUCT(S) AND / OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

Copyright © 1991–2006 TIBCO Software Inc. ALL RIGHTS RESERVED.

TIBCO Software Inc. Confidential Information

Contents

Preface xi

Related Documentation xii
TIBCO Product Documentation xii
Using the Online Documentation xii
Conventions Used in This Manual xiii
Typeface Conventions xiii
Notational Conventions. xiv
Identifiers xiv
Case xv
How to Contact TIBCO Support. xvi

Chapter 1 Introduction 1

Include Files. 2
Architecture-Independent Data Types 3
Functions 4
Callback Functions 4
Traversals 6
Lookup Functions 6
Access Functions 7
Compiling and Linking 7
Conventions. 8

Chapter 2 Utility Functions and Macros 9

Naming Conventions 10
Data Structures 11
Threads 11
Cross-Platform Thread Support 12
Callbacks 16
Time Converters 18
Hash Tables 19
Pointer Arrays 21
Options 23
Time Structure 24

Time Change Callbacks 25
Number Formats 26
File Handling 27
XML Utilities 28
Error Handling 28

Chapter 3 T_ASSERT - T_STRDUP 29

T_ASSERT 30
T_CALLOC 32
T_ENTRY 33
T_FREE 34
T_MALLOC 35
T_REALLOC 36
T_STRDUP 38

Chapter 4 TutBufAllocAligned - TutBufSetSize 39

TutBufAllocAligned 40
TutBufAppendBuf 41
TutBufAppendRange 42
TutBufCloneRange 43
TutBufCreate 44
TutBufCreateStatic 45
TutBufDeleteRange 46
TutBufDestroy 47
TutBufGetSize 48
TutBufGrow 49
TutBufNextAligned 50
TutBufReset 51
TutBufRewind 52
TutBufSetSize 53

Chapter 5 TutCalloc - TutCondWakeAll 55

TutCalloc 56
TutCbDestroy 57
TutCbGetArgument 58
TutCbGetFunction 59
TutCbGetPriority 61
TutCbSetArgument 62
TutCbSetFunction 64
TutCbSetPriority 66
TutCommandParseFile 68
TutCommandParseStr 69
TutCommandParseTypedStr 70
TutCondCreate 71
TutCondDestroy 73
TutCondWait 74
TutCondWakeAll 76

Chapter 6 TutErrNumGet - TutFree 79

TutErrNumGet 80
TutErrNumGetC 81
TutErrNumGetOs 82
TutErrNumGetSocket 83
TutErrNumSet 84
TutErrNumToStr 85
TutErrStrCreate 86
TutErrStrGet 87
TutExit 88
TutFileTraverse 90
TutFOpen 91
TutFree 93

Chapter 7 TutGetCurrentTime - TutHashTraverse 95

TutGetCurrentTime 97
TutGetCurrentTimeStruct 98
TutGetenv 99
TutGetIntFormat 100
TutGetNodeName 101
TutGetOutFlushFunc 102
TutGetOutputFunc 103
TutGetRealFormat 104
TutGetSocketDir 105
TutGetVersionName 106
TutGetVersionNumber 107
TutGetWallTime 108
TutGetWallTimeStruct 109
TutHashBucketGetValue 110
TutHashBucketSetValue 111
TutHashCreate 112
TutHashCreateInt 115
TutHashCreatePtr 117
TutHashCreateStr 119
TutHashCreateStri 120
TutHashDelete 121
TutHashDestroy 123
TutHashDump 125
TutHashGetSize 127
TutHashInsert 128
TutHashLookup 130
TutHashLookupBucket 131
TutHashReplace 132
TutHashSort 135
TutHashTraverse 137

Chapter 8 TutMalloc - TutOutFlush. 139

TutMalloc 141
TutMutexCreate 142
TutMutexCreateFast 144
TutMutexDestroy 147
TutMutexLock 148
TutMutexLockFast 149
TutMutexUnlock 150
TutOptionChangeCbCreate 151
TutOptionChangeCbLookup 153
TutOptionCreate 155
TutOptionDestroy 157
TutOptionGetBool 158
TutOptionGetEnum 159
TutOptionGetEnumList 160
TutOptionGetKnown 162
TutOptionGetName 163
TutOptionGetNum 164
TutOptionGetReadOnly 165
TutOptionGetRequired 166
TutOptionGetStr 167
TutOptionGetStrList 168
TutOptionGetType 170
TutOptionGetVerifyFunc 172
TutOptionLegValAdd 174
TutOptionLookup 175
TutOptionNamedLookup 176
TutOptionSetBool 177
TutOptionSetEnum 178
TutOptionSetEnumList 179
TutOptionSetNum 181
TutOptionSetReadOnly 182
TutOptionSetRequired 183
TutOptionSetStr 184
TutOptionSetStrList 185
TutOptionSetUnknown 187

TutOptionSetVerifyFunc 188
TutOut 190
TutOutFlush 191

Chapter 9 TutPerror - TutRwMutexWriteLocked 193

TutPerror 195
TutProcInfo 196
TutPtrAryAppend 197
TutPtrAryChangeCbCreate 198
TutPtrAryChangeCbLookup 200
TutPtrAryClear 201
TutPtrAryCreate 203
TutPtrAryDestroy 204
TutPtrAryGetItemAtIndex 206
TutPtrAryGetItemCount 207
TutPtrAryGetItemIndex 208
TutPtrAryInsertAfter 210
TutPtrAryInsertBefore 212
TutPtrAryInsertItemAtIndex 214
TutPtrAryItemExists 216
TutPtrAryRemove 217
TutPtrAryRemoveAll 219
TutPtrAryReplaceItemAtIndex 221
TutPtrArySort 223
TutPtrArySwapItems 225
TutPutenv 226
TutRealloc 227
TutRealToStr 228
TutRwMutexCreate 230
TutRwMutexDestroy 232
TutRwMutexGetReadQuota 233
TutRwMutexReadLock 234
TutRwMutexReadLocked 235
TutRwMutexSetReadQuota 236
TutRwMutexUnlock 237
TutRwMutexUpgradeLock 238

TutRwMutexWriteLock 240
TutRwMutexWriteLocked 242

Chapter 10 TutSetCurrentTime - TutXmlSetStr 243

TutSetCurrentTime 245
TutSetCurrentTimeStruct 246
TutSetFileCloseFunc 247
TutSetFileOpenFunc 249
TutSetFileReadFunc 250
TutSetOutFlushFunc 251
TutSetOutputFunc 252
TutSleep 253
TutSocketAccept 254
TutSocketCheck 255
TutSocketCreateClientLocal 257
TutSocketCreateClientTcp 259
TutSocketCreateServerLocal 262
TutSocketCreateServerTcp 265
TutSocketGetBlockMode 268
TutSocketRecvAll 270
TutSocketRecvTimeout 272
TutSocketSetBlockMode 274
TutStrDup 275
TutStrLwr 276
TutStrUpr 277
TutSystem 278
TutThreadCreate 279
TutThreadCreateVa 281
TutThreadDetach 283
TutThreadEqual 285
TutThreadExit 286
TutThreadSelf 287
TutThreadWait 288
TutTimeChangeCbCreate 290
TutTimeChangeCbLookup 291
TutTimeCvtCreate 292

TutTimeCvtDestroy . . . 296
TutTimeCvtLookup . . . 297
TutTimeCvtNumToStr . . . 298
TutTimeCvtStrToNum . . . 299
TutTimeCvtTraverse . . . 300
TutTimeNumToStr . . . 302
TutTimeStrToNum . . . 303
TutTsdGetValue . . . 304
TutTsdKeyCreate . . . 305
TutTsdKeyDestroy . . . 307
TutTsdSetValue . . . 308
TutWarning . . . 309
TutWinSetOutputListBox . . . 310
TutXmlClone . . . 312
TutXmlCreate . . . 313
TutXmlCreateStatic . . . 314
TutXmlDestroy . . . 315
TutXmlGetStr . . . 316
TutXmlSetStr . . . 317

Chapter 11 Shell Scripts . . . 319

Shell Script Summary . . . 320
Shell Script Descriptions . . . 321
rtinit.com . . . 322
rtinit.csh . . . 323
rtinit.sh . . . 325
rtlic . . . 327
rtlink . . . 328
rtmon . . . 331
rtmongdi . . . 333
rtmove . . . 335
rtserver . . . 337
rtserver64 . . . 344

Index . . . 345

Preface

TIBCO SmartSockets is a message-oriented middleware product that enables programs to communicate quickly, reliably, and securely across:

- local area networks (LANs)
- wide area networks (WANs)
- the Internet

TIBCO SmartSockets takes care of network interfaces, guarantees delivery of messages, handles communications protocols, and directs recovery after system or network problems. This enables you to focus on higher-level requirements rather than the underlying complexities of the network.

This reference provides the detailed information you need to understand and use the TIBCO SmartSockets utilities, macros, and shell scripts. Before using this reference, it is helpful to read the *TIBCO SmartSockets Tutorial* and work through the lessons in that book. The *TIBCO SmartSockets Tutorial* begins at a more basic level and is an excellent introduction to SmartSockets.

For an overview of the new features, changes, and enhancements in this Version 6.8 release, see the *TIBCO SmartSockets Installation Guide.*

Topics

- *Related Documentation, page xii*
- *Conventions Used in This Manual, page xiii*
- *How to Contact TIBCO Support, page xvi*

Related Documentation

This section lists documentation resources you may find useful.

TIBCO Product Documentation

The following documents form the TIBCO SmartSockets documentation set:

- *TIBCO SmartSockets API Quick Reference*
- *TIBCO SmartSockets Application Programming Interface*
- *TIBCO SmartSockets C++ User's Guide*
- *TIBCO SmartSockets cxxipc Class Library*
- *TIBCO SmartSockets Installation Guide*
- *TIBCO SmartSockets Java Library User's Guide and Tutorial*
- *TIBCO SmartSockets .NET User's Guide and Tutorial*
- *TIBCO SmartSockets Tutorial*
- *TIBCO SmartSockets User's Guide*
- *TIBCO SmartSockets Utilities*
- *TIBCO SmartSockets C++ and Java Class Libraries*

  C++ class library and Java application programming interface (API) reference materials are available in HTML format only. Access the references through the TIBCO HTML documentation interface.

Using the Online Documentation

The SmartSockets documentation files are available for you to download separately, or you can request a copy of the TIBCO Documentation CD.

Conventions Used in This Manual

This manual uses the following conventions.

Typeface Conventions

This manual uses the following typeface conventions

Example	Use
`monospace`	This monospace font is used for program output and code example listing and for file names, commands, configuration file parameters, and literal programming elements in running text.
`monospace bold`	This bold monospace font indicates characters in a command line that you must type exactly as shown. This font is also used for emphasis in code examples.
Italic	Italic text is used as follows: • In code examples, file names etc., for text that should be replaced with an actual value. For example: "Select *install-dir* / `runexample.bat`." • For document titles. • For emphasis.
Bold	Bold text indicates actions you take when using a GUI, for example, click **OK**, or choose **Edit** from the menu. It is intended to help you skim through procedures when you are familiar with them and just want a reminder. Submenus and options of a menu item are indicated with an angle bracket, for example, **Menu > Submenu.**
	Warning. The accompanying text describes a condition that severely affects the functioning of the software.
	Note. Be sure you read the accompanying text for important information.
	Tip. The accompanying text may be especially helpful.

Notational Conventions

The notational conventions in the table below are used for describing command syntax. When used in this context, do not type the brackets listed in the table as part of a command line.

Notation	Description	Use
[ ]	Brackets	Used to enclose an optional item in the command syntax.
< >	Angle Brackets	Used to enclose a name (usually in *Italic*) that represents an argument for which you substitute a value when you use the command. This convention is not used for XML or HTML examples or other situations where the angle brackets are part of the code.
{ }	Curly Brackets	Used to enclose two or more items among which you can choose only one at a time. Vertical bars (|) separate the choices within the curly brackets.
...	Ellipsis	Indicates that you can repeat an item any number of times in the command line.

Identifiers

The term identifier is used to refer to a valid character string that names entities created in a SmartSockets application. The string starts with an underscore (_) or alphabetic character and is followed by zero or more letters, digits, percent signs (%), or underscores. No other special characters are valid. The maximum length of the string is 63 characters. Identifiers are not case-sensitive.

These are examples of valid identifiers:

```
EPS
battery_11
K11
__
_all
```

These are invalid identifiers:

```
20
battery-11
@com
$amount
```

Case

Function names are case-sensitive, and must use the mixed-case format you see in the text. For example, TipcMsgCreate, TipcSrvStop, and TipcMonClientMsgTrafficPoll are SmartSockets functions and must use the case as shown.

Monitoring messages are also case-sensitive, and should be all upper case, such as T_MT_MON_SERVER_NAMES_POLL_CALL. This makes it easy to distinguish them from option or function names.

Although option names are not case-sensitive, they are usually presented in text with mixed case, to help distinguish them from commands or other items. For example, Server_Names, Unique_Subject, and Project are all SmartSockets options.

Identifiers used with the products in the SmartSockets family are not case-sensitive. For example, the identifiers `thermal` and `THERMAL` are equivalent in all processes.

In UNIX, shell commands and filenames are case-sensitive, though they might not be in other operating systems, such as Windows. To make it easier to port applications between operating systems, always specify filenames in lower case.

How to Contact TIBCO Support

For comments or problems with this manual or the software it addresses, please contact TIBCO Support as follows.

- For an overview of TIBCO Support, and information about getting started with TIBCO Support, visit this site:

  http://www.tibco.com/services/support

- If you already have a valid maintenance or support contract, visit this site:

  http://support.tibco.com

  Entry to this site requires a user name and password. If you do not have a user name, you can request one.

Chapter 1 Introduction

This chapter introduces the data types, functions and macros that are available in the TIBCO SmartSockets utilities library. The TIBCO SmartSockets API and shell scripts are documented in these places:

- SmartSockets utility functions and macros are described in Chapter 3, T_ASSERT - T_STRDUP through Chapter 10, TutSetCurrentTime - TutXmlSetStr
- SmartSockets shell scripts are described in Chapter 11, Shell Scripts
- SmartSockets interprocess communication APIs for C are described in the *TIBCO SmartSockets Application Programming Interface* reference
- SmartSockets APIs for C++ are described in the *TIBCO SmartSockets C++ User's Guide*
- SmartSockets APIs for Java are described in the online Java API reference in javadoc format

Topics

- *Include Files, page 2*
- *Architecture-Independent Data Types, page 3*
- *Functions, page 4*
- *Compiling and Linking, page 7*
- *Conventions, page 8*

Include Files

Several important header files are needed to compile and link with SmartSockets. These files are normally included in C/C++ code with the `#include` C/C++ preprocessor directive. The SmartSockets header files are in these directories:

UNIX:

```
$RTHOME/include/$RTARCH/rtworks
```

OpenVMS:

```
RTHOME:[INCLUDE.RTWORKS]
```

Windows:

```
%RTHOME%\include\rtworks
```

Header files should be included with the angle-bracket syntax. For example, when using C, this inclusion line is used:

```
#include <rtworks/ipc.h>
```

Use of the angle-bracket syntax is preferred over double-quotes, such as:

```
#include "ipc.h"
#include "pathname/ipc.h"
```

On UNIX, using angle brackets requires using an argument to the `C++` command:

```
$ CC -I$RTHOME/include/$RTARCH ...
```

On OpenVMS, the logical `RTWORKS` directory is `RTHOME:[INCLUDE.RTWORKS]` so that DEC C++ uses that directory for `<rtworks/ipc.h>`.

On Windows, add the SmartSockets include directory `%RTHOME%\include\rtworks` to the list of Visual C++ Developer Studio project include directories. There are several example Visual C++ makefiles in `%RTHOME%\examples\smrtsock\manual` that you can use as a starting point for your application executables.

Architecture-Independent Data Types

Instead of using native C/C++ scalar data types, function parameters and return values in the SmartSockets API use architecture-independent SmartSockets data types listed in the table. These data types make the SmartSockets API more portable and consistent across different 32-bit and 64-bit architectures.

SmartSockets Data Type	Description	Example 32-bit C Data Type	Example 64-bit C Data Type
T_STR	character string	`char *`	`char *`
T_PTR	generic pointer	`void *`	`void *`
T_BOOL	boolean	`int`	`int`
T_CHAR	character	`char`	`char`
T_UCHAR	unsigned character	`unsigned char`	`unsigned char`
T_INT1	one-byte integer	`signed char`	`signed char`
T_UINT1	unsigned one-byte integer	`unsigned char`	`unsigned char`
T_INT2	two-byte integer	`short`	`short`
T_UINT2	unsigned two-byte integer	`unsigned short`	`unsigned short`
T_INT4	four-byte integer	`long`	`int`
T_UINT4	unsigned four-byte integer	`unsigned long`	`unsigned int`
T_INT8	eight-byte integer	`long long` `__int64` (Windows)	`long` `__int64` (Windows)
T_REAL4	four-byte real	`float`	`float`
T_REAL8	eight-byte real	`double`	`double`
T_REAL16	sixteen-byte real	`char [16]`	`long double`

Functions

The API functions can be broadly classified into these types.

- Callback functions — execute a function under specified conditions
- Traversal functions — traverse or iterate over a list of items, invoking a processing function on each item
- Lookup functions — retrieve pointers to SmartSockets data structures
- Access functions — retrieve and set the values of fields within SmartSockets data structures

These function types are described in more detail in these sections:

Callback Functions

Many of the SmartSockets processes make use of callbacks to allow visibility into the internal processing. A callback is a mechanism that allows you to monitor a type of event. The callback can be associated with a certain object or with all objects of a certain type. Callbacks that are associated with all objects of a given type are called global callbacks. When the event occurs, the specified callback function is called to notify you that the event happened.

This section is a general introduction to callbacks. See Chapter 2, Utility Functions and Macros for a more detailed discussion of callbacks.

All callbacks have some common characteristics, including:

- Callback functions do not return a value (the functions are of type `void`). Callback functions are not used to affect the processing of a process. They are only used to receive notification of some event.
- Callbacks have a priority associated with them that allows order to be specified. The priority specifies the order in which callbacks are called. Higher-priority callbacks are called before lower-priority callbacks. If two callbacks have the same priority, their relative calling order is undetermined.
- Callbacks are uniquely identified by function and argument within a given list. This means that two callbacks cannot be registered that use the same callback function and callback argument within the same list of callbacks.
- All callback function definitions and prototypes must be declared with the macro T_ENTRY for cross-platform portability.

- All callback functions take exactly three arguments. The first is the object to which the callback is related, the second points to a type specific callback data structure, and the third is a user-specified argument that is specified when the callback is created. Depending on the callback type, the first argument may not be used. Check the documentation for the callback being used.

Callbacks are identified by a combination of the callback function and the callback argument. These callbacks are kept in a list that is associated with a particular data structure. All callbacks must be unique within a given list in which they are defined. Subsequent attempts to add the same callback to a given list are ignored.

Global Versus Object-Specific Callbacks

Some types of callbacks can be created for any event of a given type, for example, view-display callbacks. These are called global callbacks. Callbacks can be created specifically (for a specific view) or globally (for all views) such that whenever any view is displayed, the specified callback is called. This is accomplished by specifying a NULL value for the first argument to the callback create function. When both global and specific callbacks are permitted, the object-specific callbacks are merged with the global callback functions, and are called in priority order.

Destruction Management

To streamline SmartSockets processing, callbacks should be destroyed when they are no longer needed. This applies especially for global callbacks, which are invoked for every occurrence of a given event.

Traversals

Another important class of functions in SmartSockets is the traversal functions. These functions traverse or iterate over a list of items, invoking a predefined processing function on each item. An example of this is the function TutHashTraverse. This function calls a specified processing function for each object that exists in the specified hash table. This is useful for dealing with lists of things of indeterminate length.

The function called by the traversal is called with a fixed set of arguments that is unique to each type of traversal. When using these functions, you should be careful to ensure that arguments of the called function meet the specification required for that type of traversal. You must also specify a traversal argument that is passed to the called function, along with the arguments specific to the list being traversed. The traversal argument is not used by the SmartSockets processes and is simply passed on. If a traversal argument is not required, `NULL` should be specified for the traversal argument.

Traversals continue until all the items in the list have been traversed or (typically) until the called function returns a non-null value. The value thus returned is returned to the caller. For this reason, traversal functions generally return a `void` pointer, as do functions that induce a traversal. This may also require casting the return from the function that induced the traversal.

During a traversal, do not add or delete elements from the list being traversed.

Lookup Functions

There are a number of functions that perform a lookup to retrieve a pointer to a SmartSockets data structure; for example, TipcMtLookup. Typically, these functions take a name as an argument and return a pointer to the corresponding data structure. Lookup functions must search a list or table of some kind, such as a hash table or linked list. These functions may take a significant amount of time, depending on the type of table being searched and the length of the table or list. It may be more efficient for callers to store values returned from these functions for reuse instead of looking them up every time they are required. In general, lookup functions are not case sensitive.

Access Functions

Access functions are used to obtain fields within SmartSockets data structures, such as TipcMtGetName, and return the name of a specified T_IPC_MT structure. These functions provide public access to SmartSockets data structures and ensure future compatibility with SmartSockets. They are written to be very quick. There is error checking for null pointers (but not invalid pointers) to keep the functions as robust as possible. If there is a problem with the arguments passed to the function, the most likely result is a segmentation fault on UNIX, access violation on OpenVMS, or an application error on Windows.

Access functions are marked as such in their descriptions. Callers can use these functions frequently without causing significant run-time penalties.

The values returned by access functions should be treated as read-only by the calling function. Changing these values produces unpredictable results. For example, the calling function cannot rename a view by getting its name and then writing a new value into the string returned.

Compiling and Linking

Several libraries are always needed when linking with SmartSockets API functions. The `rtlink` UNIX shell script and OpenVMS command procedure is included with SmartSockets to make linking easier.

You should always use `rtlink` instead of linking directly with the libraries. This insulates you from changes to the names and link order of the libraries.

On Windows, add the SmartSockets library directory `%RTHOME%\lib\%RTARCH%` to the list of Visual C++ Developer Studio project library directories. There are several example Visual C++ makefiles in `%RTHOME%\examples\smrtsock\manual` that you can use as a starting point for your application executables.

Conventions

SmartSockets follows a number of conventions for the naming and syntax of functions. These conventions are intended to make the syntax of functions more predictable and consistent.

These are the major naming conventions:

- All SmartSockets global functions, macros, typedefs, structures, unions, enums, and defines begin with the letter T.
- All SmartSockets global functions use mixed case for their names. After the uppercase T follows a two or three letter code indicating which library they are from, such as TipcMsgCreate or TutExit. They never contain underscore (_) characters.
- All SmartSockets macros, typedefs, enums, and defines start with T_ and are in uppercase. These names use underscores for compound words. Examples are T_IPC_CONN, and T_TYPE_NUMERIC.
- All interprocess communication functions start with Tipc.
- All utility functions start with Tut.
- All callback functions have the string Cb in their names; for example, TipcSrvCbCreate.
- All traversal functions have the string Traverse in their names; for example, TutHashTraverse.
- All lookup functions have the string Lookup in their name; for example, TipcConnProcessCbLookup.
- All access functions have the strings Get, if retrieving, or Set, if storing, in their names; for example, TipcGetCurrentType and TipcSetLbMode.
- Functions that create or destroy things (objects, callbacks) have the strings Create or Destroy in their names; for example, TipcSrvDestroy and TipcMsgCreate.

Chapter 2 Utility Functions and Macros

This chapter describes the SmartSockets utility functions and macros, which are useful routines that SmartSockets uses and are documented for your use. In general, these functions and macros can be called from any SmartSockets application.

Topics

- *Naming Conventions, page 10*
- *Data Structures, page 11*
- *Threads, page 11*
- *Callbacks, page 16*
- *Time Converters, page 18*
- *Hash Tables, page 19*
- *Pointer Arrays, page 21*
- *Options, page 23*
- *Time Structure, page 24*
- *Time Change Callbacks, page 25*
- *Number Formats, page 26*
- *File Handling, page 27*
- *XML Utilities, page 28*
- *Error Handling, page 28*

Naming Conventions

The SmartSockets utility functions and macros use these naming conventions:

- All utility functions start with Tut
- All SmartSockets macros start with T_ and the entire macro name is in uppercase
- All utility thread functions start with TutThread
- All utility mutex functions start with TutMutex
- All utility condition variable functions start with TutCond
- All utility read / write mutex functions start with TutRwMutex
- All utility thread-specific data functions start with TutTsd
- All utility error functions start with TutErr
- All utility hash functions start with TutHash
- All utility pointer array functions start with TutPtrAry
- All utility callback functions start with TutCb
- All utility option functions start with TutOption
- All utility time converter functions start with TutTimeCvt
- All utility socket functions start with TutSocket
- All utility buffer functions start with TutBuf
- All XML utilities start with TutXml
- All key utilities start with TutKey

Data Structures

A number of utility data structures are accessible from user-written programs. These structures are defined:

```
struct T_HASH_TABLE_STRUCT T_HASH_TABLE_STRUCT, *T_HASH_TABLE;
struct T_HASH_BUCKET_STRUCT T_HASH_BUCKET_STRUCT, *T_HASH_BUCKET;
struct T_OPTION_STRUCT T_OPTION_STRUCT, *T_OPTION;
struct T_PTR_ARY_STRUCT T_PTR_ARY_STRUCT, *T_PTR_ARY;
struct T_TIME_CVT_STRUCT T_TIME_CVT_STRUCT, *T_TIME_CVT;
struct T_XML_STRUCT T_XML_STRUCT, *T_XML
```

These structure definitions are incomplete in the sense that none of the fields in the structure are defined. This is one way of creating an opaque data type in C. Pointers to these structures cannot be dereferenced to look at the data where they are pointing. These pointers are only manipulated through access functions, which provides binary compatibility that protects you from changes to the underlying data structures from one version of SmartSockets to the next.

Threads

Thread implementations can be classified into two categories: user-level and kernel-level. The characteristic that distinguishes them is the granularity of scheduling provided by the host operating system. User-level threads are not individually scheduled by the operating system. Instead, each process implements its own scheduling algorithms to divide its CPU time among its threads. User-level threads can be difficult to utilize effectively because the most commonly used operating system API calls are synchronous. When a user-level thread blocks in a system call, the entire process is suspended and other threads in the process do not get the opportunity to run.

In contrast, kernel-level threads are (typically) individually scheduled by the operating system. When a kernel-level thread blocks in a system call, other threads in the process still receive their allocations of CPU time from the operating system. This makes it much simpler to write applications which use threads effectively. Many operating systems now incorporate kernel-level threads support.

The POSIX standard P1003.1c (also known as Pthreads) defines a standard API for access to thread support. However, this standard was developed during a period of strong market demand for thread support, and many OS vendors either implemented early drafts of this standard or implemented proprietary interfaces with the intention of later layering the standard interfaces on top of them. To further complicate matters, platforms such as Windows provide only proprietary interfaces even though much of the underlying function is compatible.

Cross-Platform Thread Support

On platforms with kernel-level threads, the utility library is thread-safe. In other words, all utility library functions behave the same in a multi-threaded environment as in a single-threaded environment. Of course, this assumes that the application program avoids sharing individual data structures between threads or synchronizes operations on shared data structures. The utility library does enough internal synchronization to maintain its own integrity, uses thread-safe system calls where appropriate and uses thread-specific data for per-thread values such as the global SmartSockets error number.

In addition, the utility library includes cross-platform support for thread creation, synchronization, and thread-specific data. This common API makes it possible to write both applications and layered libraries that make use of threads and yet remain source-portable across all supported platforms. These functions are layered on top of the native OS support, which means that they are interoperable with direct use of the native system calls.

Multi-threaded applications must explicitly enable internal synchronization within the utility library with a call to TipcInitThreads. This allows single-threaded applications to avoid incurring unnecessary system call overhead. If any of the utility library's thread support functions are used, an application should call TipcInitThreads before any other utility library function. The function TipcInitThreads is described in more detail in the *TIBCO SmartSockets Application Programming Interface* reference.

Creation

The utility library provides the T_THREAD opaque type to serve as a handle to threads created by the support functions. Thread creation requires a thread function that serves as the entry point for the thread in much the same way that the main function serves a program written in C. The thread function is defined:

```
typedef T_PTR (*T_THREAD_FUNC)(T_PTR arg);
```

The T_PTR argument to the function is passed through from the thread creation routine used, thus providing a mechanism for passing in arbitrary arguments. If the thread function returns, the returned T_PTR value becomes the exit code of the thread. Threads are created and manipulated with these functions:

- TutThreadCreate — create a thread
- TutThreadCreateVa — create a thread (va_list version)
- TutThreadDetach — allow a thread to run to completion without requiring the operating system to remember its exit code
- TutThreadEqual — compare two thread handles for equality
- TutThreadExit — exit from a thread
- TutThreadSelf — return the thread handle of the calling thread
- TutThreadWait — wait for another thread to exit and optionally retrieve its exit code

Synchronization

The utility library provides support for three synchronization object types: mutexes, condition variables, and read / write mutexes (represented by the opaque types T_MUTEX, T_COND, and T_RW_MUTEX, respectively). Of the three, mutexes are the most primitive type. They provide the mechanism by which a thread can secure exclusive access to a shared resource using a simple locking metaphor. Only one thread may have a mutex locked at any given instant. The functions for creating and manipulating T_MUTEX objects are:

- TutMutexCreate — create a mutex object
- TutMutexCreateFast — create a high-performance mutex object
- TutMutexDestroy — destroy a mutex object
- TutMutexLock — lock a mutex object
- TutMutexLockFast — lock a high-performance mutex object
- TutMutexUnlock — unlock a mutex object

Condition variables provide the mechanism by which one thread can wait for another thread to complete an activity. Mutexes can only ensure serialization of an activity. T_COND instances are used in conjunction with mutexes, allowing a thread to wait until an arbitrary condition has occurred. The condition variable functions provided by the utility library are:

- TutCondCreate — create a condition variable
- TutCondDestroy — destroy a condition variable
- TutCondWait — cause the calling thread to wait to be awakened by a condition variable
- TutCondWakeAll — wake all of the threads waiting for a condition variable

Read / write mutexes are useful for protecting a shared resource in scenarios where multiple threads need to examine the state of the resource concurrently, but exclusive access must still be available so that the state can be safely changed. T_RW_MUTEX objects rely upon both normal mutexes and condition variables for their implementation. The utility library functions for creating and working with read / write mutexes are:

- TutRwMutexCreate — create a read / write mutex
- TutRwMutexDestroy — destroy a read / write mutex
- TutRwMutexGetReadQuota — get the simultaneous reader quota
- TutRwMutexReadLock — read-lock a read / write mutex
- TutRwMutexReadLocked — test for a read-lock
- TutRwMutexSetReadQuota — set the simultaneous reader quota
- TutRwMutexUnlock — unlock a read / write mutex
- TutRwMutexUpgradeLock — upgrade a read-lock to a write-lock
- TutRwMutexWriteLock — write-lock a read / write mutex
- TutRwMutexWriteLocked — test for a write-lock

Thread-Specific Data

Thread-specific data (TSD) provides a thread-safe substitute for global and file-scope variables. Rather than directly storing a value that is shared by several functions in a global variable, a TSD key (T_TSD_KEY) can be created. The value may then be placed in memory allocated from the heap and referenced by a unique T_PTR value for each thread. Each function that references the value must then use the TSD key to locate the shared value on the heap with the T_PTR value for the currently executing thread. Each TSD key may have an associated cleanup function, defined in this way:

```
void (*T_TSD_KEY_DESTRUCTOR_FUNC)(T_PTR value);
```

Whenever a utility library thread exits, it iterates through all of its TSD values and calls any cleanup functions specified when the keys for those values were created. The functions for working with T_TSD_KEY objects and thread-specific data are:

- TutTsdGetValue — get the per-thread T_PTR value associated with a TSD key
- TutTsdKeyCreate — create a TSD key
- TutTsdKeyDestroy — destroy a TSD key
- TutTsdSetValue — set the per-thread T_PTR value associated with a TSD key

Callbacks

SmartSockets makes use of callbacks to allow you visibility into the internal processing. A callback is a mechanism that allows you to monitor a type of event. The callback is associated with a certain object or with all objects of a certain type. Callbacks associated with all objects of a given type are called global callbacks. When the event occurs, the specified callback function is called to notify you that the event happened.

An example of this type of function is a variable change callback. Such a callback could be used so that the application can monitor the changing of a variable's value, and do other processing required to support the requirements. Callbacks belong to a callback list, which is an ordered set of all the callbacks created for a given event type and object.

There is a generic, abstract type of callbacks called T_CB. This type has a set of properties such as function, priority, argument, and so on. Because it is an abstract type, it is not directly accessible. You can create these abstracts with functions such as TipcSrvSubjectCbCreate. You can manipulate them with the TutCb* access functions such as TutCbSetPriority. The Tipc* functions are documented in the *TIBCO SmartSockets Application Programming Interface* reference, and the Tut* functions are documented in the *TIBCO SmartSockets Utilities* reference.

The specific types of callbacks that are provided vary depending on the process, but all SmartSockets callbacks have certain common characteristics. Each process (RTserver, RTmon, and so on.) provides a set of functions to deal with callbacks specific to that process. Typically, there is a T*processType*CbCreate function and a T*processType*CbLookup function that can be used to create and look up callbacks. Examples of these include TipcConnDefaultCbCreate, TipcConnDefaultCbLookup, TipcSrvSubjectCbCreate, and TipcSrvSubjectCbLookup. All other operations performed on callbacks are done using TutCb* functions.

These are the general common concepts that all SmartSockets callbacks share:

- Callback functions always return void.
- Callbacks have a priority associated with them that allows you to specify order. The priority specifies the order in which callbacks are called. Higher priority callbacks are called before lower priority ones. If two callbacks have the same priority, their relative calling order is undetermined.
- Callbacks are uniquely identified by function and argument within a given list. This means you cannot register two callbacks which use the same callback function and callback argument within the same list of callbacks.

- All callback function definitions and prototypes must be declared with the macro T_ENTRY for cross-platform portability.
- All callback functions take exactly three arguments. The first is the object to which the callback is related, the second points to a callback data structure, and the third is a user-specified argument that is specified when the callback is created. Depending on the callback, the first argument may not be used. Check the documentation for the callback being used. The generic callback function is defined as:

  ```
  typedef void (*T_CB_FUNC)
       (T_CB_OBJ ptr, T_CB_DATA data, T_CB_ARG arg);
  ```

  There is usually a callback data structure specific to the type of callback being used, but not always. There is a generic callback data structure definition provided that is defined as:

  ```
  typedef struct T_CB_DATA_STRUCT {
        T_CB cb;
      } T_CB_DATA_STRUCT, *T_CB_DATA;
  ```

  This data structure contains just a pointer to the callback that is being called. This is useful for managing the callback during execution of the callback function.

Any changes to a callback list, such as deletions, additions, or priority changes, are applied to the list after the current callback list has been processed. Changes to the list occur on the next traversal of the list. Here are two examples:

- Adding new callbacks: If a callback with priority 100 is being executed, and in that callback function, a new callback with priority 120 is created and added to the same list, that new callback is not called during the processing of the list.
- Changing the priority of callbacks: If a callback of priority 100 is executing, and it changes the priority of an existing priority 50 callback (in the same list) to priority 120, this callback is not processed during this traversal of the list.

Time Converters

Time converters are used to specify how time is converted from a numeric representation to a string representation, and vice versa. SmartSockets provides a number of standard time converters. You also have the option of building your own converters.

The list of known time converters can be traversed with the function TutTimeCvtTraverse. It uses a traversal function defined in this way:

```
typedef T_PTR (*T_TIME_CVT_TRAV_FUNC)
   (T_STR name, T_TIME_CVT time_cvt, T_PTR arg);
```

The specified traversal function must be defined with these arguments. The traversal function is called for each time converter currently defined, with the name, converter, and user defined argument specified in the call to TutTimeCvtTraverse.

When a time converter is created using TutTimeCvtCreate, the caller must specify two functions that are used to convert between string and numeric representations. These functions are defined in this way:

```
typedef T_BOOL (*T_TIME_CVT_NUM_TO_STR_FUNC)
   (T_REAL8 time_num, T_STR *time_str_return);

typedef T_BOOL (*T_TIME_CVT_STR_TO_NUM_FUNC)
   (T_STR time_str, T_REAL8 *time_num_return);
```

They return `TRUE` if the conversion was successful, and `FALSE` otherwise.

When a time converter is destroyed, a user-specified function can be called, if specified in the call to TutTimeCvtCreate. This function is defined in this way:

```
void (*T_TIME_CVT_DESTROY_FUNC)(void);
```

Hash Tables

The utility library provides a set of functions to manage hash tables. Hash tables are key / value pairs where the location of the key is derived from the value of the key. This technique results in high performance random access. Hash tables can be created, and entries can be inserted, deleted, and traversed. Hash tables are created with these functions:

- TutHashCreate — create a generic hash function based on a hash key calculation function provided.
- TutHashCreateInt — create a hash table with integer keys.
- TutHashCreateStr — create a hash table with string keys. These keys are case sensitive.
- TutHashCreateStri — create a hash table with string keys that are not case sensitive.
- TutHashCreatePtr — create a hash table with a key based on arbitrary pointer values.

When a hash table is created using TutHashCreate, two extra arguments are required: a test function and a calculation function. The test function is used to determine if two keys are equal and is defined as:

```
T_BOOL (*T_HASH_TEST_FUNC)(T_PTR key1, T_PTR key2);
```

It returns `TRUE` if the two keys are equal, and `FALSE` if they are different. The calculation function is used to calculate the hash table key for each entry in the hash table. It is defined as:

```
T_UINT4 (*T_HASH_CALC_FUNC) (T_PTR key);
```

This function returns a hash table key based on the value of the *key*. It is called whenever a new key / value pair is inserted into the hash table to calculate a key value, and when a lookup is performed using TutHashLookup.

The entries in a hash table are traversed using TutHashTraverse. This required traversal function is defined as:

```
T_PTR (*T_HASH_TRAV_FUNC)(T_PTR key, T_PTR value, T_PTR arg);
```

This function returns `NULL` to continue the traversal, and a non-null value to stop the traversal. All three arguments passed to the traversal function are of type T_PTR. They usually need to be cast to application-specific data structures.

Hash tables can also be sorted using TutHashSort. Because the order in which entries in a hash table are stored is not something you can control, it is sometimes useful to get a sorted list of these entries. TutHashSort provides this function by traversing the entries and selectively choosing the entries in the table (with the T_HASH_SORT_SELECT_FUNC), and then ordering them with the T_HASH_SORT_CMP_FUNC, both of which are provided to TutHashSort as arguments. The select function is defined as:

```
T_BOOL (*T_HASH_SORT_SELECT_FUNC)(T_PTR key, T_PTR value);
```

The select function is called for each entry of the hash table with the key and value part of the entry. This function must return `TRUE` if the entry is to be included, and `FALSE` if it is not to be selected.

TutHashSort also takes a compare function. It is defined as:

```
T_INT4 (*T_HASH_SORT_CMP_FUNC)(T_PTR key1, T_PTR key2);
```

The return value of this function works like the `strcmp` standard library function, and returns a negative value if the first argument is before the second, a positive value (greater than zero) if the first argument is after the second, and zero if the values are equal.

Hash tables are destroyed using TutHashDestroy. This function takes a destroy function defined as:

```
void (*T_HASH_DESTROY_FUNC)(T_PTR key, T_PTR value, T_PTR arg);
```

This function is called as each entry of the hash table is destroyed, allowing the application to perform whatever cleanup, such as freeing memory, is required for that particular entry.

When the hash function is applied to a key value, an integer is returned that is used to identify the bucket that the key belongs to. When multiple keys hash to the same bucket, the individual values are stored in a list that is traversed during a look up operation. The hash table bucket data structure is an opaque data type defined as:

```
struct T_HASH_BUCKET_STRUCT T_HASH_BUCKET_STRUCT, *T_HASH_BUCKET;
```

Pointer Arrays

Another class of functions provided by the utility library is the pointer array functions. Pointer arrays are used to manage arrays of pointers. They are similar to hash tables in some ways but they provide these differences:

- Ordering — pointer arrays are ordered by you and they maintain the order specified.
- No keys — pointer arrays do not require that a key value be specified.
- Immediate access — because pointer arrays are actual arrays, they can be accessed immediately from user code and have a very low overhead.

Two typedefs are provided for use with pointer arrays, the pointer array user function, and the pointer array compare function.

The pointer array user function is defined in this way:

```
void (*T_PTR_ARY_USER_FUNC)
   (T_INT4 index, T_PTR item, T_PTR arg);
```

This function typedef is used by TutPtrAryDestroy and TutPtrAryClear. It is called when the items in a pointer array are removed and allows you to perform cleanup actions as required.

The pointer array compare function is used when sorting the items in a pointer array. It is used by TutPtrArySort and is defined in this way:

```
T_INT4 (*T_PTR_ARY_CMP_FUNC)(T_PTR item1, T_PTR item2);
```

It returns a negative value if the first item is less than the second, zero if the two items are equal, or a positive value if the first is greater than the second.

Occasionally, you may want to receive notification when an item in a pointer array is changed, such as inserted or removed. This is accomplished using a pointer array change callback. There can be multiple pointer array change callbacks for a pointer array. The callback function is called with a pointer array change callback data structure, defined in this way:

```
struct T_PTR_ARY_CHANGE_CB_DATA_STRUCT {
   T_CB cb;
   T_PA_REASON reason;
   T_PTR item;     /* item changed. NULL if reason = T_PA_CLEARED */
   T_INT4 index;   /* index of item above. -1 if item is NULL */
   T_PTR item2;    /* item changed. NULL if reason != T_PA_ITEMS_SWAPPED */
   T_INT4 index2;  /* index of item above. -1 if item is NULL */
} T_PTR_ARY_CHANGE_CB_DATA_STRUCT, *T_PTR_ARY_CHANGE_CB_DATA;
```

The pointer array callback reasons are:

Reason	Reason Code
Item inserted	T_PA_ITEM_INSERTED
Item Removed	T_PA_ITEM_REMOVED
Item Replaced	T_PA_ITEM_REPLACED
All Items removed	T_PA_ITEMS_REMOVED
Items Swapped	T_PA_ITEMS_SWAPPED
Pointer Array Cleared	T_PA_CLEARED
Pointer Array Sorted	T_PA_SORTED

Pointer array change callbacks are managed with the functions TutPtrAryChangeCbCreate and TutPtrAryChangeCbLookup.

Options

SmartSockets uses options to control run-time configuration parameters that affect the operation of a process. You can create your own options that perform in a way similar to standard SmartSockets options, and you can write code to access and manipulate standard SmartSockets options. The types of options that are supported are defined in this way:

Type	Data Type	Restrictions
Boolean	T_OPT_TYPE_BOOLEAN	TRUE or FALSE only
Enumerated	T_OPT_TYPE_ENUM	One of a set of possible identifiers
Numeric	T_OPT_TYPE_NUMERIC	Floating point numeric value
String	T_OPT_TYPE_STRING	Any null-terminated string value
Enumerated List	T_OPT_TYPE_LIST_ENUM	List of enumerated values
Numeric List	T_OPT_TYPE_LIST_NUMERIC	List of numeric values
Enumerated String List	T_OPT_TYPE_LIST_STRING	List of string values

The option code provides a mechanism to verify a new option value before it is accepted. This is accomplished by specifying an option verify function that allows or disallows a new setting for an option value. It is defined in this way:

```
T_BOOL (*T_OPTION_VERIFY_FUNC) (T_OPTION opt, T_PTR new_value,
          T_PTR arg);
```

This function definition is used by TutOptionSetVerifyFunc and TutOptionGetVerifyFunc. There can only be one verify function for each option.

Occasionally, you may want to receive notification when the value of an option is changed. This is accomplished using an option change callback. These callbacks are called whenever the value of an option changes. There can be multiple option change callbacks for an option. The option callbacks are:

```
void (*T_OPTION_CHANGE_CB_FUNC)(T_OPTION option,
       T_OPTION_CHANGE_CB_DATA data, T_CB_ARG arg);
```

The callback function is called with an option change callback data structure, defined in this way:

```
typedef struct T_OPTION_CHANGE_CB_DATA_STRUCT {
  T_CB cb;
  T_PTR value;
} T_OPTION_CHANGE_CB_DATA_STRUCT, *T_OPTION_CHANGE_CB_DATA;
```

Option change callbacks are managed with the functions TutOptionChangeCbCreate and TutOptionChangeCbLookup.

Time Structure

SmartSockets stores the current value of data time in two different ways: as a double precision floating point value (T_REAL8) and also as a data structure (T_TIME_STRUCT) containing two integers. One for the whole part of the time value (`sec`) and one for the fractional part of the time value (`usec`), which stores the integer number of microseconds. The data structure is defined in this way:

```
struct T_TIME_STRUCT {
  T_INT4 sec;
  T_INT4 usec;
} T_TIME_STRUCT, *T_TIME;
```

API functions are provided to deal with both uses of time. TutSetCurrentTime and TutGetCurrentTime deal with time using T_REAL8 values, while TutSetCurrentTimeStruct and TutGetCurrentTimeStruct deal with time using T_TIME values. The T_REAL8 approach is provided for backward compatibility and for those applications where high precision time data is not required. The T_TIME approach is needed for applications where highly precise time definition is required. This is necessary because the number of digits of precision available in a T_REAL8 is not adequate to store time values down to millisecond precision when UNIX time values are used. In general, it is preferable to choose one method for setting and accessing time and use it consistently throughout the application.

Time Change Callbacks

SmartSockets provides a mechanism whereby you may request notification whenever the value of current time changes (data time). This is implemented with a time change callback. Once a time change callback has been created, the specified callback function is called whenever the value of current time changes. The time change callback function is defined in this way:

```
void (*T_TIME_CHANGE_CB_FUNC)
     (T_PTR dummy,    /* Always NULL */
      T_TIME_CHANGE_CB_DATA data,
      T_CB_ARG arg);
```

The API function to create a time change callback is TutTimeChangeCbCreate. A time change callback function is always called with exactly three arguments, where the first argument is always NULL, the second argument is a pointer to a time change callback data structure, and the third argument is the user-provided callback argument. The first argument is always NULL because the time change callback is a global-only type of callback and does not have a specific object with which it is associated.

The time change callback data structure is defined as:

```
struct T_TIME_CHANGE_CB_DATA_STRUCT {
  T_CB cb;         /* Pointer to the callback being called */
  T_TIME time;     /* Current value of data time */
} T_TIME_CHANGE_CB_DATA_STRUCT, *T_TIME_CHANGE_CB_DATA;
```

The current, that is, new, value of data time is contained in this structure as the `time` field.

Number Formats

Two enumerated typedefs are used to identify integer and real number formats. They are called T_INT_FORMAT and T_REAL_FORMAT. These enumerated values are returned by TutGetIntFormat and TutGetRealFormat, respectively. They identify the different integer and real number formats used by the computer where the program is running.

Integer Format	Definition
Invalid Format	T_INT_FORMAT_INVALID
Little Endian	T_INT_LITTLE_ENDIAN
Big Endian	T_INT_BIG_ENDIAN

Real Number Format	Definition
Invalid format	T_REAL_FORMAT_INVALID
DEC D Format (VAX default)	T_REAL_DEC_D
DEC G Format (Alpha AXP default)	T_REAL_DEC_G
IEEE Format	T_REAL_IEEE
IBM 370 Format	T_REAL_IBM_370

File Handling

The utility library provides some functions for working with files. These functions are helpful with file handling:

- TutFileTraverse — traverses all open files. Uses this function typedef:

  ```
  typedef T_PTR (*T_FILE_TRAV_FUNC) (T_FILE file, T_PTR arg);
  ```

  where *file* refers to the open file being visited, and *arg* is the user-specified traversal argument.

- TutFOpen — opens a file in a standard way.

- TutSetFileCloseFunc — specifies a function used to close a file. The close function is defined as:`T_BOOL (*T_FILE_CLOSE_FUNC) (FILE *file);`

  where *file* is the file being closed.

- TutSetFileOpenFunc — specifies a function used to open a file. The open function is defined as:

  ```
  FILE *(*T_FILE_OPEN_FUNC)
      (T_STR file_name, T_STR file_mode);
  ```

  where *file_name* specifies the file to be opened, and *file_mode* specifies the mode to be used to open the file.

- TutSetFileReadFunc — specifies a function used to read from a file. The read function is defined as:

  ```
  T_INT4 (*T_FILE_READ_FUNC)
      (T_PTR buff, T_UINT4 size, FILE *file);
  ```

  where *buff* specifies where data is to be placed, *size* specifies the number of bytes to be read, and *file* refers to the file being read.

XML Utilities

SmartSockets provides a simple object wrapper around an XML string. Applications can create, destroy, clone, and set an XML string into a new TutXml object by using these APIs:

- TutXmlClone — makes an identical copy of an XML object
- TutXmlCreate— creates a new XML object
- TutXmlCreateStatic— creates an XML object from a static buffer
- TutXmlDestroy — destroys an XML object
- TutXmlGetStr — returns the XML string associated with the XML object
- TutXmlSetStr— sets the XML string

Once an XML object is created, it can be added to a message using the TipcMsgAppendXml or TipcMsgAppendNamedXml APIs. See the *TIBCO SmartSockets User's Guide* for more information about adding and retrieving the XML object from a SmartSockets message.

Error Handling

SmartSockets provides some simple error handling for its C API functions. The error handling system provides additional information about why a function call failed. This is most often found in functions that return a TRUE or FALSE status. For function calls that support error handling, the global SmartSockets error number is set to a value that provides additional information about why the function failed. This global error number, which is stored in a per-thread value for multithreaded programs, can be retrieved with TutErrNumGet and set with TutErrNumSet. To simplify logging or printing notification of errors, a string describing the error can be retrieved with TutErrStrGet or TutErrNumToStr. The Diagnostics section of each function description describes how error handling is supported for that function. The error number should only be used immediately after a function which supports error handling returns a failure value. At any other time, the value of the error number is meaningless.

See the *TIBCO SmartSockets API Quick Reference* for the complete listing of the error codes and text for the C APIs, utilities, macros, and shell scripts.

Chapter 3 T_ASSERT - T_STRDUP

This chapter describes these SmartSockets utility macros:

- *T_ASSERT*
- *T_CALLOC*
- *T_ENTRY*
- *T_FREE*
- *T_MALLOC*
- *T_REALLOC*
- *T_STRDUP*

T_ASSERT

Name

T_ASSERT — program verification macro

Synopsis

```
void T_ASSERT(expression)
int expression;
```

Arguments

expression — any C expression

Return Values

None

Diagnostics

None

Description

The T_ASSERT macro is used to verify that *expression* evaluates to a non-zero value. If *expression* evaluates to 0, then T_ASSERT calls TutFatalError to print out an error message and abort. The message lists the expression that failed, the source file name, and the line number. T_ASSERT is very similar to the assert macro in `<assert.h>`. T_ASSERT is good for checking the validity of function arguments and return values. It is useful for doing sanity checks.

T_ASSERT is used internally in SmartSockets to verify program correctness. In the optimized versions of the SmartSockets processes and libraries, the T_ASSERT macros are compiled out. Any verification that always needs to be done is not done with T_ASSERT.

Caution

On non-ANSI-C compilers, you may experience problems using this macro if *expression* spans more than one line or contains string constants. The compatibility macro `T_ASSERT_STR` may be used instead in such instances. This macro is identical to `T_ASSERT` except that it does not provide the text of *expression*.

Examples This example checks a value:

```
char *foo(ptr, positive_counter)
struct some_struct *ptr;
T_INT4 positive_counter;
{
  char *ret_val;

  T_ASSERT(ptr != NULL);
  T_ASSERT(positive_counter > 0);

/* calculate ret_val */

   return ret_val;
} /* foo */

void bar()
{
  struct some_struct *ptr;
  char *val;

  val = foo(ptr, 3);
  T_ASSERT(val != NULL);
} /* bar */
```

T_CALLOC

Name

T_CALLOC — convenience macro to allocate and clear memory

Synopsis

```
void T_CALLOC(ptr, size, type)
type ptr;
int size;
```

Arguments

ptr — pointer variable to assign allocated memory to

size — number of bytes to allocate

type — data type to cast allocated memory to

Return Values

None

Diagnostics

None

Description

T_CALLOC uses TutCalloc to allocate memory, assigns the new memory to the pointer variable *ptr*, and checks for errors (TutCalloc returning `NULL`). Because TutCalloc returns a `(T_PTR)` pointer, *type* is used to cast the new memory to the appropriate type (to make the intention of the code clearer).

T_CALLOC can be used in most C contexts. T_CALLOC cannot be used in an expression.

Caution

type is used to build a cast and should not have parentheses around it.

The library function malloc is significantly faster than calloc because it does not have to clear the newly allocated memory. Use T_CALLOC instead of T_MALLOC if you need to put 0s in all bytes of the new memory. For example, a common use of T_CALLOC is to allocate storage for a pointer to a structure. If many of these structures are being allocated one at a time, it may be faster to use T_MALLOC and then explicitly clear the necessary fields in the new structure.

See Also

T_FREE, T_MALLOC, T_REALLOC, T_STRDUP

Examples

This example allocates memory:

```
char *char_ptr;
T_CALLOC (char_ptr, sizeof (char), char);
```

T_ENTRY

Name T_ENTRY — define a callback or thread function in a manner that is compatible across platforms

Synopsis

```
void T_ENTRY(thread or callback_function)
void thread or callback_function;
```

Arguments *thread* or *callback_function* — function being defined protoyped

Return Values None

Diagnostics None

Description T_ENTRY is necessary in the declaration and prototype of thread and callback functions to ensure cross-platform compatibility.

Caution None

See Also None

Examples This example defines a user-defined callback:

```
/* ============================================================ */
/* .. my_subject_cb -- a user-defined callback */
void T_ENTRY my_subject_cb (conn, data, arg)
T_IPC_CONN conn;
T_IPC_SRV_SUBJECT_CB_DATA data;
T_CB_ARG arg;
{
  /* function code here */
}
```

T_FREE

Name T_FREE — return allocated memory to the heap

Synopsis

```
void T_FREE(ptr)
T_PTR ptr;
```

Arguments *ptr* — pointer to memory to be freed

Return Values None

Diagnostics None

Description T_FREE uses TutFree to free dynamically allocated memory. T_FREE should only be used to free memory that has been allocated with T_MALLOC, T_CALLOC, T_REALLOC or T_STRDUP.

Caution Unlike the ANSI C function free, T_FREE does not accept null pointers.

See Also T_MALLOC, T_CALLOC, T_REALLOC, T_STRDUP

Examples This example allocates and frees some memory:

```
T_STR dup_1;
/* make a copy of a string */
T_STRDUP(dup_1, "hi there");
/* free it right away */
T_FREE(dup_1);
```

T_MALLOC

Name

T_MALLOC — convenience macro to allocate memory

Synopsis

```
void T_MALLOC(ptr, size, type)
type ptr;
int size;
```

Arguments

ptr — pointer variable to assign allocated memory to

size — number of bytes to allocate

type — data type to cast allocated memory to

Return Values

None

Diagnostics

None

Description

T_MALLOC uses TutMalloc to allocate memory, assigns the new memory to the pointer variable *ptr*, and checks for errors (TutMalloc returning `NULL`). Because TutMalloc returns a T_PTR pointer, *type* is used to cast the new memory to the appropriate type (to make the intention of the code clearer).

T_MALLOC can be used in most C contexts. T_MALLOC cannot be used in an expression.

Caution

type is used to build a cast and should not have parentheses around it.

The library function malloc is significantly faster than calloc because it does not have to clear the newly allocated memory. Use T_CALLOC instead of T_MALLOC if you need `0`'s placed in all bytes of the new memory. For example, a common use of T_CALLOC is to allocate storage for a pointer to a structure. If many of these structures are being allocated one at a time and most of the fields are being filled with actual data, it may be faster to use T_MALLOC and then explicitly clear the necessary fields in the new structure.

See Also

T_CALLOC, T_FREE, T_REALLOC, T_STRDUP

Examples

This example allocates space for a string:

```
T_STR string;

T_MALLOC(string, 80, T_STR);
strcpy(string, "hi there");
```

T_REALLOC

Name T_REALLOC — convenience macro to reallocate memory

Synopsis

```
void T_REALLOC(ptr, size, type)
type ptr;
int size;
```

Arguments

ptr — pointer variable to assign reallocated memory to

size — number of bytes to reallocate

type — data type to cast reallocated memory to

Return Values None

Diagnostics None

Description T_REALLOC uses TutRealloc to reallocate memory, assigns the new memory to the pointer variable *ptr*, and checks for errors (realloc returning NULL). Because TutRealloc returns a T_PTR pointer, *type* is used to cast the new memory to the appropriate type, to make the intention of the code clearer.

T_REALLOC can be used in most C contexts. T_REALLOC cannot be used in an expression.

T_REALLOC has the additional feature of being able to handle a NULL pointer. T_REALLOC checks if *ptr* is NULL, and if so, uses TutMalloc instead of TutRealloc. (Very few realloc implementations can handle NULL.) This is very useful for data structures that dynamically resize.

Caution *type* is used to build a cast and should not have parentheses around it.

See Also T_CALLOC, T_FREE, T_MALLOC, T_STRDUP

Examples This example uses an abstract data type that implements strings that do their own memory management to do a safe strcpy without T_REALLOC.

```
struct safe_string {
  T_INT4 size;
  T_STR buf;
};
T_STR safe_strcpy(string_ptr, string)
struct safe_string *string_ptr;
T_STR string;
{
  T_INT4 string_size;

  T_ASSERT(string_ptr != NULL);
  T_ASSERT(string != NULL);
  string_size = strlen(string) + 1; /* add one for NUL */

  if (string_size > string_ptr->size) {
    if (string_ptr->size == 0) {
        string_ptr->size += string_size;
        string_ptr->buf = malloc(string_ptr->size);
    }
    else {
     string_ptr->size += string_size;
      string_ptr->buf = realloc(string_ptr->buf, string_ptr->size);
    }
    if (string_ptr->buf == NULL) {
      TutWarning("safe_strcpy: alloc failed");
    }
 }
  strcpy(string_ptr->buf, string);
  return string_ptr->buf;
} /* safe_strcpy */
```

With T_REALLOC, the code for `safe_strcpy` is:

```
T_STR safe_strcpy(string_ptr, string)
struct safe_string *string_ptr;
T_STR string;
{
  T_INT4 string_size;

  T_ASSERT(string_ptr != NULL);
  T_ASSERT(string != NULL);
  string_size = strlen(string) + 1; /* add one for NUL */

  if (string_size > string_ptr->size) {
    string_ptr->size += string_size;
    T_REALLOC(string_ptr->buf, string_ptr->size, T_STR);
  }
  strcpy(string_ptr->buf, string);
  return string_ptr->buf;
/* safe_strcpy */
```

T_STRDUP

Name

T_STRDUP — duplicate a string

Synopsis

```
void T_STRDUP(dest, src)
T_STR dest;
T_STR src;
```

Arguments

dest — pointer to use for duplicate

src — string to be duplicated

Return Values

None

Diagnostics

None

Description

T_STRDUP is a macro that makes a copy of a string. The duplicated string should be freed with T_FREE when it is no longer needed.

Caution

T_STRDUP evaluates its arguments twice, so do not use ++ or -- in the arguments. For example, the results of `T_STRDUP(dest_ptr++, --src_ptr)` are undefined.

See Also

T_MALLOC, T_CALLOC, T_REALLOC, T_FREE

Examples

This example duplicates a string:

```
T_STR dup_1;
T_STR dup_2;

T_STRDUP(dup_1, "hi there");
T_STRDUP(dup_2, dup_1);
```

Chapter 4 TutBufAllocAligned - TutBufSetSize

This chapter describes the SmartSockets TutBuf utility functions:

- *TutBufAllocAligned*
- *TutBufAppendBuf*
- *TutBufAppendRange*
- *TutBufCloneRange*
- *TutBufCreate*
- *TutBufCreateStatic*
- *TutBufDeleteRange*
- *TutBufDestroy*
- *TutBufGetSize*
- *TutBufGrow*
- *TutBufNextAligned*
- *TutBufReset*
- *TutBufRewind*
- *TutBufSetSize*

TutBufAllocAligned

Name TutBufAllocAligned — allocate space in a T_BUF object

Synopsis

```
T_PTR TutBufAllocAligned(buf, size, align)
T_BUF buf;
T_INT4 size;
T_INT4 align;
```

Arguments

buf — T_BUF object in which to allocate the space

size — number of bytes to allocate

align — alignment factor

Return Values Pointer to the allocated memory block if successful, `NULL` otherwise.

Diagnostics None

Description TutBufAllocAligned moves the "write pointer" in the T_BUF object to make space for a block of data. This block of data is to be copied into the buffer directly. The buffer grows, if required, to accommodate the space. The alignment factor is used to ensure that this new block ends on a valid boundary (say, 8 bytes), and the "write pointer" is rounded up so that the end of this block is aligned to this boundary.

Caution Note that the returned pointer is valid only until the next call to a buffer operation that may resize the buffer, because the buffer may be moved during a resize.

See Also TutBufNextAligned

Examples This example allocates a block 10 bytes long that ends on an 8-byte boundary:

```
T_PTR ptr;
ptr = TutBufAllocAligned(buf, 10, 8);
if (ptr == NULL) {
   /* error */
}
```

TutBufAppendBuf

Name TutBufAppendBuf — append the contents of one buffer into another

Synopsis

```
T_PTR TutBufAppendBuf(dest, src, size)
T_BUF dest;
T_BUF src;
T_INT4 size;
```

Arguments

dest — T_BUF object to receive the data

src — T_BUF object to provide the data

size — number of bytes to copy

Return Values Pointer into the destination buffer where the data was copied if successful, `NULL` otherwise.

Diagnostics None

Description TutBufAppend copies data from the source buffer starting at that buffer's "read pointer" into the destination buffer, appending the data onto the end of the destination buffer. The destination buffer grows, if necessary, to accommodate the data. If the given size is `-1`, then all of the data in the source buffer from the current location to the end of the buffer is copied. If the given size is past the end of the source buffer, this function fails.

Caution The returned pointer is only valid until the next call to a buffer operation that may resize the buffer, because the buffer may be moved during a resize.

See Also TutBufCreate, TutBufDestroy, TutBufAppendRange

Examples This example copies eight bytes from the buffer *src* to the buffer *dest*:

```
T_PTR ptr;
ptr = TutBufAppendBuf(dest, src, 8);
if (ptr == NULL) {
   /* error */
}
```

TutBufAppendRange

Name TutBufAppendRange — append the contents of a range of one buffer into another

Synopsis

```
T_PTR TutBufAppendRange(dest, src, begin, end)
T_BUF dest;
T_BUF src;
T_INT4 begin;
T_INT4 end;
```

Arguments

dest — T_BUF object to receive the data

src — T_BUF object to provide the data

begin — start of the block to copy

end — end of the block to copy

Return Values Pointer into the destination buffer where the data was copied if successful, NULL otherwise.

Diagnostics None

Description TutBufAppendRange copies a block of data *end* – *begin* bytes long from the source buffer starting at *begin* into the destination buffer, appending the data onto the end of the destination buffer. The destination buffer grows, if necessary, to accommodate the data. If the given *end* is -1, then all of the data in the source buffer from *begin* to the end of the buffer is copied. If *begin* or *end* are outside of the range of the source buffer, this function fails.

Caution The returned pointer is only valid until the next call to a buffer operation that may resize the buffer, because the buffer may be moved during a resize.

See Also TutBufCreate, TutBufDestroy, TutBufAppendBuf

Examples This example copies 8 bytes from the buffer *src* starting at byte 3 to the buffer *dest*:

```
T_PTR ptr;
ptr = TutBufAppendBuf(dest, src, 3, 11);

if (ptr == NULL) {
  /* error */
}
```

TutBufCloneRange

Name

TutBufCloneRange — create a new buffer containing a portion of an existing buffer

Synopsis

```
T_BUF TutBufCloneRange(src, begin, end)
T_BUF src;
T_INT4 begin;
T_INT4 end;
```

Arguments

src — T_BUF object to provide data

begin — start of block to copy

end — end of block to copy

Return Values

A new T_BUF object containing the copied data if successful, `NULL` otherwise.

Diagnostics

None

Description

TutBufCloneRange creates a new buffer and copies a byte range from an existing buffer into the new buffer. The byte range in the source buffer starts at *begin*, and ends at *end*. If *end* is `-1`, then the complete block from *begin* to the end of the buffer is copied.

Caution

None

See Also

TutBufCreate, TutBufDestroy, TutBufAppendRange

Examples

This example copies eight bytes from the buffer *src*, starting at byte 3, to a new buffer:

```
T_BUF buf;
buf = TutBufCloneRange(src, 3, 11);
if (buf == NULL) {
  /* error */
}
```

TutBufCreate

Name TutBufCreate — create a new buffer

Synopsis

```
T_BUF TutBufCreate(size)
T_INT4 size;
```

Arguments *size* — size of the new buffer, in bytes

Return Values A new T_BUF object if successful, `NULL` otherwise.

Diagnostics None

Description This function creates a new buffer large enough to hold at least *size* bytes. The buffer can later grow, if necessary, to accommodate more data.

Caution None

See Also TutBufDestroy, TutBufCreateStatic

Examples This example creates a new buffer 128 bytes in size:

```
T_BUF buf;
buf = TutBufCreate(128);
if (buf == NULL) {
  /* error */
}
```

TutBufCreateStatic

Name

TutBufCreateStatic — wrap a buffer around an existing block of data

Synopsis

```
T_BUF TutBufCreateStatic(ptr, size)
T_PTR ptr;
T_INT4 size;
```

Arguments

ptr — pointer to the existing block of data

size — size of the new buffer, in bytes

Return Values

A new T_BUF object if successful, `NULL` otherwise.

Diagnostics

None

Description

TutBufCreateStatic creates a new buffer using an existing block of data. This data is not copied, but used in place, so alterations to this buffer alter the original data. Static buffers do not grow to accommodate new data. When the buffer is destroyed with TutBufDestroy, the actual data is not deallocated, only the T_BUF object itself.

Caution

None

See Also

TutBufDestroy, TutBufCreate

Examples

This example creates a static buffer from a block of data:

```
T_PTR ptr;

ptr = T_CALLOC(128);
buf = TutBufCreateStatic(ptr, 128);
if (buf == NULL) {
  /* error */
}
else {
  /* ptr may be used directly here, say to a file */
}
```

TutBufDeleteRange

Name

TutBufDeleteRange — delete a block of data from a buffer

Synopsis

```
T_BUF TutBufDeleteRange(buf, begin, end)
T_BUF buf;
T_INT4 begin;
T_INT4 end;
```

Arguments

buf — T_BUF object from which to delete the data

begin — start of the block to delete

end — end of the block to delete

Return Values

`TRUE` if successful, `FALSE` otherwise.

Diagnostics

None

Description

TutBufDeleteRange deletes a block of data from a buffer, starting at *begin*. All data starting at *end* (or the end of the buffer if *end* is `-1`) is shifted left to fill the hole. If either *begin* or *end* are outside of the range of the buffer, this function fails. This function also fails if *end* is before *begin*.

Caution

None

See Also

TutBufCreate, TutBufDestroy, TutBufAppendRange

Examples

This example deletes 12 bytes of data from a buffer, starting at byte 6:

```
T_BOOL status;
status = TutBufDeleteRange(buf, 6, 18);
if (status == FALSE) {
  /* error */
}
```

TutBufDestroy

Name TutBufDestroy — destroy a buffer

Synopsis

```
T_BOOL TutBufDestroy(buf)
T_BUF buf;
```

Arguments *buf* — the buffer to destroy

Return Values TRUE if successful, FALSE otherwise.

Diagnostics None

Description TutBufDestroy destroys a buffer. If the buffer is not static, because it was created with TutBufCreate, then the buffer object and all of the data contained by that object is destroyed. If the buffer is static, because it was created with TutBufCreateStatic, then only the buffer object itself is destroyed, not the data.

Caution None

See Also TutBufCreate, TutBufCreateStatic

Examples This example destroys a buffer:

```
T_BOOL status;
status = TutBufDestroy(buf);
if (status == FALSE) {
  /* error */
}
```

TutBufGetSize

Name TutBufGetSize — retrieve the number of bytes written to the buffer

Synopsis

```
T_BOOL TutBufGetSize(buf, size_return)
T_BUF buf;
T_INT4 *size_return;
```

Arguments *buf* — buffer to retrieve the size from

size_return — pointer to store the size

Return Values TRUE if succesful, FALSE otherwise.

Diagnostics None

Description TutBufGetSize retrieves the number of bytes written into the buffer. This can also be used as a write pointer, because all writes append to the end of the data in the buffer.

Caution None

See Also TutBufSetSize

Examples This example gets the size of a buffer:

```
T_INT4 size;
T_BOOL status;

status = TutBufGetSize(buf, &size);
if (status == FALSE) {
  /* error */
}
```

TutBufGrow

Name

TutBufGrow — grow the allocated size of the buffer

Synopsis

```
T_BOOL TutBufGrow(buf, new_size)
T_BUF buf;
T_INT4 new_size;
```

Arguments

buf — buffer to grow

new_size — new size of the buffer

Return Values

TRUE if successful, FALSE otherwise.

Diagnostics

None

Description

TutBufGrow attempts to resize a buffer to allow it to hold at least *new_size* bytes. The buffer may grow larger than the requested size. The buffer may be moved, and the existing data is copied, if necessary. This function is called automatically by any operation that needs to increase the size of the buffer. If this function is called on a static buffer, it fails.

Caution

None

See Also

TutBufCreate

Examples

This example grows a buffer to 256 bytes:

```
T_BOOL status;

status = TutBufGrow(buf, 256);
if (status == FALSE) {
  /* error */
}
```

TutBufNextAligned

Name

TutBufNextAligned — reserve space in a T_BUF object

Synopsis

```
T_PTR TutBufNextAligned(buf, size, align)
T_BUF buf;
T_INT4 size;
T_INT4 align;
```

Arguments

buf — T_BUF object in which to reserve the space

size — number of bytes to reserve

align — alignment factor

Return Values

Pointer to the memory block at the current position if successful; NULL otherwise.

Diagnostics

None

Description

TutBufNextAligned returns the current position in the T_BUF object to you, and then moves the current position to reserve the requested space. The new position is *size* bytes past the current position you started with, rounded up to align it with the given alignment factor.

Caution

The returned pointer is only valid until the next call to a buffer operation that may resize the buffer, because the buffer may be moved during a resize.

See Also

TutBufAllocAligned

Examples

This example moves the read index 10 bytes forward, ending on an 8-byte boundary:

```
T_PTR ptr;
ptr = TutBufNextAligned(buf, 10, 8);
if (ptr == NULL) {
  /* error */
}
```

TutBufReset

Name

TutBufReset — reset the buffer contents

Synopsis

```
T_BOOL TutBufReset(buf, ptr)
T_BUF buf;
T_PTR ptr;
```

Arguments

buf — T_BUF object to reset

ptr — new position

Return Values

TRUE if successful, FALSE otherwise.

Diagnostics

None

Description

TutBufReset moves the "write pointer" to the location given by *ptr*. If *ptr* is NULL, then the buffer is reset to empty. This function fails if *ptr* is outside the bounds of the buffer.

Caution

None

See Also

TutBufRewind

Examples

This example resets a buffer to the beginning:

```
T_BOOL status;
status = TutBufReset(buf, NULL);
if (status == FALSE) {
  /* error */
}
```

TutBufRewind

Name

TutBufRewind — rewind the current position in the buffer

Synopsis

```
T_BOOL TutBufRewind(buf, ptr)
T_BUF buf;
T_PTR ptr;
```

Arguments

buf — T_BUF object to rewind

ptr — new position

Return Values

TRUE if successful, FALSE otherwise.

Diagnostics

None

Description

TutBufRewind moves the current position to the location given by *ptr*. If *ptr* is NULL, then the current position is set to the beginning. This function fails if *ptr* is outside the bounds of the buffer.

Caution

None

See Also

TutBufReset

Examples

This example rewinds a buffer to the beginning:

```
T_BOOL status;
status = TutBufRewind(buf, NULL);
if (status == FALSE) {
  /* error */
}
```

TutBufSetSize

Name TutBufSetSize — set the valid number of bytes in the buffer

Synopsis

```
T_BOOL TutBufSetSize(buf, size)
T_BUF buf;
T_INT4 size;
```

Arguments *buf* — T_BUF object to set

size — new size

Return Values TRUE if successful, FALSE otherwise.

Diagnostics None

Description TutBufSetSize sets the valid number of bytes in the buffer. The buffer grows if the new size is larger than the buffer's allocated size. The allocated size does not shrink if the size is reduced, however. Any data in the buffer past the new size is discarded. The size may be changed for static buffers, so long as the new size is less than the allocated size (the initial size for the static buffer). If the static buffer must grow to accommodate the new size, this function fails.

Caution None

See Also TutBufGetSize

Examples This example sets the size of a buffer to 128 bytes:

```
T_BOOL status;
status = TutBufSetSize(buf, 128);
if (status == FALSE) {
  /* error */
}
```

Chapter 5 TutCalloc - TutCondWakeAll

This chapter describes these SmartSockets utility functions:

- *TutCalloc*
- *TutCbDestroy*
- *TutCbGetArgument*
- *TutCbGetFunction*
- *TutCbGetPriority*
- *TutCbSetArgument*
- *TutCbSetFunction*
- *TutCbSetPriority*
- *TutCommandParseFile*
- *TutCommandParseStr*
- *TutCommandParseTypedStr*
- *TutCondCreate*
- *TutCondDestroy*
- *TutCondWait*
- *TutCondWakeAll*

TutCalloc

Name

TutCalloc — allocate and clear memory

Synopsis

```
T_PTR TutCalloc(size)
T_UINT4 size;
```

Arguments

size — number of bytes to allocate

Return Values

Pointer to the allocated memory block if successful; NULL otherwise.

Diagnostics

None

Description

TutCalloc allocates and zeroes out a block of memory of the requested size. Using this function ensures the portability of the user-written applications to all platforms that are currently supported by SmartSockets.

TutCalloc allows *size* to be 0; this is useful for allocating a non-null placeholder that is resized later with TutRealloc.

All SmartSockets memory management uses the functions TutCalloc, TutFree, TutMalloc, and TutRealloc. You can set debugger breakpoints in these functions to trace SmartSockets memory usage.

Caution

TutCalloc, TutFree, TutMalloc, and TutRealloc are not compatible with other memory management functions. Memory allocated by SmartSockets (including memory allocated by TutCalloc, TutMalloc, or TutRealloc) must be reallocated with TutRealloc and freed with TutFree. Attempting to mix memory management functions will result in unpredictable and potentially unstable behavior.

See Also

TutFree, TutMalloc, TutRealloc

Examples

This example allocates and clears a string 30 characters long:

```
T_STR strptr;

/* the string has to be 30 characters long, so one space has to be */
/* reserved for the terminating NULL */
strptr = (T_STR)TutCalloc(31);
```

TutCbDestroy

Name TutCbDestroy — destroy a callback and remove it from the list

Synopsis

```
T_BOOL TutCbDestroy(cb)
T_CB cb;
```

Arguments *cb* — callback to be destroyed

Return Values TRUE if successful, FALSE otherwise.

Diagnostics None

Description This function destroys a callback. *cb* is removed from the list to which it belongs and is never called again. Once this function is called, *cb* is no longer valid. The pointer should not be referenced again.

Caution Do not use *cb* after destroying it.

See Also TutCbSetPriority

Examples This example creates a callback and then destroys it when called. The result is a callback that is processed exactly once.

```
static void T_ENTRY time_change_cb(dummy, data, arg)
T_PTR dummy; /* always NULL */
T_TIME_CHANGE_CB_DATA data;
T_CB_ARG arg;
{
  TutOut("Time change callback called\n");
  if (!TutCbDestroy(data->cb)) {
  /* error */
  }
}
...
if (TutTimeChangeCbCreate(time_change_cb, NULL) == NULL) {
  /* error */
}
```

TutCbGetArgument

Name TutCbGetArgument — get the callback argument of the specified callback

Synopsis

```
T_BOOL TutCbGetArgument(cb, arg_return)
T_CB cb;
T_CB_ARG *arg_return;
```

Arguments *cb* — callback to get argument for

arg_return — pointer to location for the argument

Return Values TRUE if successful, FALSE otherwise.

Diagnostics None

Description TutCbGetArgument returns the callback argument that is associated with *cb*.

Caution Pass in the address of a T_CB_ARG variable; otherwise, unexpected results may occur.

See Also TutCbSetArgument

Examples This example gets the argument of a callback and prints it out:

```
static void T_ENTRY time_change_cb(dummy, data, arg)
T_PTR dummy; /* always NULL */
T_TIME_CHANGE_CB_DATA data;
T_CB_ARG arg;
{
  TutOut("Time change callback called, argument: %s\n", arg);
  if (T_STREQI(arg, "hello")) {
    TutCbSetArgument(data->cb, "bye");
  }
  else {
    TutCbSetArgument(data->cb, "hello");
  }
}
T_CB cb;
T_PTR cb_arg;

cb = TutTimeChangeCbCreate(time_change_cb, "hello");
/* other code */

if (!TutCbGetArgument(cb, &cb_arg)) {
  /* error */
}
TutOut("Callback argument: %s\n", (T_STR)cb_arg);
```

TutCbGetFunction

Name

TutCbGetFunction — get the callback function of the specified callback

Synopsis

```
T_BOOL TutCbGetFunction(cb, func_return)
T_CB cb;
T_CB_FUNC *func_return;
```

Arguments

cb — callback to get the function of

func_return — pointer to location for the result

Return Values

TRUE if successful, FALSE otherwise.

Diagnostics

None

Description

TutCbGetFunction returns the callback function that is called when *cb* is next processed and stores it in the location pointed to by *func_return*.

This function uses the generic callback function definition, so it is usually necessary to cast the type-specific callback function to the generic type when using this function.

Caution

None

See Also

TutCbSetFunction

Examples

This example creates a callback and then alternates between two functions for the callback function:

```
static void T_ENTRY time_change_cb_odd(dummy, data, arg)
T_PTR dummy; /* always NULL */
T_TIME_CHANGE_CB_DATA data;
T_CB_ARG arg;
{
  T_CB_FUNC cb_func;
 /* Forward declaration of the other callback function so the */
 /* compiler will recognize this symbol */
  extern T_TIME_CHANGE_CB_FUNC time_change_cb_even;
```

```
  TutOut("Time change odd callback called\n");
 /* Make sure the callback function is set properly. */
  if (!TutCbGetFunction(data, &cb_func)) {
    /* error */
  }
  if (cb_func != (T_CB_FUNC)time_change_cb_odd) {
    /* error */
  }
  if (!TutCbSetFunction(data->cb, (T_CB_FUNC)time_change_cb_even))
{
    /* error */
  }
}

static void T_ENTRY time_change_cb_even(dummy, data, arg)
T_PTR dummy; /* always NULL */
T_TIME_CHANGE_CB_DATA data;
T_CB_ARG arg;
{
  T_CB_FUNC cb_func;
  TutOut("Time change even callback called\n");
 /* Make sure the callback function is set properly. */
  if (!TutCbGetFunction(data, &cb_func)) {
  /* error */
  }
  if (cb_func != (T_CB_FUNC)time_change_cb_even) {
    /* error */
  }
  if (!TutCbSetFunction(data->cb, (T_CB_FUNC)time_change_cb_odd))
{
    /* error */
  }
}

if (TutTimeChangeCbCreate(time_change_cb_even, NULL) == NULL) {
  /* error */
}
```

TutCbGetPriority

Name

TutCbGetPriority — get the current priority of a callback

Synopsis

```
T_BOOL TutCbGetPriority(cb, priority_return)
T_CB cb;
T_CB_PRIORITY *priority_return;
```

Arguments

cb — callback to get the priority of

priority_return — pointer to storage for the priority

Return Values

`TRUE` if successful, `FALSE` otherwise.

Diagnostics

None

Description

This function retrieves the priority of *cb* and stores it in the location pointed to by *priority_return*.

Caution

None

See Also

TutCbSetPriority

Examples

This example prints the priority of a callback:

```
static void T_ENTRY time_change_cb(dummy, data, arg)
T_PTR dummy; /* always NULL */
T_TIME_CHANGE_CB_DATA data;
T_CB_ARG arg;
{
  T_CB_PRIORITY cb_pri;
  if (!TutCbGetPriority(data->cb, &cb_pri)) {
    /* error */
  }
  TutOut("priority is: %d\n", cb_pri);
}
...
if (TutTimeChangeCbCreate(time_change_cb, NULL) == NULL) {
  /* error */
}
```

TutCbSetArgument

Name

TutCbSetArgument — set the callback argument used by the specified callback

Synopsis

```
T_BOOL TutCbSetArgument(cb, arg)
T_CB cb;
T_CB_ARG arg;
```

Arguments

cb — callback to set the argument of

arg — new value for the callback argument

Return Values

TRUE if successful, FALSE otherwise.

Diagnostics

None

Description

This function sets the callback argument that is used the next time *cb* is called. This ability means you can affect existing callbacks without having to destroy and recreate the callback.

Caution

The callback list to which *cb* belongs is searched to make sure *arg* and the existing callback function do not match any existing callbacks in the list. If a match is found, the function returns FALSE.

The callback routines do not make a copy of the argument provided. Caller must manage the storage pointed to. This means that the address of an automatic variable, for example, should not be used as *arg*.

See Also

TutCbGetArgument

Examples This example gets the argument of a callback and prints it out:

```
static void T_ENTRY time_change_cb(dummy, data, arg)
T_PTR dummy; /* always NULL */
T_TIME_CHANGE_CB_DATA data;
T_CB_ARG arg;
{
  TutOut("Time change callback called, argument: %s\n", arg);

  if (T_STREQI(arg, "hello")) {
  TutCbSetArgument(data->cb, "bye");
  }
  else {
    TutCbSetArgument(data->cb, "hello");
  }
}
...
T_CB cb;
T_PTR cb_arg;

cb = TutTimeChangeCbCreate(time_change_cb, "hello");

/* other code */

if (!TutCbGetArgument(cb, &cb_arg)) {
  /* error */
}
TutOut("Callback argument: %s\n", (T_STR)cb_arg);
```

TutCbSetFunction

Name TutCbSetFunction — set the callback function

Synopsis

```
T_BOOL TutCbSetFunction(cb, func)
T_CB cb;
T_CB_FUNC func;
```

Arguments *cb* — callback to set function for

func — callback function to be used

Return Values `TRUE` if successful, `FALSE` otherwise.

Diagnostics None

Description TutCbSetFunction changes the callback function referenced by *cb* to *func*. It is useful for situations where the purpose of a callback has changed, but the need for the callback still exists. This capability eliminates the need to destroy and recreate a new callback just to change the purpose of the callback.

This function uses the generic callback function definition, so it is usually necessary to cast any type-specific callback function to the generic T_CB_FUNC for use with this function. This results in some loss of type checking, but greater flexibility.

Caution The value provided for *func* must not be `NULL`.

See Also TutCbGetFunction

Examples This example creates a callback and then alternates between two functions for the callback function:

```
static void T_ENTRY time_change_cb_odd(dummy, data, arg)
T_PTR dummy; /* always NULL */
T_TIME_CHANGE_CB_DATA data;
T_CB_ARG arg;
{
  T_CB_FUNC cb_func;

 /* Forward declaration of the other callback function so the */
 /* compiler will recognize this symbol */
  extern T_TIME_CHANGE_CB_FUNC time_change_cb_even;

  TutOut("Time change odd callback called\n");
 /* Make sure the callback function is set properly. */
  if (!TutCbGetFunction(data, &cb_func)) {
  /* error */
  }
  if (cb_func != (T_CB_FUNC)time_change_cb_odd) {
  /* error */
  }
  if (!TutCbSetFunction(data->cb, (T_CB_FUNC)time_change_cb_even))
{
  /* error */
  }
}

static void T_ENTRY time_change_cb_even(dummy, data, arg)
T_PTR dummy; /* always NULL */
T_TIME_CHANGE_CB_DATA data;
T_CB_ARG arg;
{
  T_CB_FUNC cb_func;
  TutOut("Time change even callback called\n");
 /* Make sure the callback function is set properly. */
  if (!TutCbGetFunction(data, &cb_func)) {
  /* error */
  }
  if (cb_func != (T_CB_FUNC)time_change_cb_even) {
  /* error */
  }
  if (!TutCbSetFunction(data->cb, (T_CB_FUNC)time_change_cb_odd))
{
  /* error */
  }
}

if (TutTimeChangeCbCreate(time_change_cb_even, NULL) == NULL) {
  /* error */
}
```

TutCbSetPriority

Name

TutCbSetPriority — set the priority of a callback

Synopsis

```
T_BOOL TutCbSetPriority(cb, pri)
T_CB cb;
T_CB_PRIORITY pri;
```

Arguments

cb — callback to set priority of

pri — new value for priority

Return Values

`TRUE` if successful, `FALSE` otherwise.

Diagnostics

None

Description

TutCbSetPriority sets the priority of *cb* to *pri*. The priority is applied dynamically to the list specified, causing the order of callbacks to be changed, even if the list to which the callback belongs is active. This may result in a callback being skipped during the current traversal, or not called at all, depending on the priority of the current callback.

When the list is reordered, a newly prioritized callback is placed at the end of all callbacks with a matching priority.

If the new priority matches the current priority of the callback, the list is not changed.

New callbacks are created with a priority of `0`. Callback priorities range from `-32768` to `32767`.

Caution

May cause callbacks to be repeated or missed if used during callback list processing.

See Also

TutCbGetPriority

Examples This example increments the priority of a callback. Every time the callback is called, its priority is incremented by one.

```
static void T_ENTRY time_change_cb(dummy, data, arg)
T_PTR dummy; /* always NULL */
T_TIME_CHANGE_CB_DATA data;
T_CB_ARG arg;

{
  T_CB_PRIORITY cb_pri;
  TutOut("Time change callback called\n");
  if (!TutCbGetPriority(data->cb, &cb_pri)) {
    /* error */
  }
  cb_pri++;
  if (!TutCbSetPriority(data->cb, cb_pri)) {
    /* error */
  }
}
...
if (!TutTimeChangeCbCreate(time_change_cb, NULL) == NULL) {
  /* error */
}
```

TutCommandParseFile

Name

TutCommandParseFile — submit a file to SmartSockets process command interface

Synopsis

```
T_BOOL TutCommandParseFile(file_name)
T_STR file_name;
```

Arguments

file_name — name of file to submit to command interface

Return Values

TRUE if the file was processed successfully, FALSE otherwise.

Diagnostics

None

Description

TutCommandParseFile submits a file to the command interface of the process for execution. Each line of the file is executed with TutCommandParseStr.

Caution

TutCommandParseFile reads the file one line at a time using the fgets function. Multi-line comments, multi-line commands, and commands longer than 4094 characters are not allowed.

TutCommandParseFile does not search for *file_name* in any directories such as the SmartSockets standard directory, but simply uses *file_name* verbatim.

See Also

TutCommandParseStr, TutCommandParseTypedStr

Examples

This example executes the file `test.cm`:

```
if (!TutCommandParseFile("test.cm")) {
  /* error */
}
```

TutCommandParseStr

Name

TutCommandParseStr — submit a non-interactive string to the command interface

Synopsis

```
void TutCommandParseStr(command_string)
T_STR command_string;
```

Arguments

command_string — string to submit to the command interface

Return Values

None

Diagnostics

None

Description

TutCommandParseStr submits a non-interactive string to the command interface for execution. (A non-interactive string refers to a string that appears in a program, not with the SmartSockets command interface.) If there is a syntax error in the *command_string*, a message is printed to the output window. If the command is successfully executed, and the Command_Feedback option is set to `always`, command feedback is printed to the output window; if Command_Feedback is set to `never` or `interactive`, no feedback is generated.

To submit an interactive string, such as a string typed by you at the command interface, use the function TutCommandParseTypedStr.

Caution

If the Command_Feedback option is set to `interactive`, TutCommandParseStr does not generate output for normal completion.

See Also

TutCommandParseFile, TutCommandParseTypedStr

Examples

This example sets several options:

```
TutCommandParseStr("setopt application demo");
TutCommandParseStr("setopt log_in_data input.msg");
TutCommandParseStr("setopt editor emacs");
```

TutCommandParseTypedStr

Name TutCommandParseTypedStr — submit an interactive string to the command interface

Synopsis

```
void TutCommandParseTypedStr(command_string)
T_STR command_string;
```

Arguments *command_string* — string to submit to the command interface

Return Values None

Diagnostics None

Description TutCommandParseTypedStr submits an interactive string, such as if it was typed by you, to the command interface for execution. If there is a syntax error in *command_string*, a message is displayed on the output window. If the command is successfully executed, and the Command_Feedback option is set to `always` or `interactive`, command feedback is displayed on the output window. If Command_Feedback is set to `never`, no feedback is generated.

To submit a non-interactive string to the command interface use the function TutCommandParseStr. This function behaves identically to TutCommandParseStr, except for how command feedback is reported.

Caution If the Command_Feedback option is set to `interactive`, TutCommandParseTypedStr does not generate output for normal completion.

See Also TutCommandParseFile, TutCommandParseStr

Examples This example sets several options:

```
TutCommandParseTypedStr("setopt project demo");
TutCommandParseTypedStr("setopt log_in_data input.msg");
TutCommandParseTypedStr("setopt editor emacs");

Option project set to demo.
Now logging incoming data-related messages to <input.msg>.
Option log_in_data set to "input.msg".
Option editor set to "emacs".
```

TutCondCreate

Name TutCondCreate — create a condition variable

Synopsis

```
T_COND TutCondCreate()
```

Arguments None

Return Values Condition variable if successful, T_INVALID_COND otherwise.

Diagnostics If TutCondCreate fails, it returns T_INVALID_COND and sets the global SmartSockets error number to one of:

- T_ERR_OS — an operating system error has occurred. The function TutErrNumGetOs should be called to obtain the related OS-specific error value.
- T_ERR_TYPE_MISMATCH — the platform does not support threads

Description TutCondCreate creates a condition variable. Condition variables provide the mechanism by which one thread can wait for another thread to complete an activity. Mutexes alone cannot do this; they can only ensure serialization of an activity. Condition variables are used in conjunction with mutexes, allowing a thread to wait until an arbitrary condition has occurred.

There are two basic operations on a condition variable: waiting for the condition and waking all waiting threads when the condition occurs. First, one or more threads wait on the condition variable. When the threads are later awakened, all waiting threads are allowed to proceed. A wake operation on a condition variable that does not have any waiting threads is not remembered. The next thread that waits on the condition variable blocks until the condition variable is again awakened.

The condition variable should be destroyed when it is no longer needed by calling TutCondDestroy.

Caution None

See Also TutCondDestroy, TutCondWait, TutCondWakeAll

Examples This example creates a condition variable, waits to be awakened and then destroys the condition variable:

```
T_COND cond;
T_MUTEX mutex;
T_BOOL status;

cond = TutCondCreate();
if (T_INVALID_COND == cond) {
  /* error */
}

/* create mutex with initial lock */
mutex = TutMutexCreate(TRUE);
if (T_INVALID_MUTEX == mutex) {
  /* error */
}

/* wait for condition */
while (!predicate()) {
  status = TutCondWait(cond, mutex, T_TIMEOUT_FOREVER);
  if (FALSE == status) {
    /* error */
  }
}

...

if (!TutCondDestroy(cond)) {
  /* error */
}

if (!TutMutexDestroy(mutex)) {
  /* error */
}
```

TutCondDestroy

Name TutCondDestroy — destroy a condition variable

Synopsis

```
T_BOOL TutCondDestroy(cond)
T_COND cond;
```

Arguments *cond* — condition variable to destroy

Return Values TRUE if condition variable was successfully destroyed, FALSE otherwise.

Diagnostics If TutCondDestroy fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_OS — an operating system error has occurred. The function TutErrNumGetOs should be called to obtain the related OS-specific error value.
- T_ERR_TYPE_MISMATCH — the platform does not support threads
- T_ERR_VAL_INVALID — *cond* is not a valid condition variable

Description TutCondDestroy destroys a condition variable. All operating system resources associated with the condition variable are released.

Caution Destroying a condition variable that other threads (in the same process) are waiting on has undefined results.

See Also TutCondCreate

Examples This example creates and then destroys a condition variable:

```
T_COND cond;
cond = TutCondCreate();
if (T_INVALID_COND == cond) {
  /* error */
}
...
if (!TutCondDestroy(cond)) {
  /* error */
}
```

TutCondWait

Name

TutCondWait — wait to be awakened by a condition variable

Synopsis

```
T_BOOL TutCondWait(cond, mutex, timeout)
T_COND cond;
T_MUTEX mutex;
T_REAL8 timeout;
```

Arguments

cond — condition variable to wait on

mutex — locked mutex to release and re-obtain

timeout — maximum number of seconds to wait for condition variable

Return Values

TRUE if thread was awakened by the condition variable, FALSE otherwise.

Diagnostics

If TutCondWait fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_OS — an operating system error has occurred. The function TutErrNumGetOs should be called to obtain the related OS-specific error value.
- T_ERR_TIMEOUT_REACHED — the *timeout* period was reached
- T_ERR_TYPE_MISMATCH — the platform does not support threads
- T_ERR_VAL_INVALID — *cond* or *mutex* was not valid

Description

TutCondWait causes the calling thread to be suspended until awakened by the condition variable or until the *timeout* period specified has expired. The reserved *timeout* value T_TIMEOUT_FOREVER may be used to specify an infinite wait. The mutex passed to this function must be locked at the time the function is called. The mutex is unlocked while the thread is suspended and locked again before the function returns.

Caution

Care must be taken to assure that the mutex passed to this function has not been recursively locked by the calling thread. TutCondWait unlocks the mutex passed to it so as to allow other threads access to the protected resources utilized by the condition variable's predicate. If the calling thread holds recursive locks on the mutex, no other threads are able to obtain a lock on the mutex so as to change the state of the predicate.

The semantics of condition variables can also lead to threads being awakened at times they should not. TutCondWait must normally be called within a loop whose controlling expression is a test of the predicate that the condition variable is intended to reflect. See the example below.

See Also TutCondCreate, TutCondWakeAll

Examples This example creates a condition variable, waits to be awakened and then destroys the condition variable:

```
T_COND cond;
T_MUTEX mutex;
T_BOOL status;

cond = TutCondCreate();
if (T_INVALID_COND == cond) {
  /* error */
}

/* create mutex with initial lock */
mutex = TutMutexCreate(TRUE);
if (T_INVALID_MUTEX == mutex) {
  /* error */
}

/* wait for condition */
while (!predicate()) {
  status = TutCondWait(cond, mutex, T_TIMEOUT_FOREVER);
  if (FALSE == status) {
  /* error */
  }
}

...

if (!TutCondDestroy(cond)) {
  /* error */
}

if (!TutMutexDestroy(mutex)) {
  /* error */
}
```

TutCondWakeAll

Name TutCondWakeAll — wake all threads waiting for a condition variable

Synopsis

```
T_BOOL TutCondWakeAll(cond)
T_COND cond;
```

Arguments *cond* — condition variable whose waiting threads are to be awakened

Return Values TRUE if threads were successfully awakened or if no threads were waiting on the specified condition variable, FALSE otherwise.

Diagnostics If TutCondWakeAll fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_OS — an operating system error has occurred. The function TutErrNumGetOs should be called to obtain the related OS-specific error value
- T_ERR_TYPE_MISMATCH — the platform does not support threads
- T_ERR_VAL_INVALID — *cond* is not a valid condition variable

Description TutCondWakeAll wakes all threads that were waiting on the specified condition variable. A wake operation on a condition variable that does not have any waiting threads is not remembered. The next thread that waits on the condition variable blocks until the condition variable is again awakened.

Caution None

See Also TutCondCreate, TutCondWait

Examples This example creates a condition variable and a mutex. It then wakes any threads waiting on the condition variable. Finally, it destroys both the condition variable and the mutex.

```
T_COND cond;
T_MUTEX mutex;
T_BOOL status;

cond = TutCondCreate();
if (T_INVALID_COND == cond) {
  /* error */
}

mutex = TutMutexCreate(FALSE);
if (T_INVALID_MUTEX == mutex) {
  /* error */
}

/* lock the mutex */
if (!TutMutexLock(mutex, T_TIMEOUT_FOREVER)) {
  /* error */
}

...

/* unlock the mutex */
if (!TutMutexUnlock(mutex)) {
  /* error */
}

/* wake threads waiting for the condition */
if (!TutCondWakeAll(cond)) {
  /* error */
}

if (!TutCondDestroy(cond)) {
  /* error */
}

if (!TutMutexDestroy(mutex)) {
  /* error */
}
```

Chapter 6 TutErrNumGet - TutFree

This chapter describes these SmartSockets utility functions:

- *TutErrNumGet*
- *TutErrNumGetC*
- *TutErrNumGetOs*
- *TutErrNumGetSocket*
- *TutErrNumSet*
- *TutErrNumToStr*
- *TutErrStrCreate*
- *TutErrStrGet*
- *TutExit*
- *TutFileTraverse*
- *TutFOpen*
- *TutFree*

TutErrNumGet

Name TutErrNumGet — get the value of the global SmartSockets error number

Synopsis

```
T_INT4 TutErrNumGet()
```

Arguments None

Return Values The value of the global SmartSockets error number.

Diagnostics None

Description The global SmartSockets error number is similar to the C-style error number `errno`. A function that returns a failure status (`FALSE` or `NULL`) often needs to provide more information to you when an error occurs. This additional error information can be accessed using TutErrNumGet.

This error number is stored in a per-thread value to accommodate multi-threaded programs.

Caution Not all functions set the global SmartSockets error number. Therefore, TutErrNumGet may return a value that is unrelated to the previous function call. The error number can be reset using `TutErrNumSet(0)` before calling a function that may set the error number.

If the error number is set more than once in a function, TutErrNumGet returns only the last value set.

See Also TutErrNumGetC, TutErrNumGetOs, TutErrNumGetSocket, TutErrNumSet, TutErrStrGet

Examples This example prints the global SmartSockets error number and error string if a function fails:

```
if (TipcMsgGetPriority(msg, &priority) == FALSE) {
  TutOut("TipcMsgGetPriority failed: error %d <%s>.\n",
     TutErrNumGet(), TutErrStrGet());
}
```

TutErrNumGetC

Name TutErrNumGetC — get the value of the C error number

Synopsis

```
T_INT4 TutErrNumGetC()
```

Arguments None

Return Values The value of the C error number.

Diagnostics None

Description The operating system error number is a C-specific value indicating how the last C function failed. The error number is valid only when TutErrNumGet reports T_ERR_C.

This error number is stored in a value for each thread to accommodate multi-threaded programs.

Caution TutErrNumGetC should only be called when TutErrNumGet returns T_ERR_C.

See Also TutErrNumGet, TutErrNumGetOs, TutErrNumGetSocket

Examples This example checks if a C-specific error occurred and prints the related C error code:

```
if (TutErrNumGet() == T_ERR_C) {
  TutOut("C error %d occurred.\n", TutErrNumGetC());
}
```

TutErrNumGetOs

Name TutErrNumGetOs — get the value of the operating system error number

Synopsis

```
T_INT4 TutErrNumGetOs()
```

Arguments None

Return Values The value of the operating system error number.

Diagnostics None

Description The operating system error number is a value indicating how the last OS function failed. The error number is valid only when TutErrNumGet reports T_ERR_OS.

On OpenVMS, if the error also has a C equivalent, call the function TutErrNumGetC to obtain the related C-specific error value (TutErrNumGetC returns `0` if there is no equivalent C error).

This error number is stored in a value for each thread to accommodate multi-threaded programs.

Caution TutErrNumGetOs should only be called when TutErrNumGet returns T_ERR_OS.

See Also TutErrNumGet, TutErrNumGetC, TutErrNumGetSocket

Examples This example checks if an OS error occurred and prints the related OS error code:

```
if (TutErrNumGet() == T_ERR_OS) {
  TutOut("OS error %d occurred.\n", TutErrNumGetOs());
}
```

TutErrNumGetSocket

Name TutErrNumGetSocket — get the value of the socket error number

Synopsis

```
T_INT4 TutErrNumGetSocket()
```

Arguments None

Return Values The value of the socket error number.

Diagnostics None

Description The socket error number is a value indicating how the last socket function failed. The error number is valid only when TutErrNumGet reports T_ERR_SOCKET.

This error number is stored in a value for each thread to accommodate multi-threaded programs.

Caution TutErrNumGetSocket should only be called when TutErrNumGet returns T_ERR_SOCKET.

See Also TutErrNumGet, TutErrNumGetC, TutErrNumGetOs

Examples This example checks if a socket error occurred and prints the related socket error code:

```
if (TutErrNumGet() == T_ERR_SOCKET) {
  TutOut("Socket error %d occurred.\n", TutErrNumGetSocket());
}
```

TutErrNumSet

Name TutErrNumSet — set the value of the global SmartSockets error number

Synopsis

```
void TutErrNumSet(err_num)
T_INT4 err_num;
```

Arguments *err_num* — the value to which the global SmartSockets error number should be set

Return Values None

Diagnostics None

Description The SmartSockets error number is similar to the C-style error number `errno`. A function that returns a failure status (`FALSE` or `NULL`) often needs to provide more information when an error occurs. TutErrNumSet is used to set the global SmartSockets error number. One common use for this function is to set the error number to `0` before calling an API function that may set the error number. It may also be used in user code to provide additional error information to other user-defined code. The value of the error number is not used in SmartSockets internal code, so it does not have an effect on SmartSockets processing.

Caution When TutErrNumSet is called, the previous value of the error number is overwritten.

See Also TutErrNumGet

Examples This example sets the error number if the argument is invalid:

```
T_BOOL return_data(T_PTR data_ptr)
{
  if (data_ptr == NULL) {
    TutErrNumSet(T_ERR_NULL_PTR);
    return FALSE;
  }
  ...
} /* return_data */
```

TutErrNumToStr

Name TutErrNumToStr— convert a global SmartSockets error number into a error string

Synopsis

```
T_STR TutErrNumToStr(err_num)
T_INT4 err_num;
```

Arguments *err_num* — the global SmartSockets error number to be converted

Return Values A string describing the error.

Diagnostics None

Description This function converts a SmartSockets error number into a string describing that error. If the error number does not have a matching error string, the string "unknown error *x*" is returned, with *x* set to the error number in question.

Caution The value returned by TutErrNumToStr points directly into an internal data structure and should not be modified. The contents of the value are guaranteed to be preserved until the next call to a SmartSockets function.

See Also TutErrNumGet, TutErrStrGet

Examples This example prints the name of a retrieved error number:

```
err_num = TutErrNumGet();
TutOut("error: %s\n", TutErrNumToStr(err_num));
```

TutErrStrCreate

Name

TutErrStrCreate — add a new string to the error string table

Synopsis

```
T_BOOL TutErrStrCreate(err_num, err_str)
T_INT4 err_num;
T_STR err_str;
```

Arguments

err_num — the new error number

err_str — the descriptive name for the new error number

Return Values

TRUE if the string was successfully added, FALSE otherwise.

Diagnostics

If TutErrStrCreate fails, it returns a FALSE value and the global SmartSockets error number is set to:

- T_ERR_NULL_PTR — *err_str* was NULL

Description

This function adds a new error number with a matching descriptive string to the error string table. Subsequent calls to TutErrNumToStr or TutErrStrGet are able to retrieve this error string.

Caution

All error numbers below 100000 are reserved for standard SmartSockets error numbers. TutErrStrCreate does not make a copy of *err_str*.

See Also

TutErrNumToStr, TutErrStrGet

Examples

This example creates a new error number:

```
if (!TutErrStrCreate(200000, "disk exploded")) {
  /* error */
}
```

TutErrStrGet

Name TutErrStrGet — get the global SmartSockets error number as a descriptive string

Synopsis

```
T_STR TutErrStrGet()
```

Arguments None

Return Values A string describing the current error number.

Diagnostics None

Description TutErrStrGet retrieves the current global SmartSockets error number as a descriptive string. If the error number does not have a matching error string, the string "`unknown error` *x*" is returned, with *x* set to the error number in question.

Caution The value returned by TutErrStrGet points directly into an internal data structure and should not be modified. The contents of the value are guaranteed to be preserved until the next call to a SmartSockets function.

See Also TutErrNumGet, TutErrNumToStr

Examples This example prints the error description string:

```
TutOut("error: %s\n", TutErrStrGet());
```

TutExit

Name `TutExit` — call exit handlers and terminate process

Synopsis

```
void TutExit(status)
T_INT4 status;
```

Arguments *status* — exit status of the process

Return Values None

Diagnostics None

Description TutExit calls all exit handlers registered internally by SmartSockets in the reverse order of their registration and then calls the system call exit(). If the process has registered any functions by calling atexit(), these functions are called only after the internal SmartSockets exit handlers functions have been called.

Although TutExit does not return a value, the exiting process returns an exit status to the calling process. The exit status codes T_EXIT_FAILURE and T_EXIT_SUCCESS are defined and may be used to indicate successful or unsuccessful process termination.

Caution If the process calls the system call exit() instead of TutExit(), the internal SmartSockets exit handlers are not called and the behavior of the process or application is unknown. The correct way to terminate a SmartSockets application or process is to call TutExit().

Examples This example shows the correct way to exit a SmartSockets process:

```
#include <rtworks/ipc.h>

void MyExitHandler()
{
  /* do something .... */
  TutOut("Exit from My exit Handler\n");
} /* MyExitHandler */
int main()
{
  T_IPC_MT mt;
  T_IPC_MSG msg;
  T_BOOL status;

  mt = TipcMtCreate(T_MT_INFO);

 msg = TipcMsgCreate(mt);
  /* Do something */

  /*
   * The process needs to call TutExit() before exiting
   * in order to get the correct behavior.
   */
 TutExit(T_EXIT_SUCCESS);
} /* main */
```

TutFileTraverse

Name

TutFileTraverse — traverse open files

Synopsis

```
T_PTR TutFileTraverse(trav_func, trav_arg)
T_FILE_TRAV_FUNC trav_func;
T_PTR trav_arg;
```

Arguments

trav_func — function to be called for each open file

trav_arg — traversal argument to pass to *trav_func*

Return Values

Last value returned by *trav_func*.

Diagnostics

If TutFileTraverse fails, it returns `NULL` without calling *trav_func*, and the global SmartSockets error number is set to:

- T_ERR_NULL_PTR — *trav_func* was `NULL`

Description

TutFileTraverse traverses all the files that have been opened and calls *trav_func* once for each of these files. The traversal continues until all the open files have been traversed, or until *trav_func* returns a non-null value. The last value returned by *trav_func* is returned to the caller.

Caution

None

See Also

TutSetFileOpenFunc, TutSetFileCloseFunc, TutSetFileReadFunc

Examples

This example traverses all open files and prints out the name and mode for the file:

```
T_ENTRY trav_func(file, arg)
T_FILE file;
T_PTR arg;
{
  TutOut("File name: %s, mode %s\n", file->fname, file->mode);
  return NULL;
}
TutFileTraverse(trav_func, NULL);
```

TutFOpen

Name TutFOpen — open a file in a platform-independent way

Synopsis

```
FILE *TutFOpen(file_name, file_mode)
T_STR file_name;
T_STR file_mode;
```

Arguments *file_name* — name of file to be opened

file_mode — mode to open file in

Return Values FILE pointer of opened file. NULL if unable to open *file_name* using *file_mode*.

Diagnostics If TutFOpen fails, it returns NULL and sets the global SmartSockets error number to one of these values:

- T_ERR_C — a C error has occurred. Call the function TutErrNumGetC to obtain the related C-specific error value.
- T_ERR_NULL_PTR — either *file_name* or *file_mode* was NULL

Description This function calls the C library function fopen using a standard set of arguments that opens a file in a "normal" way. This provides an operating system independent way to open a file. If called with the mode argument of "r", "w", or "a", then any necessary platform-specific arguments are added to make fopen behave with the expected UNIX semantics. On OpenVMS platforms, additional arguments are added to specify file sharing characteristics. The arguments expected for *file_mode* and their meanings are:

- "r" — open *file_name* for reading. If *file_name* does not already exist, return NULL.
- "w" — open for writing. If *file_name* already exists, it is overwritten. If it does not exist, create it.
- "a" — open *file_name* for appending. If *file_name* does not exist, create it.

Any other value for *file_mode* causes the function fopen to be called with the specified argument.

Caution Do not use this function to open a file with special arguments. If any special arguments are required, such as additional arguments that may be needed on OpenVMS, call fopen directly.

See Also TutSetFileOpenFunc, TutSetFileCloseFunc, TutSetFileReadFunc

Examples This example opens a file in write mode, truncating any previous contents of the file, and then writes one line of data into it:

```
FILE *file;

file = TutFOpen("myfile.dat", "w");
if (file == NULL) {
  /* error */
}

fprintf(file, "%s", "This is a data file\n");

fclose(file);
```

TutFree

Name TutFree — free the previously allocated memory block

Synopsis

```
void TutFree(ptr)
T_PTR ptr;
```

Arguments *ptr* — pointer to the memory block to be freed

Return Values None

Diagnostics None

Description TutFree frees the previously allocated block of memory. Using this function in the code ensures portability to all the platforms that SmartSockets currently supports.

All SmartSockets memory management uses the functions TutCalloc, TutFree, TutMalloc, and TutRealloc. You can set debugger breakpoints in these functions to trace SmartSockets memory usage.

Caution TutCalloc, TutFree, TutMalloc, and TutRealloc are not compatible with other memory management functions. Memory allocated by SmartSockets (including memory allocated by TutCalloc, TutMalloc, or TutRealloc) must be reallocated with TutRealloc and freed with TutFree. Attempting to mix memory management functions will result in unpredictable and potentially unstable behavior.

See Also TutCalloc, TutMalloc, TutRealloc

Examples This example allocates and frees a string 30 characters long:

```
T_STR strptr;

/* the string has to be 30 characters long, so one space has to be */
/* reserved for the terminating NULL */
strptr = (T_STR)TutCalloc(31);
...
TutFree(strptr);
```

Chapter 7 TutGetCurrentTime - TutHashTraverse

This chapter describes the TutGet, TutHash, and TutKey functions:

- *TutGetCurrentTime*
- *TutGetCurrentTimeStruct*
- *TutGetenv*
- *TutGetIntFormat*
- *TutGetNodeName*
- *TutGetOutFlushFunc*
- *TutGetOutputFunc*
- *TutGetRealFormat*
- *TutGetSocketDir*
- *TutGetVersionName*
- *TutGetVersionNumber*
- *TutGetWallTime*
- *TutGetWallTimeStruct*
- *TutHashBucketGetValue*
- *TutHashBucketSetValue*
- *TutHashCreate*
- *TutHashCreateInt*
- *TutHashCreatePtr*
- *TutHashCreateStr*
- *TutHashCreateStri*
- *TutHashDelete*
- *TutHashDestroy*
- *TutHashDump*
- *TutHashGetSize*
- *TutHashInsert*

- *TutHashLookup*
- *TutHashLookupBucket*
- *TutHashReplace*
- *TutHashSort*
- *TutHashTraverse*

TutGetCurrentTime

Name TutGetCurrentTime — get the current SmartSockets data time

Synopsis

```
T_REAL8 TutGetCurrentTime()
```

Arguments None

Return Values Floating-point number containing current SmartSockets data time.

Diagnostics None

Description TutGetCurrentTime returns the current SmartSockets data time. Data time is the time contained in SmartSockets TIME messages. The interpretation of the current data time depends on the application. Normally, the data time has some relationship to the computer wall clock time, but this is not always the case.

Caution TutGetCurrentTime does *not* return the current wall clock time; to get system time, use TutGetWallTime instead.

TutGetCurrentTime returns 0.00 unless you first explicitly set the time by calling TutSetCurrentTime(). TutGetCurrentTime is a legacy function for use with the SmartSockets Product Family (RTworks). Most SmartSockets programs instead use TutGetWallTime, which returns the current system time value.

See Also TutGetWallTime, TutTimeNumToStr

Examples This example looks up the time converter `hms`, gets the current SmartSockets system time, and then converts the time to a string using the `hms` converter:

```
T_TIME_CVT time_cvtr;
T_REAL8 system_time;
T_REAL8 wall_time;

wall_time = TutGetWallTime();
TutSetCurrentTime(wall_time);
system_time = TutGetCurrentTime();
TutOut("Raw system time is %f.\n", system_time);
time_cvtr = TutTimeCvtLookup("hms");
if (time_cvtr != NULL) {
  T_STR *time_str;
  if (TutTimeCvtNumToStr(time_cvtr, system_time, time_str)) {
    TutOut("Formatted system time is %s\n", time_str);
  }
}
```

TutGetCurrentTimeStruct

Name TutGetCurrentTimeStruct — get the current SmartSockets data time as a time struct

Synopsis

```
T_BOOL TutGetCurrentTimeStruct(time_return)
T_TIME_STRUCT *time_return;
```

Arguments *time_return* — pointer to location to store time structure

Return Values TRUE if successful, FALSE otherwise.

Diagnostics None

Description TutGetCurrentTimeStruct returns the current SmartSockets data time into a T_TIME_STRUCT. Data time is the time that is contained in SmartSockets time messages. The interpretation of the current data time is application-specific. Normally, the data time has some relationship to the computer wall clock time, but this is not always the case.

Caution TutGetCurrentTimeStruct does not return the current wall clock time. Use TutGetWallTimeStruct for that.

See Also TutGetWallTimeStruct, TutTimeNumToStr

Examples This example prints the current SmartSockets data time:

```
T_TIME_STRUCT time_struct;

if (!TutGetCurrentTimeStruct(&time_struct)) {
  TutOut("Could not get current data time\n");
  return;
}
TutOut("Current data time is %d.%06d\n",
        time_struct.sec, time_struct.usec);
```

TutGetenv

Name

TutGetenv — get a SmartSockets environment variable

Synopsis

```
T_BOOL TutGetenv(name, value_return)
T_STR name;
T_STR *value_return;
```

Arguments

name — name of environment variable whose value is to be retrieved

value_return — pointer to location to store the environment variable assigned to *name*

Return Values

TRUE if if the environment was successfully retrieved, FALSE otherwise.

Diagnostics

None

Description

TutGetenv gets a value from the environment. The environment is a list of name / value pairs that is set by the operating system for each process.

On Windows, TutGetenv gets the value from the runtime associated with the SmartSockets libraries, which may be different that the runtime that the application is running in.

Caution

None

See Also

TutPutenv

Examples

This example gets the value of RTHOME from the environment:

```
T_STR val;
if (!TutGetenv("RTHOME", &val)) {
  /* error */
}
else {
  TutOut("Value of RTHOME environment variable is %s\n", val);
}
```

TutGetIntFormat

Name TutGetIntFormat — get the integer number format of this computer

Synopsis

```
T_INT_FORMAT TutGetIntFormat()
```

Arguments None

Return Values T_INT_BIG_ENDIAN or T_INT_LITTLE_ENDIAN.

Diagnostics None

Description TutGetIntFormat returns the format of integers on the current computer. Integers are either big-endian (the most significant byte is stored in the highest byte) or little-endian (the most significant byte is stored in the lowest byte). Software that transfers data between computers with different number formats, such as between a SPARC and VAX machines, needs to know the format of integers and real numbers to convert the data from one format to another.

Caution None

See Also TutGetRealFormat

Examples This example prints the byte order of integers:

```
if (TutGetIntFormat() == T_INT_BIG_ENDIAN) {
  TutOut("Integers are big-endian.\n");
}
else {
  TutOut("Integers are little-endian.\n");
}
```

TutGetNodeName

Name TutGetNodeName — get the name of the current node

Synopsis

```
T_STR TutGetNodeName(name_return)
T_STR name_return;
```

Arguments *name_return* — storage for node name

Return Values String containing node name.

Diagnostics None

Description TutGetNodeName is a convenience function that gets the name of the current node. On UNIX, TutGetNodeName uses either the uname or gethostname function. On Windows, TutGetNodeName uses the WinSock `gethostname` function. On MVS, TutGetNodeName uses the IBM TCP/IP `gethostname` function. If that fails, it uses the SMF ID. On OpenVMS TutGetNodeName translates the logical `SYS$NODE` and removes the trailing double colon.

If TutGetNodeName cannot obtain the node name from the operating system, it stores `UNKNOWN` in *name_return*.

Caution There is no way to know ahead of time how much storage is needed for *name_return*.

The value of *name_return* cannot be `NULL`. Storage for *name_return* must be managed by the caller.

See Also None

Examples This example prints the name of the current node:

```
T_CHAR node_name[80];
TutOut("Current node name is %s.\n", TutGetNodeName(node_name));
```

TutGetOutFlushFunc

Name TutGetOutFlushFunc — get the function used for TutOutFlush output

Synopsis

```
T_OUT_VA_FLUSH_FUNC TutGetOutFlushFunc()
```

Arguments None

Return Values The output function used by TutOutFlush.

Diagnostics None

Description TutGetOutFlushFunc gets the function used by TutOutFlush to display output.

Caution None

See Also TutOutFlush, TutSetOutFlushFunc

Examples This example sets and restores all TutOutFlush output temporarily:

```
T_OUT_VA_FLUSH_FUNC flush_func;

flush_func = TutGetOutFlushFunc();
TutSetOutFlushFunc(dummy_func);
TutOutFlush(); /* dummy_func used here */
TutSetOutFlushFunc(flush_func); /* restore */
```

TutGetOutputFunc

Name TutGetOutputFunc — get the function used for TutOut output

Synopsis

```
T_OUT_VA_FUNC TutGetOutputFunc()
```

Arguments None

Return Values The output function used by TutOut.

Diagnostics None

Description TutGetOutputFunc gets the function used by TutOut to display output. The default is vprintf. See the reference page for TutSetOutputFunc for more information on this output function.

Caution None

See Also TutOut, TutSetOutputFunc

Examples This example sets and restores all TutOut output temporarily:

```
T_OUT_VA_FUNC save_func;

save_func = TutGetOutputFunc();
TutSetOutputFunc(dummy_func);

/* call functions which produce output (e.g., TipcSrvCreate) */

TutSetOutputFunc(save_func); /* restore */
```

TutGetRealFormat

Name TutGetRealFormat — get the real number format of this computer

Synopsis

```
T_REAL_FORMAT TutGetRealFormat()
```

Arguments None

Return Values T_REAL_DEC_D, T_REAL_DEC_G, T_REAL_IEEE, or T_REAL_IBM_370.

Diagnostics None

Description TutGetRealFormat returns the format of real numbers on the current computer. Real numbers use one of these formats:

- DEC D — default format for OpenVMS VAX
- DEC G — default format for OpenVMS AXP
- IEEE format — all other SmartSockets architectures use this format

Real numbers are also subject to the same byte order that integers use: they are either big-endian (the most significant byte is stored in the highest byte) or little-endian (the most significant byte is stored in the lowest byte).

Software that transfers data between computers with different number formats, such as between SPARC and VAX machines, needs to know the format of integers and real numbers to convert the data from one format to another.

Caution None

See Also TutGetIntFormat

Examples This example prints the format of real numbers:

```
switch(TutGetRealFormat()) {
case T_REAL_DEC_G:
  TutOut("Real numbers are DEC G format.\n");
  break;
case T_REAL_DEC_D:
  TutOut("Real numbers are DEC D format.\n");
  break;
case T_REAL_IEEE:
  TutOut("Real numbers are IEEE format.\n");
  break;
default:
  TutOut("Unknown real format: %d\n", TutGetRealFormat());
}
```

TutGetSocketDir

Name TutGetSocketDir — get name of directory where local socket files are written

Synopsis

```
T_STR TutGetSocketDir()
```

Arguments None

Return Values String containing directory name.

Diagnostics None

Description TutGetSocketDir returns the directory name where all SmartSockets local sockets are located. For Windows systems, this does not apply. On UNIX, this is usually the directory `/tmp/rtworks`. On OpenVMS, this is usually the directory `RTSOCKETS: [SOCKETS.`*node*`]` where *node* is the node name of the computer.

TutGetSocketDir also uses `mkdir` to create the directory if it does not already exist.

When running SmartSockets, it is often necessary to create files (such as UNIX domain sockets) that are world-readable and world-writable. To avoid forcing users to create world-writable directories under their own home directory, TutGetSocketDir is used instead.

Caution The string returned by TutGetSocketDir points into an internal data structure and should not be modified.

See Also See the *TIBCO SmartSockets Application Programming Interface* reference for information on TipcConnCreateServerLocal and TipcConnCreateClientLocal.

Examples This example opens a file with the specified name in the directory returned by TutGetSocketDir:

```
FILE *open_ipc_log_file(name)
T_STR name;
{
  T_STRING file_name;

#ifdef T_OS_VMS
  sprintf(file_name, "%s%s", TutGetSocketDir(), name);
#else
  sprintf(file_name, "%s/%s", TutGetSocketDir(), name);
#endif
return TutFOpen(file_name, "w");
} /* open_ipc_log_file */
```

TutGetVersionName

Name TutGetVersionName — get the SmartSockets software version name

Synopsis

```
T_STR TutGetVersionName()
```

Arguments None

Return Values String containing version name.

Diagnostics None

Description TutGetVersionName retrieves the current version number of SmartSockets as a string of the form `Version` *a.bc mm/dd/yy status*, where *a* is the major release number, *b* is the minor release number, *c* is an optional maintenance release number, *mm/dd/yy* is an optional date, and *status* is an optional status string.

Caution The version string returned by TutGetVersionName points to read-only storage and should not be modified.

See Also TutGetVersionNumber

Examples Every SmartSockets process prints the version name to stdout and puts the version name in the window title bar:

```
/*..MyInit -- program initialization with argc, argv */
void MyInit(argc, argv)
T_INT4 argc;
T_STR *argv;
{
  TutOut("MyInit %s\n", TutGetVersionName());
  ...
} /* MyInit */
```

TutGetVersionNumber

Name TutGetVersionNumber — get the SmartSockets software version number

Synopsis

```
T_INT4 TutGetVersionNumber()
```

Arguments None

Return Values Integer version number.

Diagnostics None

Description TutGetVersionNumber returns an integer that represents the software version number. SmartSockets Version *X.Y* would have a version number of *XY0*. For example, version 3.0 corresponds to version number 300.

Caution TutGetVersionNumber returns an integer, not a floating-point number.

See Also TutGetVersionName

Examples This example prints the software version number:

```
TutOut("TIBCO SmartSockets software version number is %ld.\n",
        TutGetVersionNumber());
```

TutGetWallTime

Name TutGetWallTime — get the current wall clock (computer) time

Synopsis

```
T_REAL8 TutGetWallTime()
```

Arguments None

Return Values Floating-point number containing current computer wall clock time in the number of seconds since midnight January 1, 1970.

Diagnostics None

Description TutGetWallTime is a convenience function that returns the current wall clock time as a double-precision floating-point number.

On UNIX, MVS, and Windows, TutGetWallTime calls gettimeofday (that fills in a timeval structure) and converts the result to a T_REAL8. This value is the number of seconds since midnight January 1, 1970 GMT.

On OpenVMS, TutGetWallTime calls ftime (that fills in a timeb_t structure) and converts the result to a T_REAL8. This value is also the number of seconds since midnight, January 1, 1970 local time (OpenVMS has no concept of time zones).

Caution TutGetWallTime does not return the current SmartSockets data time. Use TutGetCurrentTime for that.

See Also TutGetCurrentTime, TutGetWallTimeStruct

Examples This example looks up the time converter `hms`, gets the current wall clock time, and then converts the time to a string using the `hms` converter:

```
T_TIME_CVT time_cvtr;
T_REAL8 wall_time;
wall_time = TutGetWallTime();
TutOut("Raw wall clock time is %f.\n", wall_time);
time_cvtr = TutTimeCvtLookup("hms");

if (time_cvtr != NULL) {
    T_STR time_str;
    if (TutTimeCvtNumToStr(time_cvtr, wall_time, &time_str)) {
      TutOut("Formatted wall clock time is %s\n", time_str);
    }
}
```

TutGetWallTimeStruct

Name

TutGetWallTimeStruct — get the current wall clock (computer) time as a time struct

Synopsis

```
T_BOOL TutGetWallTimeStruct(time_return)
T_TIME_STRUCT *time_return;
```

Arguments

time_return — pointer to location to store wall time

Return Values

TRUE if successful, FALSE otherwise.

Diagnostics

None

Description

TutGetWallTimeStruct is a convenience function that returns the current wall clock time into a T_TIME_STRUCT pointed to by the caller.

On UNIX, MVS, and Windows, TutGetWallTimeStruct calls gettimeofday (that fills in a timeval structure) and fills in the fields of the pointed to T_TIME_STRUCT. This value is the number of seconds and microseconds since midnight January 1, 1970 GMT.

On OpenVMS, TutGetWallTimeStruct calls ftime (that fills in a timeb_t structure) and fills in the fields of the provided T_TIME_STRUCT. This value is the number of seconds and microseconds since midnight, January 1, 1970 local time. (OpenVMS has no concept of time zones.)

Caution

TutGetWallTimeStruct does not return the current SmartSockets data time. Use TutGetCurrentTimeStruct for that.

See Also

TutGetCurrentTimeStruct, TutGetWallTime

Examples

This example gets the current wall clock time, and then prints out the time with a formatted print statement:

```
T_TIME_STRUCT time_struct;
if (!TutGetWallTimeStruct(&time_struct)) {
  TutOut("Could not get wall time\n");
  return;
}
TutOut("Raw wall clock time is %d.%06d.\n",
        time_struct.sec, time_struct.usec);
```

TutHashBucketGetValue

Name TutHashBucketGetValue — get the value in a hash bucket

Synopsis

```
T_PTR TutHashBucketGetValue(bucket)
T_HASH_BUCKET bucket;
```

Arguments *bucket* — the hash bucket to get value from

Return Values The current value that is stored in the bucket.

Diagnostics None

Description TutHashBucketGetValue gets the value stored in *bucket*. Before calling this function, obtain a valid bucket by calling TutHashLookupBucket.

Caution None

See Also TutHashLookupBucket, TutHashBucketSetValue

Examples This example gets the value in the hash bucket:

```
T_HASH_TABLE table;
T_HASH_BUCKET bucket;
T_PTR ptr;

table = TutHashCreateStr(20);
TutHashInsert(table, "EAST", (T_PTR)1);
/* look up the bucket that contains the entry for "EAST" */
bucket = TutHashLookupBucket(table, "EAST");
/* get the value in the bucket */
ptr = TutHashBucketGetValue(bucket);
```

TutHashBucketSetValue

Name

TutHashBucketSetValue — set the value in a hash bucket

Synopsis

```
T_PTR TutHashBucketSetValue(bucket, new_value)
T_HASH_BUCKET bucket;
T_PTR new_value;
```

Arguments

bucket — the hash bucket to set value for

new_value — value to be stored in the bucket

Return Values

The old value that was stored in the bucket.

Diagnostics

None

Description

TutHashBucketSetValue replaces the value stored in *bucket* with *new_value*. Before calling this function, obtain *bucket* by calling TutHashLookupBucket. If the value is successfully replaced, TutHashBucketSetValue returns the old value that was stored in the bucket.

Caution

None

See Also

TutHashBucketGetValue, TutHashLookupBucket, TutHashReplace

Examples

This example sets the hash bucket value:

```
T_HASH_TABLE table;
T_HASH_BUCKET bucket;

table = TutHashCreateStr(20);
TutHashInsert(table, "EAST", (T_PTR)1);
/* look up the bucket that contains the entry for "EAST" */
bucket = TutHashLookupBucket(table, "EAST");
/* replace the value in the bucket */
TutHashBucketSetValue(bucket, (T_PTR)2);
```

TutHashCreate

Name

TutHashCreate — create a hash table

Synopsis

```
T_HASH_TABLE TutHashCreate(init_size, hash_func, test_func)
T_INT4 init_size;
T_HASH_CALC_FUNC hash_func;
T_HASH_TEST_FUNC test_func;
```

Arguments

init_size — initial size of the hash table

hash_func — function to calculate the hash code for a key

test_func — function to compare two keys

Return Values

Returns a new hash table.

Diagnostics

None

Description

TutHashCreate creates a new hash table. A hash table consists of keys and values, where each key can be used to look up the corresponding value (hash tables have also been called symbol tables and dictionaries). Values can be inserted, deleted, and looked up very quickly. Hash tables provide a good balance between time used and storage used.

A key can be any pointer-sized (usually 4 bytes) data type (such as a character string, an integer, or any generic pointer). Each hash table normally uses the same data type for all of its keys; for example, all keys may be strings or integers, but not both. SmartSockets provides convenience functions that create hash tables with these types of keys:

- Case-sensitive strings; for example, `battery1` and `BATTERY1` are different keys
- Strings that are not case sensitive; for example, `battery1` and `bAtTeRY1` are the same key
- Integers
- Pointers

SmartSockets hash tables consist of a hash function *hash_func*, a test function *test_func*, and an array of buckets. When a value is looked up from its key, the key is hashed using *hash_func* to provide an index into the bucket array. Each bucket is a linked list of (key, value) pairs. SmartSockets hash tables use open hashing. In open hashing, different keys can hash to the same bucket. To find the proper (key, value) pair from the bucket, the bucket is then searched linearly using *test_func* to find the proper key.

One problem with hash tables is knowing how big to make the bucket array. If the array of buckets is large enough and the hash function provides a fairly random spread of bucket indices, then on average each bucket only has one (key, value) pair stored in it. This provides for constant time for each look up. If the bucket array is too large, then storage is wasted. If the bucket array is too small, then each bucket has more than one (key, value) pair, and the time for each look up increases. (Each bucket is searched using a linear search.)

To avoid wasting both storage and time, SmartSockets hash tables dynamically resize themselves if they get too full. Each time a new value is inserted into a hash table, the hash table checks how many entries it has and how large its bucket array is. If the number of entries is approaching the number of buckets, then the size of the bucket array is doubled, and the existing entries are redistributed throughout the new bucket array. Resizing never takes place on a look up, only on an insert.

The initial size of the hash table bucket array is specified by *init_size*. The initial size should be large enough to prevent numerous resizes as entries are inserted, yet small enough to prevent wasted storage.

Write the hash function *hash_func* to take one argument: the key it is hashing. *hash_func* should hash the key to a hash code and return that hash code. The array index in the bucket array is then calculated by (hash code) modulo (bucket array size). Hash functions should try to provide a wide range of hash codes.

Write the test function *test_func* to take two arguments: the two keys that are being compared. *test_func* should return TRUE if the two keys are the same, and FALSE if the two keys are different.

It is normally not necessary to write hash functions and test functions. The convenience functions TutHashCreateInt, TutHashCreatePtr, TutHashCreateStr, and TutHashCreateStri all create hash tables with predefined hash and test functions.

Hash tables do not manage storage for their keys and values. It is up to the caller to manage the memory allocation of the data structures for the keys and values.

Caution If *init_size* is too small, then inserting new entries into the hash table causes the hash table to be resized often.

See Also TutHashCreateInt, TutHashCreatePtr, TutHashCreateStr, TutHashCreateStri, TutHashDestroy

Examples This example creates a simple hash function for character strings:

```
T_INT4 hash_string(key)
T_PTR key;
{
  T_INT4 hash_code = 0;
  T_STR str_key = (T_STR)key;
  while (*str_key != '\0') {
    hash_code += *str_key;
    str_key++; /* move on to next character */
   }
   return hash_code;
} /* hash_string */
```

This example is a simple test function for character strings.

```
T_BOOL test_string(key1, key2)
T_PTR key1;
T_PTR key2;
{
  /* return TRUE if the keys are identical */
  return (strcmp((T_STR)key1, (T_STR)key2) == 0);
} /* test_string */
```

This example creates a hash table using those functions with enough room initially for 4 entries.

```
T_HASH_TABLE hash_table;
hash_table = TutHashCreate(4, hash_string, test_string);
```

Once the hash table is created, you can store values using TutHashInsert and look them up with TutHashLookup.

```
/* use the above hash table to map the directions to integers */
TutHashInsert(hash_table, "NORTH", (T_PTR)1);
TutHashInsert(hash_table, "EAST", (T_PTR)2);
TutHashInsert(hash_table, "SOUTH", (T_PTR)3);
TutHashInsert(hash_table, "WEST", (T_PTR)4);
/* look up "EAST" */
TutOut("the value for EAST is %d\n",
       (T_INT4)TutHashLookup(hash_table, "EAST"));
```

TutHashCreateInt

Name TutHashCreateInt — create a hash table with integer keys

Synopsis

```
T_HASH_TABLE TutHashCreateInt(init_size)
T_INT4 init_size;
```

Arguments *init_size* — the initial size of the hash table

Return Values Returns a new hash table suitable for integer keys.

Diagnostics None

Description TutHashCreateInt creates a new hash table with integers for keys.

Caution Providing an *init_size* that is too small causes the hash table to be resized often as new entries are inserted.

See Also TutHashCreate, TutHashCreatePtr, TutHashCreateStr, TutHashCreateStri

Examples This example uses a hash table to look up string values from integer keys. Integer keys should be cast to (T_PTR) to avoid warnings from C compilers that understand function prototypes.

```
T_HASH_TABLE hash_table;

hash_table = TutHashCreateInt(4);
/* map integer directions to direction strings */
TutHashInsert(hash_table, (T_PTR)1, "NORTH");
TutHashInsert(hash_table, (T_PTR)2, "EAST");
TutHashInsert(hash_table, (T_PTR)3, "SOUTH");
TutHashInsert(hash_table, (T_PTR)4, "WEST");

TutOut("The value for 3 is %s.\n",
        TutHashLookup(hash_table, (T_PTR)3));
```

This example is the hash function and test function used by TutHashCreateInt:

```
/* ============================================================ */
/*..utHashCalcInt -- hash function for ints */
static T_INT4 utHashCalcInt(key)
T_PTR key;
{
  return *(T_INT4 *)key;
} /* utHashCalcInt */
```

```
/* ============================================================ */
/*..utHashTestInt -- test function for ints */
static T_BOOL utHashTestInt(key1, key2)
T_PTR key1; /* really an int */
T_PTR key2; /* really an int */
{
  return *(T_INT4 *)key1 == *(T_INT4 *)key2;
} /* utHashTestInt */
```

This example creates the hash table:

```
/* ============================================================ */
/*..TutHashCreateInt -- convenience func to create hash table with int keys */
T_HASH_TABLE TutHashCreateInt(init_size)
T_INT4 init_size; /* initial size of hash table */
{
  return TutHashCreate(init_size, utHashCalcInt, utHashTestInt);
} /* TutHashCreateInt */
```

TutHashCreatePtr

Name TutHashCreatePtr — create a hash table with pointer keys

Synopsis

```
T_HASH_TABLE TutHashCreatePtr(init_size)
T_INT4 init_size;
```

Arguments *init_size* — the initial size of the hash table

Return Values Returns a new hash table suitable for pointer keys.

Diagnostics None

Description TutHashCreatePtr creates a new hash table with pointers for keys. Dynamically allocated pointers, such as those returned by TutMalloc, have the property of being suitably aligned to store any data object (such as integers, floats or doubles). These pointers, when cast to long integers, are multiples of 16 on most system architectures. Therefore the hash function can simply divide the pointer key by 16 to produce a nice tight spread of hash codes.

Caution Providing an *init_size* that is too small causes the hash table to be resized often as new entries are inserted.

See Also TutHashCreate, TutHashCreateInt, TutHashCreateStr, TutHashCreateStri

Examples This example uses a hash table to look up file names from open files:

```
T_HASH_TABLE hash_table;
FILE *file1;
FILE *file2;

hash_table = TutHashCreatePtr(4);
/* map file pointers to file name strings */
file1 = TutFOpen("foo.tmp", "w");
if (file1 != NULL) {
  TutHashInsert(hash_table, file1, "foo.tmp");
}
file2 = TutFOpen("bar.tmp", "w");
if (file2 != NULL) {
  TutHashInsert(hash_table, file2, "bar.tmp");
}

TutOut("The value for file %x is %s.\n",
        file1,
        TutHashLookup(hash_table, file1));
```

This example is the hash function and test function used by TutHashCreatePtr:

```
/* ============================================================ */
/*..utHashCalcPtr -- hash function for (malloced) pointers */
static T_INT4 utHashCalcPtr(key)
T_PTR key;
{
  return (T_UINT4)key >> 4; /* divide by 16 */
} /* utHashCalcPtr */

/* ============================================================ */
/*..utHashTestPtr -- test function for (malloced) pointers */
static T_BOOL utHashTestPtr(key1, key2)
T_PTR key1;
T_PTR key2;
{
   return key1 == key2;
} /* utHashTestPtr */
```

This example creates the hash table:

```
/* ============================================================ */
/*..TutHashCreatePtr -- convenience func to create hash table with ptr keys */
T_HASH_TABLE TutHashCreatePtr(init_size)
T_INT4 init_size; /* initial size of hash table */
{
  return TutHashCreate(init_size, utHashCalcPtr, utHashTestPtr);
} /* TutHashCreatePtr */
```

TutHashCreateStr

Name TutHashCreateStr — create a hash table with case-sensitive string keys

Synopsis

```
T_HASH_TABLE TutHashCreateStr(init_size)
T_INT4 init_size;
```

Arguments *init_size* — the initial size of the hash table

Return Values Returns a new hash table suitable for case-sensitive string keys.

Diagnostics None

Description TutHashCreateStr creates a new hash table with case-sensitive strings for keys.

Caution Providing an *init_size* that is too small causes the hash table to be resized often as new entries are inserted.

See Also TutHashCreate, TutHashCreateInt, TutHashCreatePtr, TutHashCreateStri

Examples This example uses a hash table to look up file pointers from file names:

```
T_HASH_TABLE hash_table;
FILE *file;

hash_table = TutHashCreateStr(4);
file = TutFOpen("foo.tmp", "w");
if (file != NULL) {
  TutHashInsert(hash_table, "foo.tmp", file);
}
file = TutFOpen("bar.tmp", "w");
if (file != NULL) {
  TutHashInsert(hash_table, "bar.tmp", file);
}
/* this will print foo.tmp's file pointer in hex. */
TutOut("The value for foo.tmp is %x.\n",
        TutHashLookup(hash_table, "foo.tmp"));
/* this will print bar.tmp's file pointer in hex. */
TutOut("The value for bar.tmp is %x.\n",
        TutHashLookup(hash_table, "bar.tmp"));
/* this will print 0 because baz.tmp isn't in the table */
TutOut("The value for baz.tmp is %x.\n",
        TutHashLookup(hash_table, "baz.tmp"));
```

TutHashCreateStri

Name TutHashCreateStri — create a hash table with string keys that are not case sensitive

Synopsis

```
T_HASH_TABLE TutHashCreateStri(init_size)
T_INT4 init_size;
```

Arguments *init_size* — the initial size of the hash table

Return Values Returns a new hash table suitable for string keys that are not case sensitive.

Diagnostics None

Description TutHashCreateStri creates a new hash table with string for keys that are not case sensitive. For example, `battery1` and `BATTERY1` are treated as the same key.

Caution Providing the initial size that is too small causes the hash table to be resized often as new entries are inserted.

See Also TutHashCreate, TutHashCreateInt, TutHashCreatePtr, TutHashCreateStr

Examples This example uses a hash table to look up file pointers from file names:

```
T_HASH_TABLE hash_table;
FILE *file;

str_hash_table = TutHashCreateStri(4);
file = TutFOpen("foo.tmp", "w");
if (file != NULL) {
  TutHashInsert(hash_table, "foo.tmp", file);
}

/* this will print foo.tmp's file pointer in hex. */
TutOut("The value for foo.tmp is %x.\n",
        TutHashLookup(hash_table, "foo.tmp"));

/* this will print FOO.TMP's file pointer in hex (same as foo.tmp). */
TutOut("The value for FOO.TMP is %x.\n",
        TutHashLookup(hash_table, "FOO.TMP"));

/* this will print Foo.Tmp's file pointer in hex (same as foo.tmp). */
TutOut("The value for Foo.Tmp is %x.\n",
        TutHashLookup(hash_table, "Foo.Tmp"));
```

TutHashDelete

Name TutHashDelete — delete a hash table entry by key

Synopsis

```
T_PTR TutHashDelete(table, key)
T_HASH_TABLE table;
T_PTR key;
```

Arguments *table* — hash table to delete entry from

key — key used to look up value to be deleted

Return Values TutHashDelete returns the value that was deleted from the hash table, or NULL if the value could not be found.

Diagnostics None

Description TutHashDelete deletes an entry from *table*. If an entry is found with *key*, it is deleted from *table* and the entry is returned. If the entry is not found, NULL is returned.

Caution Always verify the result of this function. After the deletion, you must free the space occupied by the deleted entry and key. If the key is stored as part of the entry, a single free on the returned pointer is enough to free the space. Otherwise, you need two frees: one for the deleted entry and one for the deleted key.

See Also TutHashInsert, TutHashBucketSetValue

Examples This example creates a hash table with string keys, adds and deletes items:

```
typedef struct entry {
  T_STR name;
  T_PTR addr;
} ENTRY;

T_PTR print_entry(key, value, arg)
T_PTR key;
T_PTR value;
T_PTR arg;
{
  TutOut("Name <%s> has address <%s>\n", key, ((ENTRY
*)value)->addr);
   return NULL;
}
int main()
{
  T_HASH_TABLE table;
  ENTRY *e;
  ENTRY *tofree;
```

```
/* Initialize the hash table */
  table = TutHashCreateStr(4);

  /* Allocate one entry */
    e = TutMalloc(sizeof(struct entry));
    e->name = TutMalloc(25);
    e->addr = TutMalloc(50);

  /* Set some values in the entry */
    strcpy(e->name, "Johnson");
    strcpy(e->addr, "1234 Mulberry Lane");

 /* Insert the entry into the table */
    TutHashInsert(table, e->name, e);

  /* Allocate the second entry */
    e = TutMalloc(sizeof(ENTRY));
    e->name = TutMalloc(25);
    e->addr = TutMalloc(50);

/* Set some values in the second entry */
    strcpy(e->name, "Jones");
    strcpy(e->addr, "10 Downing Street");

 /* Insert the second entry into the table */
    TutHashInsert(table, e->name, e);

 /* Echo the whole table */
    TutOut("\nFirst time:\n");
    TutHashTraverse(table, print_entry, NULL);

 /* Delete one entry */
    tofree = TutHashDelete(table, "Johnson");
    if (tofree != NULL) {
      TutOut("Deleted entry for <%s>\n", tofree->name);
      TutFree(tofree->name);
      TutFree(tofree->addr);
      TutFree(tofree);
   }
 /* Echo the whole table again */
    TutOut("\nSecond time:\n");
    TutHashTraverse(table, print_entry, (T_PTR)NULL);
} /* main */
```

This example produces this output:

```
First time:
Name <Jones> has address <10 Downing Street>
Name <Johnson> has address <1234 Mulberry Lane>
Deleted entry for <Johnson>

Second time:
Name <Jones> has address <10 Downing Street>
```

It is your responsibility to allocate space for the key and for the entry to be inserted before calling TutHashInsert.

TutHashDestroy

Name TutHashDestroy — deallocate the memory used by a hash table and destroy the table

Synopsis

```
void TutHashDestroy(table, destroy_func, destroy_arg)
T_HASH_TABLE table;
T_HASH_DESTROY_FUNC destroy_func;
T_PTR destroy_arg;
```

Arguments

table — hash table to be destroyed

destroy_func — user-defined function that gets called for every entry in the table (or `NULL` to signify no user-defined deallocation is necessary)

destroy_arg — the argument that gets passed to *destroy_func*

Return Values None

Diagnostics None

Description TutHashDestroy destroys a hash table. This procedure deallocates all the table structures created by the call to TutHashCreate and the related functions. It is up to you to deallocate the space occupied by the entries stored in the table and their keys. The deallocation of the entries can be accomplished either by repetitive calls to TutHashDelete for every entry that was inserted into the table, or by specifying the destroy function that gets called for every entry still in the table before the actual table deletion. The destroy function is used to deallocate the space occupied by the keys and entries that were inserted into the table.

This function should be defined as `void` and have three arguments. All the arguments should be of type T_PTR. The first argument is the key to the entry, the second is the entry itself, and the third is the argument that was passed to TutHashDestroy. Inside the function, void pointers can be recast to the appropriate types to perform the deallocation correctly.

Caution Always provide the destroy function, because it is the best and safest way to deallocate the memory occupied by hash table entries and their keys.

See Also TutHashCreate

Examples This example destroys a hash table with TutHashDestroy:

```
#define NUM_ITEMS 50
typedef struct mods MODS;
struct mods {
  T_STRING name;
  T_INT4 number;
};

T_HASH_TABLE mod_table;

void destroy_mods(key, mod_rec, arg)
T_PTR key;
T_PTR mod_rec;
T_PTR arg;
{
  MODS modrec = mod_rec;
  TutOut("Freeing memory for record number %d.", modrec->number);
  T_FREE(mod_rec);
} /* destroy_mods */

int main()
{
  T_INT4 i;
  MODS *mod_rec;

  /* create the hash table with NUM_ITEMS entries and string keys */
  mod_table = TutHashCreateStri(NUM_ITEMS);

  /* fill the table with data */
  for (i = 0; i < NUM_ITEMS; i++) {
 /* allocate the memory for the data */
    T_MALLOC(mod_rec, sizeof(MODS), MODS *);

    mod_rec->number = i;
    sprintf(mod_rec->name, "Item_number_%d", i);

    /* insert the record into the table */
    TutHashInsert(mod_table, mod_rec->name, (T_PTR)mod_rec);
  }

  /* Perform the necessary processing */
  ...

  /* destroy the hash table and deallocate the memory */
  TutHashDestroy(mod_table, destroy_mods, NULL);
} /* main */
```

TutHashDump

Name

TutHashDump — dump the information about a hash table

Synopsis

```
void TutHashDump(table, output_func)
T_HASH_TABLE table;
T_OUT_FUNC output_func;
```

Arguments

table — table to be dumped

output_func — function used to print the data

Return Values

None

Diagnostics

None

Description

TutHashDump prints statistical information about the specified hash table, such as size, number of entries, maximum number of entries, number of collisions, and some bucket statistics. It takes two arguments as input. *table* is a hash table to be dumped. *output_func* is an output function that is used to print the hash table information. The output function should accept arguments in the same format as a printf statement. If *output_func* is `NULL`, an internal function is used that sends the data to stderr.

Caution

None

See Also

None

Examples This example creates a hash table, inserts two entries, then dumps the table:

```
int main()
{
  T_HASH_TABLE table;

  table = TutHashCreateStri(4);
  TutHashInsert(table, "hi", (T_PTR)1);
  TutHashInsert(table, "there", (T_PTR)2);
  TutHashDump(table, TutOut);
  TutExit(T_EXIT_SUCCESS);
}
```

This is a sample of the output:

```
*** Dumping Hash Table ***
 Size = 8, Num Entries = 2, Max Entries = 6, Collisions = 0
 All Buckets: Avg Size = 0.250000, Std Dev = 1.224745
 Non-Empty Buckets (2): Avg Size = 1.000000, Std Dev = 1.224745
 Bucket 0 has 0 entries
 Bucket 1 has 1 entries
 hi 1
 Bucket 2 has 0 entries
 Bucket 3 has 0 entries
 Bucket 4 has 1 entries
 there 2
 Bucket 5 has 0 entries
 Bucket 6 has 0 entries
 Bucket 7 has 0 entries
```

TutHashGetSize

Name TutHashGetSize — return the number of entries in the hash table

Synopsis

```
T_INT4 TutHashGetSize(table)
T_HASH_TABLE table;
```

Arguments *table* — hash table to get number of entries from

Return Values Number of entries in the hash table

Diagnostics None

Description TutHashGetSize returns the number of entries in *table*. Because the count is maintained as part of the hash table, this function accesses that information rather than counting the entries in the table.

Caution None

See Also None

Examples This example gets the number of entries in the hash table:

```
table = TutHashCreateStr(20);
TutOut("size = %d\n", TutHashGetSize(table)); /* prints 0 */
TutHashInsert(table, "hi", "there");
TutOut("size = %d\n", TutHashGetSize(table)); /* prints 1 */
```

TutHashInsert

Name TutHashInsert — insert an entry into the hash table

Synopsis

```
void TutHashInsert(table, key, value)
T_HASH_TABLE table;
T_PTR key;
T_PTR value;
```

Arguments

table — table where entry is inserted

key — the value that is used as a key to the entry

value — the entry that is stored in the table

Return Values None

Diagnostics None

Description TutHashInsert inserts an entry into *table*. The type of *key* should be consistent with the key type specified when *table* was created.

Caution Hash tables do not manage storage for their keys and values. It is up to the caller to manage the memory allocation of the data structures for the keys and values.

value cannot be a null pointer.

See Also TutHashCreate, TutHashDelete, TutHashLookup, TutHashBucketSetValue

Examples This example adds an entry to the hash table:

```
typedef struct entry {
  T_STR name;
  T_PTR addr;
} ENTRY;

T_PTR print_entry(key, value, arg)
T_PTR key;
T_PTR value;
T_PTR arg;
{
  TutOut("Name <%s> has address <%s>\n", key,
         ((ENTRY *)value)->addr);
  return NULL;
}

main()
{
  T_HASH_TABLE table;
  ENTRY *e;

  /* Initialize the hash table */
  table = TutHashCreateStr(4);

  /* Allocate one entry */
  e = TutMalloc(sizeof(ENTRY));
  e->name = TutMalloc(25);
  e->addr = TutMalloc(50);

  /* Set the values in the entry */
  strcpy(e->name, "Johnson");
  strcpy(e->addr, "1234 Mulberry Lane");

  /* Insert the entry into the table */
  TutHashInsert(table, e->name, e);

  /* Echo the whole table */
  TutHashTraverse(table, print_entry, NULL);
}
```

The example produces this output:

```
Name <Johnson> has address <1234 Mulberry Lane>
```

TutHashLookup

Name

TutHashLookup — find an entry in the hash table

Synopsis

```
T_PTR TutHashLookup(table, key)
T_HASH_TABLE table;
T_PTR key;
```

Arguments

table — table where the entry should be looked for

key — the key to use to find the entry

Return Values

The entry if it was located, NULL otherwise.

Diagnostics

If TutHashLookup fails, it returns NULL and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *table* was NULL
- T_ERR_DOESNT_EXIST — an entry with the desired *key* does not exist in *table*

Description

TutHashLookup returns the entry that was previously stored in *table*. *key* is used to locate the entry. If the entry is not found, a null pointer is returned. The type of *key* should be consistent with the key type specified when the hash table was created.

Caution

None

See Also

TutHashCreate, TutHashInsert, TutHashTraverse

Examples

This example looks up an entry in the hash table:

```
T_HASH_TABLE table;
table = TutHashCreateStr(20);
/* this will print 0 because look up will fail */
TutOut("look up of EAST is %d\n",
        (T_INT4)TutHashLookup(table, "EAST"));
/* insert a value */
TutHashInsert(table, "EAST", (T_PTR)1);
/* this will print 1 *TutOut("look up of EAST is %d\n",
        (T_INT4)TutHashLookup(table, "EAST"));
```

TutHashLookupBucket

Name

TutHashLookupBucket — look up a bucket in a hash table

Synopsis

```
T_HASH_BUCKET TutHashLookupBucket(table, key)
T_HASH_TABLE table;
T_PTR key;
```

Arguments

table — the hash table that should be searched for the bucket

key — key to the bucket

Return Values

The bucket with a given key if successful, NULL otherwise.

Diagnostics

If TutHashLookupBucket fails, it returns NULL and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *table* was NULL
- T_ERR_DOESNT_EXIST — an entry with the desired *key* does not exist in *table*

Description

TutHashLookupBucket returns the hash bucket that matches a given key. If it cannot find the appropriate bucket, it returns NULL. This function may be used to speed up replacement of values in the table.

Caution

None

See Also

TutHashLookup, TutHashInsert, TutHashBucketSetValue

Examples

This example gets the hash bucket for a key:

```
T_HASH_TABLE table;
T_HASH_BUCKET bucket;

table = TutHashCreateStr(20);
TutHashInsert(table, "EAST", (T_PTR)1);
/* look up the bucket that contains the entry for "EAST" */
bucket = TutHashLookupBucket(table, "EAST");
/* replace the value in the bucket */
TutHashBucketSetValue(bucket, (T_PTR)2);
```

TutHashReplace

Name TutHashReplace — replace a value stored in the hash table

Synopsis

```
T_PTR TutHashReplace(table, key, new_value)
T_HASH_TABLE table;
T_PTR key;
T_PTR new_value;
```

Arguments *table* — hash table where the value should be replaced

key — the key to the value

new_value — the new value that is stored in the table

Return Values The old value if successful, or NULL if no old value exists.

Diagnostics None

Description TutHashReplace replaces the existing value in the hash table that matches the *key* with *new_value*. If the existing value cannot be located, the *new_value* is inserted in the table. In that case, the call to TutHashReplace is equivalent to a call to TutHashInsert.

After replacing, it is up to you to deallocate the space occupied by the replaced value.

Caution Hash tables do not manage storage for their keys and values. It is up to the caller to manage the memory allocation of the data structures for the keys and values.

new_value cannot be a null pointer.

See Also TutHashInsert

Examples This example replaces a value in the hash bucket:

```
typedef struct entry {
 T_STR name;
 T_PTR addr;
} ENTRY;

T_PTR print_entry(key, value, arg)
T_PTR key;
T_PTR value;
T_PTR arg;
{
   TutOut("Name <%s> has address <%s>\n", key, ((ENTRY
*)value)->addr);
   return NULL;
}
```

```
int main()
{
   T_HASH_TABLE table;
   ENTRY *e;
   ENTRY *tofree;

  /* Initialize the hash table */
   table = TutHashCreateStr(4);

  /* Allocate one entry */
   e = TutMalloc(sizeof(ENTRY));
   e->name = TutMalloc(25);
   e->addr = TutMalloc(50);

  /* Set some values in the entry */
   strcpy(e->name, "Johnson");
   strcpy(e->addr, "1234 Mulberry Lane");

  /* Insert the entry into the table */
   TutHashInsert(table, e->name, e);

   /* Allocate the second entry */
   e = TutMalloc(sizeof(ENTRY);
   e->name = TutMalloc(25);
   e->addr = TutMalloc(50);

 /* Set some values in the second entry */
   strcpy(e->name, "Jones");
   strcpy(e->addr, "10 Downing Street");

  /* Insert the second entry into the table */
   TutHashInsert(table, e->name, e);

 /* Echo the whole table */
   TutOut("\nFirst time:\n");
   TutHashTraverse(table, print_entry, NULL);

 /* Allocate a replacement entry */
   e = TutMalloc(sizeof(ENTRY));
   e->name = TutMalloc(25);
   e->addr = TutMalloc(50);

  /* Set some values in the replacement entry */
   strcpy(e->name, "Johnson");
   strcpy(e->addr, "555 Main Street");
```

```
  /* Replace the old entry with the replacement entry */
    tofree = TutHashReplace(table, e->name, e);
    if (tofree != NULL) {
      TutOut("Replaced <%s> with <%s> for <%s>\n",
             tofree->addr, e->addr, e->name);
      TutFree(tofree->name);
      TutFree(tofree->addr);
      TutFree(tofree);
    }

  /* Echo the whole table again */
    TutOut("\nSecond time:\n");
    TutHashTraverse(table, print_entry, NULL);
}
```

The example produces this output:

```
First time:
Name <Jones> has address <10 Downing Street>
Name <Johnson> has address <1234 Mulberry Lane>
Replaced <1234 Mulberry Lane> with <555 Main Street> for <Johnson>

Second time:
Name <Jones> has address <10 Downing Street>
Name <Johnson> has address <555 Main Street>
```

TutHashSort

Name

TutHashSort — return an array of sorted values from a hash table

Synopsis

```
T_PTR TutHashSort(table, array_size_return, cmp_func, select_func)
T_HASH_TABLE table;
T_INT4 *array_size_return;
T_HASH_SORT_CMP_FUNC cmp_func;
T_HASH_SORT_SELECT_FUNC select_func;
```

Arguments

table — hash table to be searched for entries.

array_size_return — pointer to integer where TutHashSort stores the number of values in the list after the sort.

cmp_func — user-defined function used to compare keys while sorting. If NULL is specified, the default function strcmp is used to compare keys.

select_func — user-defined function used to qualify keys for the sort. If NULL is specified, all entries in the table are sorted.

Return Values

Array of sorted values in the table, selected according to the criteria imposed by *select_func*.

Diagnostics

None

Description

TutHashSort returns a sorted array of values from the *table*. The entries are selected based on the criteria provided in *select_func*. If *select_func* is NULL, all entries in the hash table are sorted and returned. For comparison, a user-defined comparison function is used. If *cmp_func* is NULL, strcmp is used to compare the entries. Using different combinations of select and comparison functions produces lists of entries sorted in different order and selected according to various criteria. Note that the sort in no way affects the order in which entries are stored in the hash table.

cmp_func takes as arguments two T_PTR keys to entries in the table. The first argument holds a key to an already sorted entry. The second argument is the key to the current entry. If the default comparison function strcmp is used, the returned list is sorted in the ascending order. The comparison function should have return values equivalent to that of the strcmp function.

select_func takes as arguments the T_PTR key and value for the entry into the hash table. It returns TRUE if the entry satisfies the selection criteria, otherwise it returns FALSE.

The return value from TutHashSort is a dynamically allocated array of T_PTR-sized hash table entries. The return value is of type T_PTR instead of T_PTR * so that no cast is needed by the calling function.

Caution TutHashSort allocates the space for the sorted array of entries. It is up to you to free that memory.

If TutHashSort does not select any entries from the table, it returns NULL.

See Also None

Examples This example returns a sorted array of hash table entries:

```
T_HASH_TABLE table;
T_PTR *values; /* each pointer really contains an integer value */
T_INT4 counter;
T_INT4 num_values;

/* create a hash table and put 4 integer values in it */
table = TutHashCreateStr(4);
TutHashInsert(table, "EAST", (T_PTR)90);
TutHashInsert(table, "SOUTH", (T_PTR)180);
TutHashInsert(table, "WEST", (T_PTR)270);
TutHashInsert(table, "NORTH", (T_PTR)360);

/* sort the table and print the values */
values = TutHashSort(table, &num_values, NULL, NULL);
TutOut("Sorted %d entries.\n", num_values);
for (counter = 0; counter < num_values; counter++) {
  TutOut("Value[%d] = %d\n", counter, (int)values[counter]);
}
T_FREE(values);
```

TutHashTraverse

Name TutHashTraverse — traverse all entries in a hash table

Synopsis

```
T_PTR TutHashTraverse(table, traverse_func, traverse_arg)
T_HASH_TABLE table;
T_HASH_TRAV_FUNC traverse_func;
T_PTR traverse_arg;
```

Arguments

table — hash table to traverse

traverse_func — function to call once for each entry in the table

traverse_arg — argument to pass to *traverse_func*

Return Values The first non-null return value from *traverse_func*, or NULL if *traverse_func* always returns NULL.

Diagnostics If TutHashTraverse fails or traverses all entries in the table, it returns NULL and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *table* was NULL or *traverse_func* was NULL
- T_ERR_END_OF_TRAVERSAL — all entries in *table* were traversed

Description TutHashTraverse traverses all entries in a hash table and calls *traverse_func* once for each entry. The *traverse_func* function is called with three arguments: the *key*, the *value*, and the *traverse_arg* that TutHashTraverse was called with. When *traverse_func* returns a non-null value, then TutHashTraverse returns with that value.

There are two common uses for TutHashTraverse: to perform an operation on each entry and to search for a particular entry. In the first case, write *traverse_func* to always return NULL. In the second case, *traverse_func* should return NULL until it finds the desired entry, and then return some kind of non-null value (such as the entry value).

Caution Be careful to always return a value from *traverse_func*. The order that the entries are traversed is indeterminate. Do not add or delete entries from the hash table during the traversal.

See Also TutHashDump, TutHashLookup

Examples This example is a traversal function that prints the values of `key`, `value`, and `arg`:

```
T_PTR print_info(key, value, arg)
T_PTR key;
T_PTR value;
T_PTR arg;
{
  TutOut("key = %x, value = %x, arg = %x\n", key, value, arg);
  return NULL; /* continue traversal */
} /* print_info */
```

To call this function, this code could be used:

```
TutHashTraverse(table, print_info, NULL);
```

Chapter 8 TutMalloc - TutOutFlush

This chapter describes these SmartSockets utility functions:

- *TutMalloc*
- *TutMutexCreate*
- *TutMutexCreateFast*
- *TutMutexDestroy*
- *TutMutexLock*
- *TutMutexLockFast*
- *TutMutexUnlock*
- *TutOptionChangeCbCreate*
- *TutOptionChangeCbLookup*
- *TutOptionCreate*
- *TutOptionDestroy*
- *TutOptionGetBool*
- *TutOptionGetEnum*
- *TutOptionGetEnumList*
- *TutOptionGetKnown*
- *TutOptionGetName*
- *TutOptionGetNum*
- *TutOptionGetReadOnly*
- *TutOptionGetRequired*
- *TutOptionGetStr*
- *TutOptionGetStrList*
- *TutOptionGetType*
- *TutOptionGetVerifyFunc*
- *TutOptionLegValAdd*
- *TutOptionLookup*

- *TutOptionNamedLookup*
- *TutOptionSetBool*
- *TutOptionSetEnum*
- *TutOptionSetEnumList*
- *TutOptionSetNum*
- *TutOptionSetReadOnly*
- *TutOptionSetRequired*
- *TutOptionSetStr*
- *TutOptionSetStrList*
- *TutOptionSetUnknown*
- *TutOptionSetVerifyFunc*
- *TutOut*
- *TutOutFlush*

TutMalloc

Name

TutMalloc — allocate a block of memory

Synopsis

```
T_PTR TutMalloc(size)
T_UINT4 size;
```

Arguments

size — number of bytes to allocate

Return Values

Pointer to the allocated memory block if successful; `NULL` if the requested memory could not be allocated.

Diagnostics

None

Description

TutMalloc allocates a block of memory of the requested size. Using this function in the code ensures the portability to all the platforms that are currently supported by SmartSockets.

TutMalloc allows *size* to be 0; this is useful for allocating a non-null placeholder that is resized later with TutRealloc.

All SmartSockets memory management uses the functions TutCalloc, TutFree, TutMalloc, and TutRealloc. You can set debugger breakpoints in these functions to trace SmartSockets memory usage.

Caution

TutCalloc, TutFree, TutMalloc, and TutRealloc are not compatible with other memory management functions. Memory allocated by SmartSockets (including memory allocated by TutCalloc, TutMalloc, or TutRealloc) must be reallocated with TutRealloc and freed with TutFree. Attempting to mix memory management functions will result in unpredictable and potentially unstable behavior.

See Also

TutFree, TutCalloc, TutRealloc

Examples

This example allocates and frees a string 30 characters long:

```
T_STR strptr;

/* the string has to be 30 characters long, so one space has to be */
/* reserved for the terminating NULL */
strptr = (T_STR)TutMalloc(31);
...
TutFree(strptr);
```

TutMutexCreate

Name

TutMutexCreate — create a mutex object

Synopsis

```
T_MUTEX TutMutexCreate(initial_state)
T_BOOL initial_state;
```

Arguments

initial_state — `TRUE` if mutex is to be initially locked

Return Values

Mutex object if successful, `T_INVALID_MUTEX` otherwise.

Diagnostics

If TutMutexCreate fails, it returns `T_INVALID_MUTEX` and sets the global SmartSockets error number to one of:

- T_ERR_OS — an operating system error has occurred. Call the function TutErrNumGetOs to obtain the related OS error value.
- T_ERR_TYPE_MISMATCH — the platform does not support threads.
- T_ERR_VAL_INVALID — *initial_state* was not a valid boolean value.

Description

TutMutexCreate creates a mutual exclusion synchronization (mutex) object. Mutexes provide the mechanism by which a thread can secure exclusive access to a shared resource. All threads that wish to use the resource must first obey the convention of calling TutMutexLock with the associated mutex before accessing the resource and TutMutexUnlock with the mutex when they are finished.

The mutexes implemented by this API are recursive. This means that a thread that has locked a mutex can safely lock the same mutex again without deadlocking. The thread continues to own the mutex until each of its calls to TutMutexLock has been balanced by a call to TutMutexUnlock.

Destroy the mutex object when it is no longer needed by calling TutMutexDestroy.

Caution

None

See Also

TutMutexDestroy, TutMutexLock, TutMutexUnlock

Examples This example creates a mutex, locks it around a protected region of code, and then destroys the mutex:

```
T_MUTEX mutex;
T_BOOL status;

mutex = TutMutexCreate(FALSE);
if (T_INVALID_MUTEX == mutex) {
  /* error */
}

/* lock the mutex */
if (!TutMutexLock(mutex, T_TIMEOUT_FOREVER)) {
  /* error */
}

...

/* unlock the mutex */
if (!TutMutexUnlock(mutex)) {
  /* error */
}

if (!TutMutexDestroy(mutex)) {
  /* error */
}
```

TutMutexCreateFast

Name TutMutexCreateFast — create a high-performance mutex object

Synopsis `T_MUTEX TutMutexCreateFast(`*initial_state*`)`

Arguments *initial_state* — TRUE if mutex is to be initially locked

Return Values Mutex object if successful, T_INVALID_MUTEX otherwise.

Diagnostics If TutMutexCreateFast fails, it returns T_INVALID_MUTEX, and sets the global C error number.

Description TutMutexCreateFast creates a mutex object much the same as TutMutexCreate does, but with several important differences:

- TutMutexCreateFast mutexes are much faster than TutMutexCreate objects (approximately twice as fast)
- they are non-recursive
- they cannot use arbitrary timeout values with calls to TutMutexLock

To lock these mutex objects, call TutMutexLockFast or TutMutexLock with the special timeout value of `0.0`. TutMutexLockFast blocks until the mutex can be acquired. TutMutexLock with timeout of `0.0` makes a single attempt to acquire the mutex, and if it is not available, returns FALSE without blocking.

If your application needs recursive mutexes, use those created by TutMutexCreate. If speed is of primary importance, TutMutexCreateFast mutex objects may be a better choice.

Caution Mutex objects created with TutMutexCreateFast cannot be recursively locked; attempts to do so result in deadlock. Ensure that only the thread owning the lock on a particular "fast" mutex attempts to unlock it. Unlocking a mutex held by another thread causes unpredictable behavior.

See Also TutMutexCreate, TutMutexLock, TutMutexLockFast, TutMutexUnlock, TutMutexDestroy

Examples This code creates several threads which all update global data, using a fast mutex for synchronization:

```
#include <rtworks/common.h>
#include <rtworks/util.h>

#define NTHREADS (4)
#define DURATION (5.0)

T_MUTEX fast_mutex;
T_INT4 counter = 0;

T_PTR thread_func( T_PTR arg )
{
  T_REAL8 end;
  T_INT4 local_hits = 0;
  T_BOOL lock;

  TutOut("Thread %d starting...\n", (T_INT4)arg);
  end = TutGetWallTime() + DURATION;
  while (TutGetWallTime() < end) {
    /* do some calculation, I/O, etc. */
    lock = TutMutexLockFast(fast_mutex);
    if (!lock) {
      TutOut("Could not lock mutex!\n");
      break;
    }
    ++counter;
    ++local_hits;
    if (!TutMutexUnlock(fast_mutex)) {
      TutOut("Could not unlock mutex!\n");
      break;
    }
    TutSleep(0.1);
  }
  TutThreadExit((T_PTR)local_hits);
} /* thread_func */
```

```
int main( void )
{
  T_THREAD threads[NTHREADS];
  T_INT4 count;

  fast_mutex = TutMutexCreateFast(TRUE);
  if (T_INVALID_MUTEX == fast_mutex) {
    TutOut("Cannot create fast mutex!\n");
    TutExit(1);
  }
  TutOut("Creating %d threads...\n", NTHREADS);
  for (count=0; count<NTHREADS; count++) {
    threads[count] = TutThreadCreate(thread_func, (T_PTR)(count+1),
                                     NULL);
    if (TutThreadEqual(threads[count], T_INVALID_THREAD)) {
      TutOut("Can't start thread %d!\n", count);
      TutExit(2);
    }
  }
  TutOut("Releasing threads...\n");
  TutMutexUnlock(fast_mutex);

  for (count=0; count<NTHREADS; count++) {
    T_PTR result;

    if (TutThreadWait(threads[count], &result)) {
      TutOut("Thread %d exited, contributing %d hits.\n", count+1,
             (T_INT4)result);
    }
  }
  TutOut("In total, the counter was hit %d times.\n", counter);
} /* main */
```

TutMutexDestroy

Name TutMutexDestroy — destroy a mutex object

Synopsis

```
T_BOOL TutMutexDestroy(mutex)
T_MUTEX mutex;
```

Arguments *mutex* — mutex object to destroy

Return Values TRUE if mutex object was successfully destroyed, FALSE otherwise.

Diagnostics If TutMutexDestroy fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_OS — an operating system error has occurred. Call the function TutErrNumGetOs to obtain the related OS error value.
- T_ERR_TYPE_MISMATCH — the platform does not support threads.
- T_ERR_VAL_INVALID — *mutex* was not a valid mutex object.

Description TutMutexDestroy destroys a mutex object. All operating system resources associated with the mutex are released.

Caution Destroying a mutex that other threads in the same process have locked or are waiting to lock has undefined results.

See Also TutMutexCreate

Examples This example creates and then destroys a mutex object:

```
T_MUTEX mutex;
mutex = TutMutexCreate(FALSE);
if (T_INVALID_MUTEX == mutex) {
  /* error */
}
...
if (!TutMutexDestroy(mutex)) {
  /* error */
}
```

TutMutexLock

Name

TutMutexLock — lock a mutex object

Synopsis

```
T_BOOL TutMutexLock(mutex, timeout)
T_MUTEX mutex;
T_REAL8 timeout;
```

Arguments

mutex — mutex object to lock

timeout — maximum number of seconds to wait for mutex

Return Values

TRUE if mutex object was successfully locked, FALSE otherwise.

Diagnostics

If TutMutexLock fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_OS — call an operating system to obtain the related OS error value.
- T_ERR_TIMEOUT_REACHED — the *timeout* period was reached.
- T_ERR_TYPE_MISMATCH — the platform does not support threads.
- T_ERR_VAL_INVALID — *mutex* was not a valid mutex object.

Description

TutMutexLock attempts to lock a mutex object. If *mutex* is not immediately available, the calling thread is suspended until the mutex becomes available or until the *timeout* period specified has expired. The reserved *timeout* value T_TIMEOUT_FOREVER may be used to specify an indefinite wait.

Caution

Every call to TutMutexLock should be balanced by a subsequent call to TutMutexUnlock.

See Also

TutMutexCreate, TutMutexUnlock

Examples

This example attempts to lock a mutex object with a one-second timeout:

```
T_MUTEX mutex;
...
if (!TutMutexLock(mutex, 1.0)) {
  if (T_ERR_TIMEOUT_REACHED != TutErrNumGet()) {
    /* error */
  }
  /* timeout */
  ...
}
...
```

TutMutexLockFast

Name TutMutexLockFast — lock a high-performance mutex object

Synopsis

```
T_BOOL TutMutexLockFast(mutex)
T_MUTEX mutex;
```

Arguments *mutex* — the mutex object to lock

Return Values TRUE if the mutex object was successfully locked, FALSE otherwise.

Diagnostics If TutMutexLockFast fails, it returns FALSE, and sets the global C error number.

Description TutMutexLockFast performs a blocking (non-time-bounded) lock operation on mutexes created with the TutMutexCreateFast function. If the mutex is not immediately available, the calling thread is suspended until it becomes available.

Caution Every call to TutMutexLockFast should be balanced by a subsequent call from the same thread to TutMutexUnlock.

See Also TutMutexCreateFast, TutMutexLock, TutMutexUnlock, TutMutexDestroy

Examples The example for TutMutexCreateFast on page 144 demonstrates the use of TutMutexLockFast.

TutMutexUnlock

Name

TutMutexUnlock — unlock a mutex object

Synopsis

```
T_BOOL TutMutexUnlock(mutex)
T_MUTEX mutex;
```

Arguments

mutex — mutex object to unlock

Return Values

TRUE if mutex object was successfully unlocked, FALSE otherwise.

Diagnostics

If TutMutexUnlock fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_OS — an operating system error has occurred. Call the function TutErrNumGetOs to obtain the related OS error value.
- T_ERR_TYPE_MISMATCH — the platform does not support threads.
- T_ERR_VAL_INVALID — *mutex* was not a valid mutex object.

Description

TutMutexUnlock unlocks a locked mutex object. Only the thread that owns the lock on *mutex* can successfully unlock it.

Caution

Every call to TutMutexUnlock must balance a previous call to TutMutexLock.

See Also

TutMutexCreate, TutMutexLock

Examples

This example attempts to unlock a mutex object:

```
T_MUTEX mutex;
...
if (!TutMutexUnlock(mutex)) {
  /* error */
}
```

TutOptionChangeCbCreate

Name

TutOptionChangeCbCreate — create a callback that is called when an option value changes

Synopsis

```
T_CB TutOptionChangeCbCreate(option, change_func, arg)
T_OPTION option;
T_OPTION_CHANGE_CB_FUNC change_func;
T_CB_ARG arg;
```

Arguments

option — option to register the change callback for; NULL if callback is to be registered for all options

change_func — callback function

arg — callback function argument

Return Values

New callback if successful, NULL otherwise.

Diagnostics

If TutOptionChangeCbCreate fails, it returns NULL and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *change_func* argument was NULL
- T_ERR_ALREADY_EXISTS — a callback with *change_func* and *arg* already exists

Description

TutOptionChangeCbCreate attaches the *change_func* callback to *option*. When the value of *option* changes, *change_func* is called with three arguments:

- *option* — option whose value changed
- *data* — pointer to callback data structure
- *arg* — callback function argument

Caution

The T_ENTRY declaration specifier is required in the definition of all callback functions as well as their prototypes.

See Also

TutOptionChangeCbLookup

Examples This example prints the value of `bool_opt` when its value changes:

```
/* This callback function is called when the */
/* value of option "bool_opt" changes. */
void T_ENTRY BoolChanged(option, data, arg)
T_OPTION option;
T_OPTION_CHANGE_CB_DATA data;
T_CB_ARG arg;
{
  T_BOOL bool_return;

  if (!TutOptionGetBool(option, &bool_return)) {
  /* error */
  }
  if (bool_return) {
    TutOut("Value of option %s is TRUE\n",
TutOptionGetName(option));
  }
  else {
    TutOut("Value of option %s is FALSE\n",
TutOptionGetName(option));
  }
} /* BoolChanged */

/* Below is a code fragment that could go into an initialization routine. */
T_OPTION option;
T_CB cb;

option = TutOptionCreate("bool_opt", T_OPT_TYPE_BOOLEAN);
if (option == NULL) {
 /* error */
}

cb = TutOptionChangeCbCreate(option, BoolChanged, NULL);
if (cb == NULL) {
 /* error */
}
```

TutOptionChangeCbLookup

Name

TutOptionChangeCbLookup — look up an option change callback

Synopsis

```
T_CB TutOptionChangeCbLookup(option, change_func, arg)
T_OPTION option;
T_OPTION_CHANGE_CB_FUNC change_func;
T_CB_ARG arg;
```

Arguments

option — option to look up the change callback for; NULL if looking up a global callback (for all options)

change_func — callback function

arg — callback function argument

Return Values

Returns callback if successful, NULL otherwise.

Diagnostics

If TutOptionChangeCbLookup fails, it returns NULL and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *change_func* argument was NULL
- T_ERR_DOESNT_EXIST — no such callback exists

Description

TutOptionChangeCbLookup returns the callback registered with TutOptionChangeCbCreate, if any. The arguments, *option*, *change_func*, and *arg*, must match precisely those used in TutOptionChangeCbCreate for the call to be successful.

Caution

None

See Also

TutOptionChangeCbCreate

Examples This example looks up the change callback value of `bool_opt`:

```
/* This callback function is called when the value of option "bool_opt" changes. */
void T_ENTRY BoolChanged(option, data, arg)
T_OPTION option;
T_OPTION_CHANGE_CB_DATA data;
T_CB_ARG arg;
{
  T_BOOL bool_return;

  if (!TutOptionGetBool(option, &bool_return)) {
  /* error */
  }
  if (bool_return) {
    TutOut("Value of option %s is TRUE\n",
             TutOptionGetName(option));
  }
  else {
    TutOut("Value of option %s is FALSE\n",
             TutOptionGetName(option));
  }
} /* BoolChanged */

/* Below is a code fragment that could go into  */
/* an initialization routine.          */
T_OPTION option;
T_CB cb, bool_cb;

option = TutOptionCreate("bool_opt", T_OPT_TYPE_BOOLEAN);
if (option == NULL) {
 /* error */
}

cb = TutOptionChangeCbCreate(option, BoolChanged, NULL);
if (cb == NULL) {
 /* error */
}

bool_cb = TutOptionChangeCbLookup(option, BoolChanged, NULL);
if (bool_cb != cb) {
 /* error */
}
```

TutOptionCreate

Name TutOptionCreate — create a new option

Synopsis

```
T_OPTION TutOptionCreate(name, type)
T_STR name;
T_OPT_TYPE type;
```

Arguments *name* — option name (not case sensitive)

type — option value type

Return Values Returns pointer to new option, `NULL` if option already exists.

Diagnostics If TutOptionCreate fails, it returns `NULL` and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *name* argument was `NULL`
- T_ERR_VAL_INVALID — *type* argument was not valid
- T_ERR_ALREADY_EXISTS — option has already been created

Description TutOptionCreate creates a new SmartSockets option. An option consists of *name* and *type*, where *name* is a string that is not case sensitive, and *type* is one of:

- T_OPT_TYPE_BOOLEAN
- T_OPT_TYPE_ENUM
- T_OPT_TYPE_NUMERIC
- T_OPT_TYPE_STRING
- T_OPT_TYPE_LIST_ENUM
- T_OPT_TYPE_LIST_NUMERIC
- T_OPT_TYPE_LIST_STRING

SmartSockets options are similar to environment variables used by operating systems, plus they have additional built-in type-checking capabilities. Option change callbacks can be used to notify you when the option's value changes. See TutOptionChangeCbCreate on page 151 for more information on option change callbacks.

Caution The *name* argument must be an identifier.

See Also TutOptionLookup, TutOptionDestroy, TutOptionChangeCbCreate

Examples This example creates a boolean option named `bool_opt`:

```
T_OPTION option;

option = TutOptionCreate("bool_opt", T_OPT_TYPE_BOOLEAN);

if (option == NULL) {
 /* error */
}
```

TutOptionDestroy

Name TutOptionDestroy — destroy an existing option

Synopsis

```
T_BOOL TutOptionDestroy(option)
T_OPTION option;
```

Arguments *option* — option to destroy

Return Values TRUE if the option was successfully destroyed, FALSE otherwise.

Diagnostics If TutOptionDestroy fails, it returns FALSE and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *option* argument was NULL

Description TutOptionDestroy destroys *option*. All storage allocated for *option* is deallocated.

Caution In general, it is unsafe to destroy standard options that are provided in SmartSockets processes. These processes frequently fail in unexpected ways if any of the standard options have been destroyed.

See Also TutOptionCreate, TutOptionLookup

Examples This example destroys the option named bool_opt:

```
T_OPTION option;
T_BOOL status;

/* Look up the option first */
option =     TutOptionLookup("bool_opt");
if (option == NULL) {
  /* error */
}
if (!TutOptionDestroy(option)) {
  /* error */
}
```

TutOptionGetBool

Name

TutOptionGetBool — get the value of a boolean option

Synopsis

```
T_BOOL TutOptionGetBool(option, bool_val_return)
T_OPTION option;
T_BOOL *bool_val_return;
```

Arguments

option — option that contains a boolean value

bool_val_return — pointer to a T_BOOL into which TutOptionGetBool copies the value of the option

Return Values

`TRUE` if the value is retrieved successfully, `FALSE` otherwise.

Diagnostics

If TutOptionGetBool fails, it returns `FALSE` and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* or *bool_val_return* was `NULL`
- T_ERR_TYPE_INVALID — *option* was not of type T_OPT_TYPE_BOOL
- T_ERR_VAL_UNKNOWN — value of *option* is `UNKNOWN`

Description

TutOptionGetBool gets the boolean value of *option*. A T_BOOL value (`TRUE` or `FALSE`) is copied into the storage pointed to by *bool_val_return*.

The caller must manage the storage pointed to by *bool_val_return*.

Caution

None

See Also

TutOptionSetBool

Examples

This example gets the value of the option named `bool_opt`:

```
T_OPTION bool_opt;
T_BOOL bool_val;

bool_opt = TutOptionLookup("bool_opt");
if (bool_opt == NULL) {
 /* error */
}
if (!TutOptionGetBool(bool_opt, &bool_val)) {
 /* error */
}
```

TutOptionGetEnum

Name TutOptionGetEnum — get the value of an enumerated option

Synopsis

```
T_BOOL TutOptionGetEnum(option, enum_val_return)
T_OPTION option;
T_STR *enum_val_return;
```

Arguments *option* — option that contains an enumerated value.

enum_val_return — pointer to a T_STR into which TutOptionGetEnum copies a pointer to the enumerated value.

Return Values TRUE if the value is retrieved successfully, FALSE otherwise.

Diagnostics If TutOptionGetEnum fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* or *enum_val_return* was NULL
- T_ERR_TYPE_INVALID — *option* was not of type T_OPT_TYPE_ENUM
- T_ERR_VAL_UNKNOWN — value of *option* is UNKNOWN

Description TutOptionGetEnum gets the enumerated value of *option*. A T_STR pointer value is copied into storage pointed to by *enum_val_return*, but no additional space is allocated for the string itself.

Caution The storage pointed to by *enum_val_return* should not be modified or freed. The storage is thread safe, but may be overwritten by subsequent calls to SmartSockets APIs within the same thread. If you wish to preserve the value, you should make a copy of the value.

See Also TutOptionSetEnum

Examples This example gets the value of the option named enum_opt:

```
T_OPTION enum_opt;
T_STR enum_val;

enum_opt = TutOptionLookup("enum_opt");
if (enum_opt == NULL) {
 /* error */
}

if (!TutOptionGetEnum(enum_opt, &enum_val)) {
 /* error */
}
```

TutOptionGetEnumList

Name TutOptionGetEnumList — get the value of an enumerated list option

Synopsis

```
T_BOOL TutOptionGetEnumList(option, enum_list_val_return)
T_OPTION option;
T_STR_LIST *enum_list_val_return;
```

Arguments *option* — option that contains an enumerated list.

enum_list_val_return — pointer to a T_STR_LIST into which TutOptionGetEnumList copies a pointer to the enumerated list.

Return Values TRUE if the value is retrieved successfully, FALSE otherwise.

Diagnostics If TutOptionGetEnumList fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* or *enum_list_val_return* was NULL
- T_ERR_TYPE_INVALID — *option* was not of type T_OPT_TYPE_LIST_ENUM
- T_ERR_VAL_UNKNOWN — value of *option* is UNKNOWN

Description TutOptionGetEnumList gets the enumerated list value of *option*. A T_STR_LIST pointer value is copied into storage pointed to by *enum_list_val_return*, but no additional space is allocated for the string itself. A T_STR_LIST is a linked list of strings.

Caution The storage pointed to by *enum_list_val_return* should not be modified or freed. The storage is thread safe, but may be overwritten by subsequent calls to SmartSockets APIs within the same thread. If you wish to preserve the value, you should make a copy of the value.

See Also TutOptionSetEnumList

Examples This example gets the value of the option named `enum_list_opt`:

```
T_OPTION enum_list_opt;
T_STR_LIST enum_list_val, node;

enum_list_opt = TutOptionLookup("enum_list_opt");
if (enum_list_opt == NULL) {
 /* error */
}
if (!TutOptionGetEnumList(enum_list_opt, &enum_list_val)) {
 /* error */
}
/* Dump the string values */
for (node = enum_list_val; node != NULL; node = node->next) {
  TutOut("%s\n", node->string);
}
```

TutOptionGetKnown

Name

TutOptionGetKnown — retrieves the known state for an option

Synopsis

```
T_BOOL TutOptionGetKnown(option, known_return)
T_OPTION option;
T_BOOL *known_return;
```

Arguments

option — option to determine if the value is known

known_return — pointer to storage location to place current known property value

Return Values

TRUE if able to determine known state of *option*, FALSE otherwise.

Diagnostics

If TutOptionGetKnown fails, it returns FALSE and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *option* or *known_return* is NULL.

Description

TutOptionGetKnown retrieves the current known state of *option*, such as set to a valid value, and stores it in the location pointed to by *known_return*.

Caution

None

See Also

TutOptionSetUnknown, TutOptionLookup

Examples

This example determines if the value of the option str_opt is known:

```
T_OPTION str_opt;
T_BOOL known;

str_opt = TutOptionLookup("str_opt");
if (str_opt == NULL) {
 /* error */
}

if (!TutOptionGetKnown(str_opt, &known)) {
  /* error */
}
TutOut("str_opt value is %s\n", known ? "KNOWN" : "UNKNOWN");
```

TutOptionGetName

Name TutOptionGetName — get the name of an option

Synopsis
```
T_STR TutOptionGetName(option)
T_OPTION option;
```

Arguments *option* — option from which to get the name

Return Values Name of *option*

Diagnostics If TutOptionGetName fails, it returns NULL and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *option* was NULL

Description TutOptionGetName returns the name of *option*.

Caution The pointer returned by this function points to the actual storage used by the option, and thus should not be modified.

See Also TutOptionCreate, TutOptionLookup

Examples This example prints the name of an option from a change callback (see TutOptionChangeCbCreate on page 151):

```
void T_ENTRY OptChanged(option, data, arg)
T_OPTION option;
T_OPTION_CHANGE_CB_DATA data;
T_CB_ARG arg;
{
  TutOut("Option name: %s\n",TutOptionGetName(option));
} /* OptChanged */
```

TutOptionGetNum

Name

TutOptionGetNum — get the value of a numeric option

Synopsis

```
T_BOOL TutOptionGetNum(option, num_val_return)
T_OPTION option;
T_REAL8 *num_val_return;
```

Arguments

option — option to get numeric value of

num_val_return — pointer to T_REAL8 field into which TutOptionGetNum copies the numeric value

Return Values

`TRUE` if value is retrieved successfully, `FALSE` otherwise.

Diagnostics

If TutOptionGetNum fails, it returns `FALSE` and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* or *num_val_return* was `NULL`
- T_ERR_TYPE_INVALID — *option* was not of type T_OPT_TYPE_NUMERIC
- T_ERR_VAL_UNKNOWN — value of *option* is `UNKNOWN`

Description

TutOptionGetNum gets the numeric value of *option*. The value is copied into storage pointed to by *num_val_return*, which must be managed by the caller.

Caution

None

See Also

TutOptionSetNum

Examples

This example gets the value of the option named `num_opt`:

```
T_OPTION num_opt;
T_REAL8 num_val;
num_opt = TutOptionLookup("num_opt");
if (num_opt == NULL) {
 /* error */
}
if (!TutOptionGetNum(num_opt, &num_val)) {
 /* error */
}
```

TutOptionGetReadOnly

Name

TutOptionGetReadOnly — get the read-only property of an option

Synopsis

```
T_BOOL TutOptionGetReadOnly(option, read_only_return)
T_OPTION option;
T_BOOL *read_only_return;
```

Arguments

option — option whose read-only property is to be accessed

read_only_return — pointer to storage location to place read-only property

Return Values

TRUE if able to determine value of the read-only property for *option*, FALSE otherwise.

Diagnostics

If TutOptionGetReadOnly fails, it returns FALSE and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *option* or *read_only_return* was NULL

Description

TutOptionGetReadOnly retrieves the current state of the read-only property for *option*. A read-only option is an option whose value cannot be changed.

Caution

None

See Also

TutOptionSetReadOnly, TutOptionLookup

Examples

This example prints whether or not the option is read-only:

```
T_OPTION opt;
T_BOOL read_only;

opt = TutOptionLookup("any_option");
if (opt == NULL) {
 /* error */
}

if (!TutOptionGetReadOnly(opt, &read_only)) {
  /* error */
}
TutOut("Option any_option is %s READ ONLY\n",
         read_only ? "" : "NOT");
```

TutOptionGetRequired

Name

TutOptionGetRequired — get the required property of an option

Synopsis

```
T_BOOL TutOptionGetRequired(option, required_return)
T_OPTION option;
T_BOOL *required_return;
```

Arguments

option — option whose required property is to be accessed

required_return — pointer to storage location to place required property

Return Values

TRUE if able to determine value of the required state for *option*, FALSE otherwise.

Diagnostics

If TutOptionGetRequired fails, it returns FALSE and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *option* or *required_return* was NULL

Description

TutOptionGetRequired retrieves the current state of the required property for *option.* A required option is an option that cannot be unset.

Caution

None

See Also

TutOptionSetRequired, TutOptionLookup

Examples

This example prints whether the option is required:

```
T_OPTION opt;
T_BOOL required;

opt = TutOptionLookup("any_option");
if (opt == NULL) {
 /* error */
}
if (!TutOptionGetRequired(opt, &required)) {
  /* error */
}
TutOut("Option any_option is %s REQUIRED\n",
        required ? "" : "NOT");
```

TutOptionGetStr

Name

TutOptionGetStr — get the value of a string option

Synopsis

```
T_BOOL TutOptionGetStr(option, str_val_return)
T_OPTION option;
T_STR *str_val_return;
```

Arguments

option — option that contains a string value.

str_val_return — pointer to a T_STR into which TutOptionGetStr copies a pointer to the string.

Return Values

TRUE if the value is retrieved successfully, FALSE otherwise.

Diagnostics

If TutOptionGetStr fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* or *str_val_return* was NULL
- T_ERR_TYPE_INVALID — *option* was not of type T_OPT_TYPE_STRING
- T_ERR_VAL_UNKNOWN — value of *option* is UNKNOWN

Description

TutOptionGetStr gets the string value of *option*. A T_STR pointer value is copied into storage pointed to by *str_val_return*, but no additional space is allocated for the string itself.

Caution

The storage pointed to by *str_val_return* should not be modified or freed. The storage is thread safe, but may be overwritten by subsequent calls to SmartSockets APIs within the same thread. If you wish to preserve the value, you should make a copy of the value.

See Also

TutOptionSetStr

Examples

This example gets the value of the option named str_opt:

```
T_OPTION str_opt;
T_STR str_val;

str_opt = TutOptionLookup("str_opt");
if (str_opt == NULL) {
 /* error */
}

if (!TutOptionGetStr(str_opt, &str_val)) {
 /* error */
}
```

TutOptionGetStrList

Name

TutOptionGetStrList — get the value of a string list option

Synopsis

```
T_BOOL TutOptionGetStrList(option, str_list_val_return)
T_OPTION option;
T_STR_LIST *str_list_val_return;
```

Arguments

option — option that contains a string list.

str_list_val_return — pointer to a T_STR_LIST into which TutOptionGetStrList copies a pointer to the string list.

Return Values

TRUE if value is retrieved successfully, FALSE otherwise.

Diagnostics

If TutOptionGetStrList fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* or *str_list_val_return* was NULL
- T_ERR_TYPE_INVALID — *option* was not of type T_OPT_TYPE_LIST_STRING
- T_ERR_VAL_UNKNOWN — value of *option* is UNKNOWN

Description

TutOptionGetStrList gets the string list value of *option*. A T_STR_LIST pointer value is copied into storage pointed to by *str_list_val_return*, but no additional space is allocated for the string itself. A T_STR_LIST is a linked list of strings.

Caution

The storage pointed to by *str_list_val_return* should not be modified or freed. The storage is thread safe, but may be overwritten by subsequent calls to SmartSockets APIs within the same thread. If you wish to preserve the value, you should make a copy of the value.

See Also

TutOptionSetStrList

Examples This example gets the value of option name `str_list_opt`:

```
T_OPTION str_list_opt;
T_STR_LIST str_list_val, node;

str_list_opt = TutOptionLookup("str_list_opt");
if (str_list_opt == NULL) {
 /* error */
}
if (!TutOptionGetStrList(str_list_opt, &str_list_val)) {
 /* error */
}
/* Dump the string values */
for (node = str_list_val; node != NULL; node = node->next) {
  TutOut("%s\n", node->string);
}
```

TutOptionGetType

Name TutOptionGetType — get the type of an option

Synopsis

```
T_BOOL TutOptionGetType(option, opt_type)
T_OPTION option;
T_OPT_TYPE *opt_type;
```

Arguments *option* — option from which to retrieve type

opt_type — location to store type of specified option.

Return Values Type of *option*.

Diagnostics If TutOptionGetType fails, it returns FALSE and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *option* or *opt_type* was NULL

Description TutOptionGetType returns the type of *option*. Valid types are:

- T_OPT_TYPE_BOOLEAN
- T_OPT_TYPE_ENUM
- T_OPT_TYPE_NUMERIC
- T_OPT_TYPE_STRING
- T_OPT_TYPE_LIST_ENUM
- T_OPT_TYPE_LIST_NUMERIC
- T_OPT_TYPE_LIST_STRING

Caution None

See Also TutOptionCreate, TutOptionLookup

Examples This example prints the option type:

```
T_OPTION opt;
T_OPT_TYPE option_type;

opt = TutOptionLookup("any_option");
if (opt == NULL) {
 /* error */
}

if (!TutOptionGetType(opt, &option_type)) {
  /* error */
}
switch (option_type) {
case T_OPT_TYPE_NUMERIC:
  TutOut("Type of any_option is NUMERIC\n");
  break;
case T_OPT_TYPE_ENUM:
  TutOut("Type of any_option is ENUMERATED\n");
  break;
case T_OPT_TYPE_BOOLEAN:
  TutOut("Type of any_option is BOOLEAN\n");
  break;
case T_OPT_TYPE_STRING:
  TutOut("Type of any_option is STRING\n");
  break;
case T_OPT_TYPE_LIST_ENUM:
  TutOut("Type of any_option is ENUM LIST\n");
  break;
case T_OPT_TYPE_LIST_STRING:
  TutOut("Type of any_option is STRING LIST\n");
  break;
case T_OPT_TYPE_LIST_NUMERIC:
  TutOut("Type of any_option is NUMERIC LIST\n");
  break;
default:
  TutOut("Unknown Type\n");
} /* switch */
```

TutOptionGetVerifyFunc

Name

TutOptionGetVerifyFunc — get the verify function of an option

Synopsis

```
T_BOOL TutOptionGetVerifyFunc(option, verify_func_return)
T_OPTION option;
T_OPTION_VERIFY_FUNC *verify_func_return;
```

Arguments

option — option to get the verify function from

verify_func_return — pointer to a T_OPTION_VERIFY_FUNC into which TutOptionGetVerifyFunc copies a pointer to the verify function

Return Values

TRUE if value is retrieved successfully, FALSE otherwise.

Diagnostics

If TutOptionGetVerifyFunc fails, it returns FALSE and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *option* or *verify_func_return* was NULL

Description

TutOptionGetVerifyFunc retrieves the verify function from *option*. For more information on verify functions, see TutOptionSetVerifyFunc on page 188.

Caution

None

See Also

TutOptionSetVerifyFunc

Examples

This example retrieves and calls a verify function:

```
/* This verify function is called when the */
/* value of option "num_opt" changes. */
T_BOOL VerifyNum(option, new_value, arg)
T_OPTION option;
T_PTR new_value;
T_PTR arg;
{
  T_REAL8 num_return;

  num_return = *((T_REAL8 *)new_value);
  if (num_return >= 0.0 && num_return <= 10.0) {
 /* Value of "num_opt" will be changed to new_value */
    return TRUE;
  }
  else {
 /* Value of "num_opt" will NOT be changed */
    return FALSE;
  }
} /* VerifyNum */
```

```
/* Below is a code fragment that could go into   */
/* an initialization routine.                    */
T_OPTION option;

option =     TutOptionCreate("num_opt", T_OPT_TYPE_NUMERIC);
if (option == NULL) {
 /* error */
}

if (!TutOptionSetVerifyFunc(option, VerifyNum, NULL) ) {
 /* error */
}

...

/* The function below retrieves the verify */
/* function of option "num_opt", and then calls the */
/* verify function directly to see if the value */
/* of num is valid. */
void CheckNum(num)
T_REAL8 num;
{
  T_OPTION option;
  T_OPTION_VERIFY_FUNC verify_func;

 /* Retrieve the verify function */
  option = TutOptionLookup("num_opt");
  if (option == NULL) {
  /* error */
  }

  if (TutOptionGetVerifyFunc(option, &verify_func)) {
  /* Provides a way to check a value to determine  */
  /* if it passes the option's validation test.    */
  /* Note that the address of the value is expected  */
  /* by the verify function.                       */
     if ((*verify_func)(option, (T_PTR)&num, NULL)) {
     TutOut("Value of %f is valid.\n", num);
     }
     else {
     TutOut("Value of %f is invalid.\n", num);
     }
  }
} /* CheckNum */
```

TutOptionLegValAdd

Name TutOptionLegValAdd — add a legal value to an enumerated option

Synopsis

```
T_BOOL TutOptionLegValAdd(option, val)
T_OPTION option;
T_STR val;
```

Arguments *option* — option to add a legal value to

val — legal value to be added to the option

Return Values TRUE if the value is added successfully, FALSE otherwise.

Diagnostics If TutOptionLegValAdd fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* or *val* was NULL
- T_ERR_TYPE_INVALID — *option* was not of type T_OPT_TYPE_ENUM
- T_ERR_VAL_INVALID — *val* was set to unknown, which is not a legal value
- T_ERR_ALREADY_EXISTS — *val* already exists in legal values table for *option*

Description TutOptionLegValAdd adds *val* to the list of legal values allowed for *option*.

Caution unknown is reserved and cannot be added as an enumerated value.

See Also TutOptionCreate, TutOptionLookup, TutOptionGetEnum, TutOptionSetEnum, TutOptionGetEnumList

Examples This example adds the legal value red to the enumerated option named colors:

```
T_OPTION color_opt;

color_opt = TutOptionCreate("colors", T_OPT_TYPE_ENUM);
if (color_opt == NULL) {
 /* error */
}

if (!TutOptionLegValAdd(color_opt, "red")) {
 /* error */
}
```

TutOptionLookup

Name TutOptionLookup — look up an option by name

Synopsis

```
T_OPTION TutOptionLookup(name)
T_STR name;
```

Arguments *name* — option name (not case sensitive)

Return Values Returns pointer to the option if found, NULL otherwise.

Diagnostics If TutOptionLookup fails, it returns NULL and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *name* was NULL

Description TutOptionLookup returns a pointer to the option *name*, or NULL if no such option exists.

Caution None

See Also TutOptionCreate

Examples This example looks up the option named bool_opt:

```
T_OPTION option;

option = TutOptionLookup("bool_opt");
if (option == NULL) {
 /* error */
}
```

TutOptionNamedLookup

Name

TutOptionLookup — look up a named option by name

Synopsis

```
T_OPTION TutOptionNamedLookup(name, option_name)
T_STR name;
T_STR option_name;
```

Arguments

name — named options name (not case sensitive)

option_name — option name (not case sensitive)

Return Values

Returns pointer to the option if found, `NULL` otherwise.

Diagnostics

If TutOptionNamedLookup fails, it returns `NULL` and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *name* or *option_name* was `NULL`

Description

TutOptionNamedLookup returns a pointer to the option *option_name* found for the named options name *name*, or `NULL` if no such option exists.

Caution

None

See Also

TutOptionCreate

Examples

This example looks up the option named `bool_opt` from the named options `client_1`:

```
T_OPTION option;

option = TutOptionNamedLookup("client_1", "bool_opt");
if (option == NULL) {
 /* error */
}
```

TutOptionSetBool

Name TutOptionSetBool — set the value of a boolean option

Synopsis

```
T_BOOL TutOptionSetBool(option, bool_val)
T_OPTION option;
T_BOOL bool_val;
```

Arguments *option* — option to set

bool_val — boolean value to set option to

Return Values TRUE if value is set successfully, FALSE otherwise.

Diagnostics If TutOptionSetBool fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* was NULL.
- T_ERR_TYPE_INVALID — *option* was not of type T_OPT_TYPE_BOOL.
- T_ERR_VAL_INVALID — *bool_val* not a boolean value (TRUE or FALSE).
- T_ERR_VERIFY_FUNC — verify function failed. Other error numbers are possible depending on the verify function for *option*.

Description TutOptionSetBool sets the boolean value of *option* to *bool_val*.

Caution None

See Also TutOptionGetBool

Examples This example sets the value of the option named bool_opt to TRUE:

```
T_OPTION bool_opt;

bool_opt = TutOptionLookup("bool_opt");
if (bool_opt == NULL) {
 /* error */
}
if (!TutOptionSetBool(bool_opt, TRUE)) {
 /* error */
}
```

TutOptionSetEnum

Name

TutOptionSetEnum — set the value of an enumerated option

Synopsis

```
T_BOOL TutOptionSetEnum(option, enum_val)
T_OPTION option;
T_STR enum_val;
```

Arguments

option — option to set

enum_val — enumerated value to set option to

Return Values

`TRUE` if value is set successfully, `FALSE` otherwise.

Diagnostics

If TutOptionSetEnum fails, it returns `FALSE` and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* or *enum_val* was `NULL`.
- T_ERR_TYPE_INVALID — *option* was not of type T_OPT_TYPE_ENUM.
- T_ERR_VAL_INVALID — *enum_val* was not a legal value for *option.*
- T_ERR_VERIFY_FAILED — verify function failed. Other error numbers are possible depending on the verify function in use for *option*.

Description

TutOptionSetEnum sets the enumerated value of *option* to *enum_val*.

If no legal values for *option* have been specified using TutOptionLegValAdd, any value is accepted.

Caution

None

See Also

TutOptionGetEnum

Examples

This example sets the value of the option named `enum_opt` to the enumerated value `red`:

```
T_OPTION enum_opt;

enum_opt = TutOptionLookup("enum_opt");
if (enum_opt == NULL) {
 /* error */
}

if (!TutOptionSetEnum(enum_opt, "red")) {
 /* error */
}
```

TutOptionSetEnumList

Name TutOptionSetEnumList — set the value of an enumerated list option

Synopsis

```
T_BOOL TutOptionSetEnumList(option, enum_list_val)
T_OPTION option;
T_STR_LIST enum_list_val;
```

Arguments

option — option to set

enum_list_val — enumerated list value to set option to

Return Values TRUE if value is set successfully, FALSE otherwise.

Diagnostics If TutOptionSetEnumList fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* or *enum_list_val* was NULL.
- T_ERR_TYPE_INVALID — *option* was not of type T_OPT_TYPE_LIST_ENUM.
- T_ERR_VERIFY_FAILED — verify function failed. Other possible error numbers are possible depending on the verify function used.

Description TutOptionSetEnumList sets the enumerated list value of *option*. The argument *enum_list_val* is a linked list of strings. The string list passed in is copied, so the caller does not need to maintain storage for this list.

Caution There is no way to specify legal values for enumerated list options. If this type of checking is required, it must be implemented using an option verify function. See TutOptionSetVerifyFunc on page 188 for more information on option verify functions.

See Also TutOptionGetEnumList, TutOptionSetVerifyFunc

Examples This example sets the value of the option named `enum_list_opt` to contain two enumerated values:

```
T_OPTION enum_list_opt;
T_STR_LIST enum_list_val, node;
T_STR_LIST_STRUCT node1, node2;

enum_list_opt = TutOptionLookup("enum_list_opt");
if (enum_list_opt == NULL) {
 /* error */
}

node1.string = "red";
node2.string = "blue";
node1.next = &node2;
node2.next = NULL;

if (!TutOptionSetEnumList(enum_list_opt, &node1)) {
 /* error */
}

/* Dump the string values */
if (!TutOptionGetEnumList(enum_list_opt, &enum_list_val)) {
 /* error */
}

for (node = enum_list_val; node != NULL; node = node->next) {
  TutOut("%s\n", node->string);
}
```

TutOptionSetNum

Name

TutOptionSetNum — set the value of a numeric option

Synopsis

```
T_BOOL TutOptionSetNum(option, num_val)
T_OPTION option;
T_REAL8 num_val;
```

Arguments

option — option to set

num_val — numeric value to set option to

Return Values

TRUE if value is set successfully, FALSE otherwise.

Diagnostics

If TutOptionSetNum fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* was NULL.
- T_ERR_TYPE_INVALID — *option* was not of type T_OPT_TYPE_NUMERIC.
- T_ERR_VERIFY_FAILED — verify function failed. Various other error numbers are possible depending on the verify function in use.

Description

TutOptionSetNum sets the numeric value of *option* to *num_val*.

Caution

None

See Also

TutOptionGetNum

Examples

This example sets the value of the option named num_opt to 10.0:

```
T_OPTION num_opt;

num_opt = TutOptionLookup("num_opt");
if (num_opt == NULL) {
 /* error */
}

if (!TutOptionSetNum(num_opt, 10.0)) {
 /* error */
}
```

TutOptionSetReadOnly

Name

TutOptionSetReadOnly — set whether or not an option is read-only

Synopsis

```
T_BOOL TutOptionSetReadOnly(option, read_only)
T_OPTION option;
T_BOOL read_only;
```

Arguments

option — option to set

read_only — sets whether or not the option is read-only

Return Values

TRUE if operation was successful, FALSE otherwise.

Diagnostics

If TutOptionSetReadOnly fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* was NULL
- T_ERR_VAL_INVALID — *read_only* was not a valid boolean value

Description

TutOptionSetReadOnly sets whether or not *option* is read-only. A read-only option is an option whose value cannot be changed.

Caution

None

See Also

TutOptionGetReadOnly, TutOptionLookup

Examples

This example makes an option read-only:

```
T_OPTION opt;

opt = TutOptionLookup("any_option");
if (opt == NULL) {
 /* error */
}

if (!TutOptionSetReadOnly(opt, TRUE)) {
 /* error */
}
```

TutOptionSetRequired

Name

TutOptionSetRequired — set whether or not an option is required

Synopsis

```
T_BOOL TutOptionSetRequired(option, required)
T_OPTION option;
T_BOOL required;
```

Arguments

option — option to set

required — sets whether or not the option is required

Return Values

TRUE if operation was successful, FALSE otherwise.

Diagnostics

If TutOptionSetRequired fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* was NULL
- T_ERR_VAL_INVALID — *required* was not a valid boolean value

Description

TutOptionSetRequired sets whether or not *option* is required. A required option is an option that cannot be unset.

Caution

None

See Also

TutOptionGetRequired, TutOptionLookup

Examples

This example makes an option required:

```
T_OPTION opt;

opt = TutOptionLookup("any_option");
if (opt == NULL) {
 /* error */
}

if (!TutOptionSetRequired(opt, TRUE)) {
 /* error */
}
```

TutOptionSetStr

Name

TutOptionSetStr — set the value of a string option

Synopsis

```
T_BOOL TutOptionSetStr(option, str_val)
T_OPTION option;
T_STR str_val;
```

Arguments

option — option to set

str_val — string value to set option to

Return Values

`TRUE` if value is set successfully, `FALSE` otherwise.

Diagnostics

If TutOptionSetStr fails, it returns `FALSE` and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* or *str_val* was `NULL`
- T_ERR_TYPE_INVALID — *option* was not of type T_OPT_TYPE_STRING
- T_ERR_VERIFY_FAILED — verify function failed. Other error numbers may be possible depending on the verify function for *option*.

Description

TutOptionSetStr sets the string value of *option* to *str_val*.

Caution

None

See Also

TutOptionGetStr

Examples

This example sets the value of the option named `str_opt` to `"hello world"`:

```
T_OPTION str_opt;

str_opt = TutOptionLookup("str_opt");
if (str_opt == NULL) {
 /* error */
}
if (!TutOptionSetStr(str_opt, "hello world")) {
 /* error */
}
```

TutOptionSetStrList

Name

TutOptionSetStrList — set the value of a string list option

Synopsis

```
T_BOOL TutOptionSetStrList(option, str_list_val)
T_OPTION option;
T_STR_LIST str_list_val;
```

Arguments

option — option to set

str_list_val — string list value to set option to

Return Values

`TRUE` if value is set successfully, `FALSE` otherwise.

Diagnostics

If TutOptionSetStrList fails, it returns `FALSE` and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* or *str_list_val* was `NULL`.
- T_ERR_TYPE_INVALID — *option* was not of type T_OPT_TYPE_LIST_STRING.
- T_ERR_VERIFY_FAILED — verify function failed. Other error numbers are possible depending on the verify function for *option*.

Description

TutOptionSetStrList sets the string list value of *option*. The argument *str_list_val* is a linked list of strings. The string list values are copied to internal storage, so the caller does not need to maintain storage for them.

Caution

None

See Also

TutOptionGetStrList

Examples This example sets the value of the option named `str_list_opt` to contain a list of two string values:

```
T_OPTION str_list_opt;
T_STR_LIST str_list_val, node;
T_STR_LIST_STRUCT node1, node2;

str_list_opt = TutOptionLookup("str_list_opt");
if (str_list_opt == NULL) {
 /* error */
}

node1.string = "string of first node";
node2.string = "string of second node";
node1.next = &node2;
node2.next = NULL;

if (!TutOptionSetStrList(str_list_opt, &node1)) {
 /* error */
}

/* Dump the string values */
if (!TutOptionGetStrList(str_list_opt, &str_list_val)) {
 /* error */
}

for (node = str_list_val; node != NULL; node = node->next) {
  TutOut("%s\n", node->string);
}
```

TutOptionSetUnknown

Name TutOptionSetUnknown — set the value of the option to UNKNOWN

Synopsis
```
T_BOOL TutOptionSetUnknown(option)
T_OPTION option;
```

Arguments *option* — option to be set to UNKNOWN

Return Values TRUE if operation is successful, FALSE otherwise.

Diagnostics If TutOptionSetUnknown fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *option* was NULL
- T_ERR_VAL_REQUIRED — attempted to set required *option* value to UNKNOWN

Description TutOptionSetUnknown sets the current value of *option* to UNKNOWN. This is similar to using the `unsetopt` command from the command interface.

Caution None

See Also TutOptionGetKnown, TutOptionLookup

Examples This example sets the value of the option named `str_opt` to UNKNOWN:

```
T_OPTION str_opt;

str_opt = TutOptionLookup("str_opt");
if (str_opt == NULL) {
 /* error */
}

if (!TutOptionSetUnknown(str_opt)) {
 /* error */
}
```

TutOptionSetVerifyFunc

Name

TutOptionSetVerifyFunc — create a function to verify that the option's value is acceptable

Synopsis

```
T_BOOL TutOptionSetVerifyFunc(option, verify_func, arg)
T_OPTION option;
T_OPTION_VERIFY_FUNC verify_func;
T_PTR arg;
```

Arguments

option — option to register the verify function

verify_func — verify function

arg — function argument

Return Values

TRUE if operation was successful, FALSE otherwise.

Diagnostics

If TutOptionSetVerifyFunc fails, it returns FALSE and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *option* was NULL

Description

TutOptionSetVerifyFunc attaches a verify function *verify_func* to *option*. Before the value of *option* changes, *verify_func* is called. Write the code to validate the new value and return TRUE if it is valid, FALSE otherwise. The value of the option is changed only if TRUE is returned by *verify_func*. The function *verify_func* must have three arguments:

- *option* — option whose value changed.
- *new_value* — pointer to the new option value. Cast this value to the appropriate type for the option first so that its contents can be accessed.
- *arg* — function argument.

If the verify function returns FALSE, it issues diagnostic output with TutOut or TutWarning (before returning) to provide feedback to you; the other TutOption* functions do not issue any output in the case of the verify function returning FALSE.

Caution

The argument *verify_func* must return TRUE or FALSE. Other values are undefined.

When writing an option verify function, be sure to account for the possibility that *new_value* may be NULL. This happens when an option is set to UNKNOWN.

See Also

TutOptionCreate, TutOptionLookup

Examples This example verifies that the option named `num_opt` is between `0.0` and `10.0`:

```
/* This verify function is called when the */
/* value of option "num_opt" changes.    */
T_BOOL VerifyNum(T_OPTION option,
                 T_PTR new_value,
                 T_PTR arg)
{
  T_REAL8 num_return;

  num_return = *((T_REAL8 *)new_value);
  if (num_return >= 0.0 && num_return <= 10.0) {
  /* Value of "num_opt" will be changed to new_value */
    return TRUE;
  }
  else {
  /* Value of "num_opt" will NOT be changed */
    TutOut("Illegal value %f for option %s.\n",
         num_return, TutOptionGetName(option));
    return FALSE;
  }
} /* VerifyNum */

/* Below is a code fragment that could go into  */
/* an initialization routine.                   */
T_OPTION option;

option = TutOptionCreate("num_opt", T_OPT_TYPE_NUMERIC);
if (option == NULL) {
 /* error */
}

if (!TutOptionSetVerifyFunc(option, VerifyNum, NULL) ) {
 /* error */
}
```

TutOut

Name

TutOut — unconditional `printf`-style output to an output window

Synopsis

```
void TutOut(format_str, ...)
T_STR format_str;
```

Arguments

format_str — `printf`-style format string

additional args — `printf`-style arguments

Return Values

None

Diagnostics

None

Description

TutOut unconditionally prints to the output window of the SmartSockets process with which the call is being used. TutOut is useful for printing error and informational messages. It supports all the conversion specifications that `printf` does (%f, %g, %d, %s, and so on). See a C reference manual for more information on the conversion specifications.

The output function used by TutOut can be set with TutSetOutputFunc.

Caution

None

See Also

TutSetOutputFunc, TutWarning

Examples

This example uses TutOut for error messages:

```
/* perform some processing */
  if (error_occurs) {
    TutOut("ERROR:could not process data.\n");
    return FALSE;
  }
```

TutOutFlush

Name TutOutFlush — unconditional `printf`-style output to stdout and flush

Synopsis

```
void TutSetOutFlushFunc(format_str, ...)
T_STR format_str;
```

Arguments *format_str* — `printf`-style format string

additional arguments — `printf`-style arguments

Return Values None

Diagnostics None

Description TutOutFlush unconditionally prints to the output window and also causes any unwritten data to be flushed immediately. TutOutFlush is useful for printing error and informational messages. It supports all the conversion specifications that `printf` does.

The flush function used by TutOutFlush can be set with TutSetOutFlushFunc.

Caution None

See Also TutGetOutFlushFunc, TutSetOutFlushFunc

Examples This example uses TutOutFlush to display error messages:

```
if (error_occurs) {
  TutOutFlush("Error: Could not read message\n");
} /* if */
```

Chapter 9 TutPerror - TutRwMutexWriteLocked

This chapter describes these SmartSockets utility functions:

- *TutPerror*
- *TutProcInfo*
- *TutPtrAryAppend*
- *TutPtrAryChangeCbCreate*
- *TutPtrAryChangeCbLookup*
- *TutPtrAryClear*
- *TutPtrAryCreate*
- *TutPtrAryDestroy*
- *TutPtrAryGetItemAtIndex*
- *TutPtrAryGetItemCount*
- *TutPtrAryGetItemIndex*
- *TutPtrAryInsertAfter*
- *TutPtrAryInsertBefore*
- *TutPtrAryInsertItemAtIndex*
- *TutPtrAryItemExists*
- *TutPtrAryRemove*
- *TutPtrAryRemoveAll*
- *TutPtrAryReplaceItemAtIndex*
- *TutPtrArySort*
- *TutPtrArySwapItems*
- *TutPutenv*
- *TutRealloc*
- *TutRealToStr*
- *TutRwMutexCreate*
- *TutRwMutexDestroy*

- *TutRwMutexGetReadQuota*
- *TutRwMutexReadLock*
- *TutRwMutexReadLocked*
- *TutRwMutexSetReadQuota*
- *TutRwMutexUnlock*
- *TutRwMutexUpgradeLock*
- *TutRwMutexWriteLock*
- *TutRwMutexWriteLocked*

TutPerror

Name TutPerror — report UNIX error messages to an output window

Synopsis

```
void TutPerror(msg)
T_STR msg;
```

Arguments *msg* — message to be printed at the beginning of the output

Return Values None

Diagnostics None

Description TutPerror emulates the perror function but directs the output through TutOut, which causes the output to go to the process output window.

Caution TutPerror should only be called when the global SmartSockets error number is set to T_ERR_C, T_ERR_OS or T_ERR_SOCKET.

See Also TutWarning, TutOut

Examples This example prints an error message when a function fails:

```
if (!some_api_func()
      && (TutErrNumGet() == T_ERR_C
          || TutErrNumGet() == T_ERR_OS
          || TutErrNumGet() == T_ERR_SOCKET)) {
  TutPerror("some_api_func");
}
```

TutProcInfo

Name

TutProcInfo — returns process related information

Synopsis

```
void TutProcInfo(request, data)
T_PRINFO_TYPE request;
T_PTR data;
```

Arguments

request — the information type requested.

data — a pointer to the location to store the returned data.

Return Values

TRUE if successful, FALSE otherwise.

Diagnostics

None

Description

TutProcInfo returns process related information. The value given for *request* determines the information returned. Available values for *request* are:

- T_PRINFO_CPU_UTILIZATION — returns the percentage of CPU time currently being used by the process. The *data* argument must be a pointer of type T_REAL8.

See Also

None

Examples

This example gets the percentage of CPU time used by the current process then prints it:

```
T_REAL8 cpu_utilization;
T_BOOL  status = T_TRUE;

status = TutProcInfo(T_PRINFO_CPU_UTILIZATION, &cpu_utilization);
if (T_FALSE == status) {
  /* print error */
}
```

TutPtrAryAppend

Name TutPtrAryAppend — append an item to a pointer array

Synopsis

```
T_BOOL TutPtrAryAppend(ptr_ary, pointer, index_return)
T_PTR_ARY ptr_ary;
T_PTR pointer;
T_INT4 *index_return;
```

Arguments

ptr_ary — pointer array to which the new element should be appended.

pointer — element to be appended to the specified pointer array.

index_return — pointer to optional storage for the index of the newly appended element in the array. This argument may be NULL.

Return Values Returns TRUE if operation is a success, FALSE otherwise.

Diagnostics None

Description TutPtrAryAppend appends a new element to *ptr_ary*. During this operation, *ptr_ary* may be resized to create space for the new element. *index_return* is an optional argument that, if used, contains the index of the newly appended element in the array upon the successful completion of the operation.

TutPtrAryAppend calls the pointer array change callbacks with the reason T_PA_ITEM_INSERTED.

Caution Allocate memory for *index_return* before calling TutPtrAryAppend.

See Also TutPtrAryReplaceItemAtIndex

Examples This example appends an item to a pointer array:

```
T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr;

/* create the pointer array with initial size of 30 */

ptr_ary = TutPtrAryCreate(30);
/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
   /* allocate the space for the next element */
   T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
   *int_ptr = i;
   /* append the item to the array, ignore the index */
   TutPtrAryAppend(ptr_ary, int_ptr, NULL);
}
```

TutPtrAryChangeCbCreate

Name

TutPtrAryChangeCbCreate — create a callback that is called when a pointer array changes

Synopsis

```
T_CB TutPtrAryChangeCbCreate(ptr_ary, change_func, arg)
T_PTR_ARY ptr_ary;
T_PTR_ARY_CHANGE_CB_FUNC change_func;
T_CB_ARG arg;
```

Arguments

ptr_ary — pointer array to register the change callback for

change_func — callback function

arg — callback function argument

Return Values

New callback if successful, NULL otherwise.

Diagnostics

If TutPtrAryChangeCbCreate fails, it returns NULL and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *ptr_ary* or *change_func* argument was NULL
- T_ERR_ALREADY_EXISTS — a callback with *change_func* and *arg* already exists

Description

TutPtrAryChangeCbCreate attaches the *change_func* callback to *ptr_ary*. When an item in *ptr_ary* is inserted or removed, *change_func* is called with three arguments:

- *ptr_ary* — pointer array whose item changed
- *data* — pointer to callback data structure
- *arg* — callback function argument

Caution

The T_ENTRY declaration specifier is required in the definition of all callback functions in addition to their prototypes.

See Also

TutPtrAryChangeCbLookup

Examples This example displays output when a pointer array changes:

```
void T_ENTRY PtrAryChanged(ptr_ary, data, arg)
T_PTR_ARY ptr_ary;
T_PTR_ARY_CHANGE_CB_DATA data;
T_CB_ARG arg;
{
  TutOut("Pointer array change reason is %d.\n", data->reason);
} /* PtrAryChanged */
.
.
.
T_CB cb;

cb = TutPtrAryChangeCbCreate(ptr_ary, PtrAryChanged, NULL);
if (cb == NULL) {
 /* error */
}
```

TutPtrAryChangeCbLookup

Name

TutPtrAryChangeCbLookup — look up a pointer array change callback

Synopsis

```
T_CB TutPtrAryChangeCbLookup(ptr_ary, change_func, arg)
T_PTR_ARY ptr_ary;
T_PTR_ARY_CHANGE_CB_FUNC change_func;
T_CB_ARG arg;
```

Arguments

ptr_ary — pointer array to look up the change callback for

change_func — callback function

arg — callback function argument

Return Values

Returns callback if successful, NULL otherwise.

Diagnostics

If TutPtrAryChangeCbLookup fails, it returns NULL and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *ptr_ary* or *change_func* argument was NULL
- T_ERR_DOESNT_EXIST — no such callback exists

Description

TutPtrAryChangeCbLookup returns the callback registered with TutPtrAryChangeCbCreate, if any. The arguments *ptr_ary*, *change_func*, and *arg* must match precisely those used in TutPtrAryChangeCbCreate for the call to be successful.

Caution

None

See Also

TutPtrAryChangeCbCreate

Examples

This example looks up a change callback:

```
T_CB cb;

cb = TutPtrAryChangeCbLookup(ptr_ary, PtrAryChanged, NULL);
if (cb == NULL) {
 /* error */
}
```

TutPtrAryClear

Name

TutPtrAryClear — clear a pointer array

Synopsis

```
T_BOOL TutPtrAryClear(ptr_ary, user_func, user_arg)
T_PTR_ARY ptr_ary;
T_PTR_ARY_USER_FUNC user_func;
T_PTR user_arg;
```

Arguments

ptr_ary — pointer array to be cleared.

user_func — optional function that is called once for every element in the array. This argument may be `NULL`.

user_arg — argument that is passed to *user_func*.

Return Values

Returns `TRUE` if operation is a success, `FALSE` otherwise.

Diagnostics

None

Description

TutPtrAryClear clears *ptr_ary* of all elements. It does not free the memory allocated to the array, it merely resets the internal array counters to their initial values, so that the array appears to be empty. After the array is cleared, all pointers that were stored in the array become inaccessible, so it is up to you to free the memory that was pointed to by these pointers either before the array is cleared, or while it is being cleared using *user_func*.

If specified, *user_func* is called by TutPtrAryClear for every item in the array before the internal array counters are reset. It is called with three arguments. The first argument is the *index* of the item, the second argument is *item*, and the third argument is the *user_arg* TutPtrAryClear was called with.

TutPtrAryClear calls the pointer array change callbacks with the reason T_PA_ITEM_CLEARED.

Caution

Clearing the pointer array without freeing the memory pointed to by its items may create a memory leak.

See Also

TutPtrAryDestroy, TutPtrAryCreate

Examples This example clears a pointer array:

```
void *clear_func(index, item, arg)
T_INT4 index;
T_PTR item;
T_PTR arg;

{
 T_FREE((T_STR)item);
}

T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr;

/* create the pointer array with initial size of 30 */

ptr_ary = TutPtrAryCreate(30);
/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
   /* allocate the space for the next element */
   T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
   *int_ptr = i;
   /* append the item to the array, ignore the index */
   TutPtrAryAppend(ptr_ary, int_ptr, NULL);
}

/* perform various operations on the array */
.
.
.

/* now, clear the array of all elements */
TutPtrAryClear(ptr_ary, clear_func, NULL);
```

TutPtrAryCreate

Name TutPtrAryCreate — create a pointer array

Synopsis

```
T_PTR_ARY TutPtrAryCreate(size)
T_INT4 size;
```

Arguments *size* — initial size of the pointer array

Return Values Returns a pointer to the new pointer array.

Diagnostics None

Description TutPtrAryCreate creates a new pointer array.

A pointer array is an opaque data structure that contains a zero-based array of pointers to values. The key features of the pointer array data structure are the automatic resizing of the array to provide the room for the next element, thus allowing an unlimited number of elements in the array, and the re-use of previously allocated storage to minimize system calls to allocate memory.

During the call to TutPtrAryCreate, the space for the pointer array data structure is allocated, and the pointer array of *size* is created. *size* is then used as an increment by which the size of the array increases every time the array gets filled up, and a new element needs to be inserted.

For example, if *size* is six, and the seventh element is added, the array size increases by six. After the addition the pointer array contains seven elements, but has enough memory allocated to hold twelve. The average speed of addition of an element depends on the total number of elements to be added and the initial size specified. By monitoring the array size during the program run and adjusting the initial size, it may be possible to optimize memory usage and speed of addition. The speed of access is independent of either the initial size or the total number of elements and is always a constant.

Caution None

See Also TutPtrAryDestroy, TutPtrAryClear

Examples This example creates a new pointer array:

```
T_PTR_ARY ptr_ary;
/* create the pointer array with initial size of 30 */
ptr_ary = TutPtrAryCreate(30);
```

TutPtrAryDestroy

Name

TutPtrAryDestroy — destroy a pointer array

Synopsis

```
T_BOOL TutPtrAryDestroy(ptr_ary, user_func, user_arg)
T_PTR_ARY ptr_ary;
T_PTR_ARY_USER_FUNC user_func;
T_PTR user_arg;
```

Arguments

ptr_ary — pointer array to be destroyed.

user_func — optional function that is called once for every element in the array. This argument may be `NULL`.

user_arg — argument that is passed to *user_func*.

Return Values

Returns `TRUE` if operation is a success, `FALSE` otherwise.

Diagnostics

None

Description

TutPtrAryDestroy destroys the *ptr_ary*. It does not necessarily free the memory allocated to the array. It may, depending on the number of previous calls to TutPtrAryDestroy and TutPtrAryCreate, add the memory to the internal memory pool, thus speeding up the next call to TutPtrAryCreate. After the array is destroyed, all pointers that were stored in the array become inaccessible, so it is up to you to free the memory that was pointed to by these pointers either before the array is destroyed, or while it is being destroyed using *user_func*.

If specified, *user_func* is called by TutPtrAryDestroy for every item in the array before the array is destroyed. It is called with three arguments. The first argument is the *index* of the item, the second argument is the *item*, and the third argument is the *user_arg* TutPtrAryDestroy was called with.

Caution

Destroying the pointer array without freeing the memory pointed to by its items may create a memory leak.

After calling TutPtrAryDestroy, the value of *ptr_ary* becomes undefined and should not be used.

See Also

TutPtrAryClear, TutPtrAryCreate

Examples This example destroys a pointer array:

```
void clear_func(index, item, arg)
T_INT4 index;
T_PTR item;
T_PTR arg;
{
  T_FREE(item);
}

T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr;

/* create the pointer array with initial size of 30 */

ptr_ary = TutPtrAryCreate(30);
/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
  /* allocate the space for the next element */
  T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
  *int_ptr = i;
  /* append the item to the array, ignore the index */
  TutPtrAryAppend(ptr_ary, int_ptr, NULL);
}

/* perform various operations on the array */
.
.
.

/* now, destroy the array */
TutPtrAryDestroy(ptr_ary, clear_func, NULL);
```

TutPtrAryGetItemAtIndex

Name

TutPtrAryGetItemAtIndex — get the item stored at the given index in a pointer array

Synopsis

```
T_BOOL TutPtrAryGetItemAtIndex(ptr_ary, index, element)
T_PTR_ARY ptr_ary;
T_INT4 index;
T_PTR *element;
```

Arguments

ptr_ary — pointer array to get the element from

index — index of the element to get

element — pointer to returned element

Return Values

Returns TRUE if operation is a success; FALSE otherwise.

Diagnostics

None

Description

TutPtrAryGetItemAtIndex returns the element stored in the array at the specified index. Pointer arrays have a zero-based index.

Caution

If *index* is outside of legal boundaries (less than zero or greater than array item count minus one), this function returns FALSE, and the value of the element is undefined.

See Also

TutPtrAryGetItemIndex, TutPtrAryGetItemCount

Examples

This example returns an element stored in an array, from a given index:

```
T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr;

/* create the pointer array with initial size of 30 */
ptr_ary = TutPtrAryCreate(30);

/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
   /* allocate the space for the next element */
   T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
   *int_ptr = i;
   /* append the item to the array, ignore the index */
   TutPtrAryAppend(ptr_ary, int_ptr, NULL);
}
/* look up item at index 13 */
TutPtrAryGetItemAtIndex(ptr_ary, 13, (T_PTR *)&int_ptr);
/* after this operation, int_ptr should point to the location that */
/* contains number 13 */
```

TutPtrAryGetItemCount

Name TutPtrAryGetItemCount — returns the number of items in a pointer array

Synopsis

```
T_BOOL TutPtrAryGetItemCount(ptr_ary, count)
T_PTR_ARY ptr_ary;
T_INT4 *count;
```

Arguments *ptr_ary* — pointer array for which the count is returned

count — pointer to number of items in the array

Return Values Returns TRUE if operation is a success; FALSE if it was passed an invalid pointer array.

Diagnostics None

Description TutPtrAryGetItemCount returns the number of elements currently stored in the pointer array. This function does not actually count the elements, so it is very fast.

Caution Allocate memory for *count* before calling TutPtrAryGetItemCount.

See Also TutPtrAryCreate

Examples This example returns the number of items in the pointer array:

```
T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr, count;

/* create the pointer array with initial size of 30 */
ptr_ary = TutPtrAryCreate(30);
/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
 /* allocate the space for the next element */
  T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
  *int_ptr = i;
 /* append the item to the array, ignore the index */
  TutPtrAryAppend(ptr_ary, int_ptr, NULL);
}
/* now check and see how many items are there in the array */
TutPtrAryGetItemCount(ptr_ary, &count);
/* at this point, count should contain 25 */
```

TutPtrAryGetItemIndex

Name TutPtrAryGetItemIndex — get the array index of the specified item in a pointer array

Synopsis

```
T_BOOL TutPtrAryGetItemIndex(ptr_ary, item, index)
T_PTR_ARY ptr_ary;
T_PTR item;
T_INT4 *index;
```

Arguments

ptr_ary — pointer array searches for the item

item — item to search for in the pointer array

index —pointer to index of the item in the array

Return Values TRUE if item is found in the array; FALSE otherwise.

Diagnostics None

Description TutPtrAryGetItemIndex searches *ptr_ary* for the first occurrence of *item* and returns *index* if the item is found in the array. If the item is not found in the array, TutPtrAryGetItemIndex returns FALSE, and the returned *index* is undefined. Pointer arrays have a zero-based index.

Caution Allocate memory for *index* before calling TutPtrAryGetItemIndex.

See Also TutPtrAryInsertItemAtIndex

Examples This example returns the array index of a given item in the array:

```
T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr, *save_ptr, index;

/* create the pointer array with initial size of 30 */

ptr_ary = TutPtrAryCreate(30);
/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
   /* allocate the space for the next element */
   T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
   *int_ptr = i;
   /* append the item to the array, ignore the index */
   TutPtrAryAppend(ptr_ary, int_ptr, NULL);
   /* save the pointer to the memory location containing number 13 */
   /* for the future */
   if (i == 13) {
   save_ptr = int_ptr;
   }
}

/* perform various operations on the array */
.
.
.

/* now, look up the index of the pointer that points to number 13 */
/* in the array */
if (TutPtrAryGetItemIndex(ptr_ary, save_ptr, &index)) {
   /* we were able to find the item in the array */
   TutOut("The index of the element is %d \n", index);
   }
else {
   TutOut("The expected element was not found in the array. \n");
   TutOut("It was probably removed. \n");
   }
```

TutPtrAryInsertAfter

Name

TutPtrAryInsertAfter — insert a new item in a pointer array after the specified item

Synopsis

```
T_BOOL TutPtrAryInsertAfter(ptr_ary, index_item, new_item)
T_PTR_ARY ptr_ary;
T_PTR index_item;
T_PTR new_item;
```

Arguments

ptr_ary — pointer array to operate on

index_item — item in the array after which the new item is inserted

new_item — new item that is inserted in the array

Return Values

TRUE if the operation is a success, FALSE if *index_item* is not an element in the array, or if *ptr_ary* is invalid.

Diagnostics

None

Description

TutPtrAryInsertAfter inserts *new_item* in *ptr_ary* after the first occurrence of *index_item*. *index_item* must be part of the array for the insertion to work. During this operation, *ptr_ary* may be resized as necessary to accommodate the new element.

TutPtrAryInsertAfter calls the pointer array change callbacks with the reason T_PA_ITEM_INSERTED.

Caution

None

See Also

TutPtrAryAppend, TutPtrAryInsertBefore

Examples This example inserts a new item in the pointer array:

```
T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr, *mark_ptr;

/* create the pointer array with initial size of 30 */

ptr_ary = TutPtrAryCreate(30);
/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
   /* allocate the space for the next element */
   T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
   *int_ptr = i;
   /* append the item to the array, ignore the index */
   TutPtrAryAppend(ptr_ary, int_ptr, NULL);
   if (i == 13) {
     mark_ptr = int_ptr;
   }
}

/* now create and insert a new item after the marked one */
T_MALLOC(int_ptr, sizeof(T_INT4, T_INT4 *);
*int_ptr = 169;
TutPtrAryInsertAfter(ptr_ary, mark_ptr, int_ptr);

/* after the call to TutPtrAryInsertAfter, the array will have
   the following elements:
   0 1 2 3 4 5 6 7 8 9 10 11 12 13 169 14 ... 24

  TutPtrAryGetItemCount returns 26 as the number of items in
  the array */
```

TutPtrAryInsertBefore

Name

TutPtrAryInsertBefore — insert a new item in a pointer array before the specified item

Synopsis

```
T_BOOL TutPtrAryInsertBefore(ptr_ary, index_item, new_item)
T_PTR_ARY ptr_ary;
T_PTR index_item;
T_PTR new_item;
```

Arguments

ptr_ary — pointer array to operate on

index_item — item in the array before the new item is inserted

new_item — new item that is inserted in the array

Return Values

TRUE if the operation is a success, FALSE if *index_item* is not an element in the array, or if the *ptr_ary* is invalid.

Diagnostics

None

Description

TutPtrAryInsertBefore inserts the new item in the array before the first occurrence of the specified index item. *index_item* must be part of the array for the insertion to work. During this operation, the array may be resized as necessary to accommodate the new element.

TutPtrAryInsertBefore calls the pointer array change callbacks with the reason T_PA_ITEM_INSERTED.

Caution

None

See Also

TutPtrAryAppend, TutPtrAryInsertAfter

Examples This example inserts a new item in the pointer array:

```
T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr, *mark_ptr;

/* create the pointer array with initial size of 30 */

ptr_ary = TutPtrAryCreate(30);
/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
   /* allocate the space for the next element */
   T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
   *int_ptr = i;
   /* append the item to the array, ignore the index */
   TutPtrAryAppend(ptr_ary, int_ptr, NULL);
   if (i == 13) {
     mark_ptr = int_ptr;
   }
}

/* now create and insert a new item before the marked one */
T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
*int_ptr = 169;
TutPtrAryInsertBefore(ptr_ary, mark_ptr, int_ptr);

/ * after the call to TutPtrAryInsertBefore, the array will have
   the following elements:
   0 1 2 3 4 5 6 7 8 9 10 11 12 169 13 14 ... 24

  TutPtrAryGetItemCount returns 26 as the number of items in
  the array */
```

TutPtrAryInsertItemAtIndex

Name

TutPtrAryInsertItemAtIndex — insert item in a pointer array at the specified index

Synopsis

```
T_BOOL TutPtrAryInsertItemAtIndex(ptr_ary, new_item, index)
T_PTR_ARY ptr_ary;
T_PTR new_item;
T_INT4 index;
```

Arguments

ptr_ary — pointer array where new item is inserted

new_item — item to be inserted

index — index at which item is inserted

Return Values

TRUE if the operation is a success, FALSE otherwise.

Diagnostics

None

Description

TutPtrAryInsertItemAtIndex inserts a new item in the pointer array at the specified index. The items stored in the rest of the array shifts to accommodate the insertion. The index has a range from zero to the total number of items in the array minus one. Specifying an index outside of those boundaries causes the insertion to fail, and causes TutPtrAryInsertItemAtIndex to return FALSE. Arrays with gaps are not permitted.

TutPtrAryInsertItemAtIndex calls the pointer array change callbacks with the reason T_PA_ITEM_INSERTED.

Caution

None

See Also

TutPtrAryAppend, TutPtrAryReplaceItemAtIndex

Examples This example inserts a new item in the pointer array:

```
T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr;

/* create the pointer array with initial size of 30 */

ptr_ary = TutPtrAryCreate(30);
/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
   /* allocate the space for the next element */
   T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
   *int_ptr = i;
   /* append the item to the array, ignore the index */
   TutPtrAryAppend(ptr_ary, int_ptr, NULL);
}

/* now create and insert a new item at index 13 */
T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
*int_ptr = 169;
TutPtrAryInsertItemAtIndex(ptr_ary, int_ptr, 13);

/* after the call to TutPtrAryInsertItemAtIndex, the array will have
   the following elements:
   0 1 2 3 4 5 6 7 8 9 10 11 12 169 13 14 ... 24

  TutPtrAryGetItemCount returns 26 as the number of items in
  the array */
```

TutPtrAryItemExists

Name TutPtrAryItemExists — verify that the item exists in a pointer array

Synopsis

```
T_BOOL TutPtrAryItemExists(ptr_ary, item)
T_PTR_ARY ptr_ary;
T_PTR item;
```

Arguments *ptr_ary* — pointer array where *item* is looked for

item — item to look for

Return Values TRUE if the specified item is in the array, FALSE otherwise.

Diagnostics None

Description TutPtrAryItemExists searches *ptr_ary* for the first occurrence of the specified item.

Caution None

See Also TutPtrAryGetItemAtIndex

Examples This example verifies that a given item exists in the pointer array:

```
T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr, *mark_ptr;

/* create the pointer array with initial size of 30 */
ptr_ary = TutPtrAryCreate(30);
/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
  /* allocate the space for the next element */
  T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
  *int_ptr = i;
  /* append the item to the array, ignore the index */
  TutPtrAryAppend(ptr_ary, int_ptr, NULL);
  if (i == 13) {
    mark_ptr = int_ptr;
  }
}
/* perform various pointer array operations */
.
.
.
if (TutPtrAryItemExists(ptr_ary, mark_ptr)) {
  TutOut("Marked item is still in the array \n");
}
else {
  TutOut("Marked item has been removed from the array \n");
}
```

TutPtrAryRemove

Name

TutPtrAryRemove — remove the specified item from a pointer array

Synopsis

```
T_BOOL TutPtrAryRemove(ptr_ary, item)
T_PTR_ARY ptr_ary;
T_PTR item;
```

Arguments

ptr_ary — pointer array to perform the operation on

item — item to be removed from the pointer array

Return Values

TRUE if the specified item was found in the array and removed, or FALSE if the item was not in the array.

Diagnostics

None

Description

TutPtrAryRemove removes the specified item from the array. After the removal, array compaction is performed, thus getting rid of the hole that was created as a result of a removal. Note that the removal operation may cause item indices to change.

TutPtrAryRemove calls the pointer array change callbacks with the reason T_PA_ITEM_REMOVED.

Caution

Removal of an item from a very large array can be a time-consuming operation.

TutPtrAryRemove does not free the item removed from the array.

This function only removes the first occurrence of *item*. Use TutPtrAryRemoveAll to remove all occurrences.

See Also

TutPtrAryClear, TutPtrAryCreate, TutPtrAryDestroy, TutPtrAryRemove, TutPtrAryRemoveAll

Examples This example removes an item from the pointer array:

```
T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr, *mark_item, index;

/* create the pointer array with initial size of 30 */

ptr_ary = TutPtrAryCreate(30);
/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
  /* allocate the space for the next element */
  T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
  *int_ptr = i;
  /* append the item to the array, ignore the index */
  TutPtrAryAppend(ptr_ary, int_ptr, NULL);
}

/* perform various operations on the array */
.
.
.
/* get the item at index 13 and remove it*/
if (TutPtrAryGetItemAtIndex(ptr_ary, 13, (T_PTR *)&int_ptr)) {
  TutPtrAryRemove(ptr_ary, int_ptr);
  TutOut("Successfully removed item at index 13 from the
           array \n");
}
else {
  TutOut("Array has shrunk, and has less than 13 elements \n");
}
```

TutPtrAryRemoveAll

Name

TutPtrAryRemoveAll — remove all instances of the specified item from a pointer array

Synopsis

```
T_BOOL TutPtrAryRemoveAll(ptr_ary, item, counter)
T_PTR_ARY ptr_ary;
T_PTR item;
T_INT4 *counter;
```

Arguments

ptr_ary — pointer array to operate on

item — item that is removed from the array

counter — pointer to count of the number of instances of the item removed

Return Values

`TRUE` if at least one instance of the item was found in the array and removed, `FALSE` otherwise.

Diagnostics

None

Description

TutPtrAryRemoveAll removes all instances of the specified item from the pointer array. Because array compaction is performed after every removal, TutPtrAryRemoveAll is not very efficient while performing operations on very large arrays. Note that the removal operation may cause item indices to change.

TutPtrAryRemoveAll calls the pointer array change callbacks with the reason T_PA_ALL_ITEMS_REMOVED.

Caution

It is up to you to allocate the space for the counter. The value stored in the counter is always modified after the call to TutPtrAryRemoveAll.

TutPtrAryRemoveAll does not free the items removed from the array. If these items do need to be deallocated, then use TutPtrAryRemove.

See Also

TutPtrAryRemove, TutPtrAryRemoveAll

Examples This example removes all instances of an item from the pointer array:

```
T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr, *save_ptr, counter;

/* create the pointer array with initial size of 30 */
ptr_ary = TutPtrAryCreate(30);
/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
  /* allocate the space for the next element */
  T_MALLOC(int_ptr, sizeof(int), int*);
  *int_ptr = i;
  /* append the item to the array, ignore the index */
  TutPtrAryAppend(ptr_ary, int_ptr, NULL);
  if (i == 7) {
    /* insert a second copy of the same item in the array */
    TutPtrAryAppend(ptr_ary, int_ptr, NULL);
    save_ptr = int_ptr;
  }
}

.
.
.
/* remove all instances of the saved item from the array */
if (TutPtrAryRemoveAll(ptr_ary, save_ptr, &counter)) {
  TutOut("Removed %d instances of item containing", counter);
  TutOut(" %d from the array\n", *save_ptr);
}
```

TutPtrAryReplaceItemAtIndex

Name

TutPtrAryReplaceItemAtIndex — replace an item in a pointer array at the specified index with a new one

Synopsis

```
T_BOOL TutPtrAryReplaceItemAtIndex(ptr_ary, new_item, index, old_item_return)
T_PTR_ARY ptr_ary;
T_PTR new_item;
T_INt4 index;
T_PTR *old_item_return;
```

Arguments

ptr_ary — pointer array to operate on.

new_item — new item to put in the array.

index — location in the array where the item is put.

old_item_return — pointer to optional item currently stored in the array at the specified location. This argument may be NULL.

Return Values

TRUE if operation is a success, FALSE otherwise.

Diagnostics

None

Description

TutPtrAryReplaceItemAtIndex replaces the element stored in the array at the specified index with the new element. Old element is returned in the optional *old_item_return* argument. The function returns FALSE if the specified index is less than zero or greater than the total number of elements in the array minus one.

TutPtrAryReplaceItemAtIndex calls the pointer array change callbacks with the reason T_PA_ITEM_REPLACED.

Caution

TutPtrAryReplaceItemAtIndex does not free the item replaced in the array.

See Also

TutPtrAryInsertItemAtIndex

Examples This example replaces an item in the pointer array:

```
T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr, *new_ptr;
/* create the pointer array with initial size of 30 */

ptr_ary = TutPtrAryCreate(30);
/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
  /* allocate the space for the next element */
  T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
  *int_ptr = i;
  /* append the item to the array, ignore the index */
  TutPtrAryAppend(ptr_ary, int_ptr, NULL);
}

T_MALLOC(new_ptr, sizeof(T_INT4), T_INT4 *);
*new_ptr = 169;

/* Replace the item at index 13 with the new one */
TutPtrAryReplaceItemAtIndex(ptr_ary, new_ptr, 13,
                            (T_PTR *)&int_ptr);
```

TutPtrArySort

Name

TutPtrArySort — sort a pointer array

Synopsis

```
T_BOOL TutPtrArySort(ptr_ary, compare_func)
T_PTR_ARY ptr_ary;
T_PTR_ARY_CMP_FUNC compare_func;
```

Arguments

ptr_ary — pointer array to sort.

compare_func — user-defined function used to compare the elements in the array while sorting. This argument is mandatory and should never be NULL.

Return Values

TRUE if successful, FALSE otherwise.

Diagnostics

None

Description

TutPtrArySort sorts the elements in *ptr_ary*. The array is sorted using the user-supplied *compare_func*. That function must always be supplied. The comparison function should have return values equivalent to that of the strcmp function — it should return 0 when elements are equal, -1 when the first element is less than the second element, and 1 when the first element is greater than the second element. Note that the sort changes the order in which elements are stored in the array.

TutPtrArySort calls the pointer array change callbacks with the reason T_PA_SORTED.

Caution

None

See Also

TutPtrAryCreate

Examples This example sorts a pointer array:

```
T_INT4 compare_func(item1, item2)
T_PTR item1;
T_PTR item2;
{
  if (*item1 == *item2) {
    return 0;
  }
  if (*item1 < *item2) {
    return -1;
  }

  if (*item1 > *item2) {
    return 1;
  }
}
T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr;

/* create the pointer array with initial size of 30 */

ptr_ary = TutPtrAryCreate(30);
/* seed the random number generator */
srand(255);
/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
  /* allocate the space for the next element */
  T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
  *int_ptr = rand();
  /* append the item to the array, ignore the index */
  TutPtrAryAppend(ptr_ary, int_ptr, NULL);
}

/* sort the array of random integers */
TutPtrArySort(ptr_ary, compare_func);
```

TutPtrArySwapItems

Name TutPtrArySwapItems — exchange the two specified elements in a pointer array

Synopsis

```
T_BOOL TutPtrArySwapItems(ptr_ary, index1, index2)
T_PTR_ARY ptr_ary;
T_INT4 index1;
T_INT4 index2;
```

Arguments

ptr_ary — pointer array to operate on

index1 — index of the first element

index2 — index of the second element

Return Values TRUE if the operation is a success, FALSE otherwise.

Diagnostics None

Description TutPtrArySwapItems exchanges the two items at the specified indices. The function returns TRUE if the operation was successful and FALSE if invalid indices were specified.

TutPtrArySwapItems calls the pointer array change callbacks with the reason T_PA_ITEMS_SWAPPED.

Caution None

See Also TutPtrAryCreate, TutPtrArySort

Examples This example swaps items in a pointer array:

```
T_PTR_ARY ptr_ary;
T_INT4 i, *int_ptr;

/* create the pointer array with initial size of 30 */

ptr_ary = TutPtrAryCreate(30);
/* fill it up with 25 entries */
for (i = 0; i < 25; i++) {
   /* allocate the space for the next element */
   T_MALLOC(int_ptr, sizeof(T_INT4), T_INT4 *);
   *int_ptr = i;
   /* append the item to the array, ignore the index */
   TutPtrAryAppend(ptr_ary, int_ptr, NULL);
}
/* swap items at indices 4 and 12 */
TutPtrArySwapItems(ptr_ary, 4, 12);
```

TutPutenv

Name

TutPutenv — set a SmartSockets environment variable

Synopsis

```
T_BOOL TutPutenv(name_and_value)
T_STR name_and_value;
```

Arguments

name_and_value — both the name and value to be put into the environment, separated by an equal sign (=)

Return Values

TRUE if the environment was successfully updated, FALSE otherwise.

Diagnostics

None

Description

TutPutenv puts a value into the environment. The environment is a list of name / value pairs that is set by the operating system for each process. The parameter *name_and_value* is a string that has both the name of the environment variable and value of the environment, separated by an equal sign (=). For example, to set YOUR_ENV_VAR to something, use this:

```
TutPutenv("YOUR_ENV_VAR=something")
```

The environment only effects the currently running SmartSockets process.

On Windows, TutPutenv puts the value into the runtime associated with the SmartSockets libraries, which may be different that the runtime that the application is running in.

Caution

None

See Also

TutGetenv

Examples

This example puts the value of RTHOME into the environment:

```
if (!TutPutenv("RTHOME=c:\\SmartSockets")) {
  /* error */
}
```

TutRealloc

Name TutRealloc — reallocate memory

Synopsis

```
T_PTR TutRealloc(ptr, size)
T_PTR ptr;
T_UINT4 size;
```

Arguments *ptr* — pointer to the previously allocated block of memory

size — number of bytes to allocate

Return Values Pointer to the reallocated memory block if successful, `NULL` if the requested memory could not be allocated.

Diagnostics None

Description TutRealloc changes the size of the previously allocated memory block to the specified size. Using this function ensures the portability to all the platforms that are currently supported by SmartSockets.

TutRealloc allows *size* to be `0`. This is useful for allocating a non-null placeholder that is resized later with TutRealloc.

All SmartSockets memory management uses the functions TutCalloc, TutFree, TutMalloc, and TutRealloc. You can set debugger breakpoints in these functions to trace SmartSockets memory usage.

Caution TutCalloc, TutFree, TutMalloc, and TutRealloc are not compatible with other memory management functions. Memory allocated by SmartSockets (including memory allocated by TutCalloc, TutMalloc, or TutRealloc) must be reallocated with TutRealloc and freed with TutFree. Attempting to mix memory management functions will result in unpredictable and potentially unstable behavior.

See Also TutFree, TutMalloc, TutCalloc

Examples This example allocates a string 30 characters long and then increases the size of the string to 50 characters:

```
T_STR strptr;

/* the string has to be 30 characters long, so one space has to be */
/* reserved for the terminating NULL */
strptr = (T_STR)TutMalloc(31);
.
.
.
strptr = TutRealloc(strptr, 51);
```

TutRealToStr

Name TutRealToStr — convert a real number to a string using the option Real_Number_Format

Synopsis

```
T_STR TutRealToStr(num)
T_REAL8 num;
```

Arguments *num* — numeric value to be converted to a string

Return Values String representation of *num*.

Diagnostics None

Description TutRealToStr is a convenience function that converts a numeric value to its string representation. TutRealToStr uses the value of the option Real_Number_Format and the standard C function sprintf. SmartSockets uses TutRealToStr to print real numbers.

Caution The string returned by TutRealToStr is a static string managed internally by SmartSockets and should not be modified or deallocated. If a permanent copy of the string is needed, use T_STRDUP or TutStrDup to make one.

See Also TutTimeNumToStr

Examples This example prints two numbers using the format in the option Real_Number_Format:

```
TutOut("SmartSockets time is %s.\n",
TutRealToStr(TutGetCurrentTime()));
TutOut("Wall clock time is %s\n.",
TutRealToStr(TutGetWallTime()));
```

If the value of the option Real_Number_Format was `%f`, the output from the above fragment would be similar to this output:

```
SmartSockets time is 1.230000.
Wall clock time is 769378847.976632.
```

If the value of the option Real_Number_Format was `%g`, the output from the previous fragment would be similar to this output:

```
SmartSockets time is 1.23.
Wall clock time is 7.69379e+08.
```

Note that because the same string storage is reused by TutRealToStr, this code would not work:

```
TutOut("SmartSockets time is %s.\nWall clock time is %s\n",
        TutRealToStr(TutGetCurrentTime()),
        TutRealToStr(TutGetWallTime()));
```

This code causes the same string value to be printed.

TutRwMutexCreate

Name

TutRwMutexCreate — create a multiple-reader / single-writer (R / W) mutex

Synopsis

```
T_RW_MUTEX TutRwMutexCreate(initial_state, read_quota)
T_BOOL initial_state;
T_INT4 read_quota;
```

Arguments

initial_state — `TRUE` if R / W mutex is to be initially write-locked

read_quota — initial number of simultaneous readers to be supported

Return Values

R / W mutex object if successful, `T_INVALID_RW_MUTEX` otherwise.

Diagnostics

If TutRwMutexCreate fails, it returns `T_INVALID_RW_MUTEX` and sets the global SmartSockets error number to one of:

- T_ERR_OS — an operating system error has occurred. Call the function TutErrNumGetOs to obtain the related OS-specific error value.
- T_ERR_TYPE_MISMATCH — the platform does not support threads.
- T_ERR_VAL_INVALID — *initial_state* or *read_quota* was not valid.

Description

TutRwMutexCreate creates a multiple-reader / single-writer mutex. R / W mutexes are useful for protecting a shared resource in scenarios where multiple threads need to examine the state of the resource concurrently, but exclusive access must still be available so that the state can be safely changed.

Normal mutexes have only one locking operation (TutMutexLock) and can only be locked by one thread. R / W mutexes have two different locking operations: TutRwMutexReadLock and TutRwMutexWriteLock. TutRwMutexReadLock is used to obtain a read-lock, making the calling thread a reader-thread. Likewise, TutRwMutexWriteLock is used to obtain a write-lock and makes the caller a writer-thread. Up to T_MAX_READ_QUOTA reader-threads are allowed to lock a R / W mutex concurrently. In contrast, only one writer-thread is allowed at a time, and it excludes access to all other reader-threads and writer-threads for as long as it holds a lock. This defines the priority of locks — write-locks are of greater priority than read-locks.

Because there is no mechanism for taking a lock away from a thread once it has been granted, TutRwMutexWriteLock may need to suspend the calling thread until all of the read-locks have been released. If new read-locks were granted during this time, the thread calling TutRwMutexWriteLock might wait

indefinitely, never getting a chance to update the protected resource. For that reason, waiting writer-threads preempt new reader threads. Threads calling TutRwMutexReadLock are suspended until waiting writer-threads have had their turn (unless the thread calling TutRwMutexReadLock already has a lock).

Like normal mutexes, R/W mutexes can be recursively locked. A thread is never suspended by a recursive lock operation, which for a R/W mutex is specially defined as a lock operation of equal or lesser priority to a lock already granted. This means that a thread already granted a read-lock always succeeds immediately in obtaining another read-lock, and a thread already granted a write-lock succeeds immediately in obtaining either a read-lock or a write-lock. A single thread holding multiple read-locks is still only a single reader-thread so far as the quota is concerned.

If a reader-thread attempts to write-lock the R/W mutex, it automatically has its read-lock upgraded. This means the thread is suspended (if necessary) until all other reader-threads have released their locks, and then is granted a write-lock (upgrades preempt normal write-lock acquisitions and are FIFO with respect to other upgraders). The thread is then a writer-thread and remains a writer-thread until all of its locks have been released. It is possible to explicitly upgrade to a write-lock with TutRwMutexUpgradeLock. TutRwMutexUpgradeLock also takes a timeout, making it useful for acquiring write-locks in time-sensitive scenarios.

Caution None

See Also TutRwMutexDestroy, TutRwMutexReadLock, TutRwMutexUnlock, TutRwMutexUpgradeLock, TutRwMutexWriteLock

Examples This example creates a R/W mutex:

```
T_RW_MUTEX rw_mutex;

rw_mutex = TutRwMutexCreate(FALSE, 4);
if (T_INVALID_RW_MUTEX == rw_mutex) {
  /* error */
}
```

TutRwMutexDestroy

Name TutRwMutexDestroy — destroy a multiple-reader/single-writer mutex

Synopsis

```
T_BOOL TutRwMutexDestroy(rw_mutex)
T_RW_MUTEX rw_mutex;
```

Arguments *rw_mutex* — R/W mutex object to destroy

Return Values TRUE if R/W mutex object was successfully destroyed, FALSE otherwise.

Diagnostics If TutRwMutexDestroy fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_OS — an operating system error has occurred. Call the function TutErrNumGetOs to obtain the related OS-specific error value.
- T_ERR_TYPE_MISMATCH — the platform does not support threads.
- T_ERR_VAL_INVALID — *rw_mutex* was not a valid R/W mutex object.

Description TutRwMutexDestroy destroys a R/W mutex object. All operating system resources associated with the R/W mutex are released.

Caution Destroying a R/W mutex that other threads have locked or are waiting to lock has undefined results.

See Also TutRwMutexCreate

Examples This example destroys a R/W mutex:

```
T_RW_MUTEX rw_mutex;
.
.
.
if (!TutRwMutexDestroy(rw_mutex)) {
  /* error */
}
```

TutRwMutexGetReadQuota

Name

TutRwMutexGetReadQuota — get the current simultaneous reader quota of a R/W mutex

Synopsis

```
T_BOOL TutRwMutexGetReadQuota(rw_mutex, read_quota_return)
T_RW_MUTEX rw_mutex;
T_INT4 *read_quota_return;
```

Arguments

rw_mutex — R/W mutex object to query

read_quota_return — pointer to location of an INT4 field into which the read quota is stored

Return Values

TRUE if query was successful, FALSE otherwise.

Diagnostics

If TutRwMutexGetReadQuota fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *read_quota_return* was NULL
- T_ERR_TYPE_MISMATCH — the platform does not support threads
- T_ERR_VAL_INVALID — *rw_mutex* was not a valid R/W mutex object

Description

TutRwMutexGetReadQuota retrieves the current simultaneous reader quota of a R/W mutex. This quota increases dynamically as necessary to allocate additional resources for concurrent reader-threads. By querying the current value, it is possible to determine the high-water mark of reader-threads that have locked a given R/W mutex concurrently.

Caution

None

See Also

TutRwMutexCreate, TutRwMutexSetReadQuota

Examples

This example gets the reader quota from a R/W mutex:

```
T_RW_MUTEX rw_mutex;
T_INT4 quota;
.
.
.
if (!TutRwMutexGetReadQuota(rw_mutex, &quota)) {
  /* error */
}
```

TutRwMutexReadLock

Name

TutRwMutexReadLock — read-lock a R/W mutex

Synopsis

```
T_BOOL TutRwMutexReadLock(rw_mutex)
T_RW_MUTEX rw_mutex;
```

Arguments

rw_mutex — R/W mutex object to lock

Return Values

TRUE if lock was granted, FALSE otherwise.

Diagnostics

If TutRwMutexReadLock fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_TYPE_MISMATCH — the platform does not support threads
- T_ERR_VAL_INVALID — *rw_mutex* was not a valid R/W mutex object

Description

TutRwMutexReadLock read-locks a R/W mutex. A read-lock implies that the thread may safely examine (but not modify) the state of the resource protected by the R/W mutex. Read-locks may be held simultaneously by more than one thread.

Like normal mutexes, R/W mutexes can be recursively locked. A thread is never suspended by a recursive lock operation. This means that a thread already granted a read-lock always succeeds immediately in obtaining another read-lock, and a thread already granted a write-lock succeeds immediately in obtaining either a read-lock or a write-lock. A single thread holding multiple read-locks is still only a single reader-thread so far as the quota is concerned.

Caution

Every call to TutRwMutexReadLock should be balanced by a subsequent call to TutRwMutexUnlock.

See Also

TutRwMutexCreate, TutRwMutexUnlock, TutRwMutexWriteLock

Examples

This example read-locks a R/W mutex:

```
T_RW_MUTEX rw_mutex;
.
.
.
if (!TutRwMutexReadLock(rw_mutex)) {
  /* error */
}
```

TutRwMutexReadLocked

Name

TutRwMutexReadLocked — check if a R/W mutex is read-locked by the current thread

Synopsis

```
T_BOOL TutRwMutexReadLocked(rw_mutex)
T_RW_MUTEX rw_mutex;
```

Arguments

rw_mutex — R/W mutex object to query

Return Values

TRUE if calling thread has been granted a read-lock, FALSE otherwise.

Diagnostics

If TutRwMutexReadLocked fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_TYPE_MISMATCH — the platform does not support threads
- T_ERR_VAL_INVALID — *rw_mutex* was not a valid R/W mutex object

Description

TutRwMutexReadLocked checks if a R/W mutex is read-locked by the current thread.

Caution

None

See Also

TutRwMutexCreate, TutRwMutexWriteLocked

Examples

This example prints the calling thread's current lock status:

```
void show_rw_lock(rw_mutex)
T_RW_MUTEX rw_mutex;
{
  if (TutRwMutexWriteLocked(rw_mutex)) {
    TutOut("WRITE-LOCK");
  }
  else if (TutRwMutexReadLocked(rw_mutex)) {
    TutOut("READ-LOCK");
  }
  else {
    TutOut("NONE");
  }
}
```

TutRwMutexSetReadQuota

Name

TutRwMutexSetReadQuota — set the simultaneous reader quota of a R/W mutex

Synopsis

```
T_BOOL TutRwMutexSetReadQuota(rw_mutex, read_quota)
T_RW_MUTEX rw_mutex;
T_INT4 read_quota;
```

Arguments

rw_mutex — R/W mutex object to update

read_quota — number of simultaneous readers to be supported

Return Values

TRUE if the operation was successful, FALSE otherwise.

Diagnostics

If TutRwMutexSetReadQuota fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_TYPE_MISMATCH — the platform does not support threads
- T_ERR_VAL_INVALID — *rw_mutex* or *read_quota* was not valid

Description

TutRwMutexSetReadQuota can be used to reset the reader quota assigned to a R/W mutex. The quota tracks the number of reader-threads that have locked a R/W mutex concurrently. In some circumstances it may be desirable to reset the quota to a baseline value after it has grown dynamically, or to increase it in a single operation in anticipation of an increase in concurrent utilization. The valid range is from 1 to T_MAX_READ_QUOTA, inclusive.

Caution

TutRwMutexSetReadQuota obtains a write-lock on the R/W mutex before it changes the quota.

See Also

TutRwMutexCreate, TutRwMutexGetReadQuota, TutRwMutexReadLock

Examples

This example sets the reader quota of the R/W mutex to 8:

```
T_RW_MUTEX rw_mutex;
.
.
.
if (!TutRwMutexSetReadQuota(rw_mutex, 8)) {
  /* error */
}
```

TutRwMutexUnlock

Name TutRwMutexUnlock — release a lock on a R/W mutex

Synopsis

```
T_BOOL TutRwMutexUnlock(rw_mutex)
T_RW_MUTEX rw_mutex;
```

Arguments *rw_mutex* — R/W mutex object to unlock

Return Values TRUE if the operation was successful, FALSE otherwise.

Diagnostics If TutRwMutexUnlock fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_TYPE_MISMATCH — the platform does not support threads
- T_ERR_VAL_INVALID — *rw_mutex* was not a valid R/W mutex object

Description TutRwMutexUnlock releases a lock on a R/W mutex. Only a thread that owns a lock on *rw_mutex* can successfully unlock it.

Caution Every call to TutRwMutexUnlock must balance a previous call to either TutRwMutexReadLock or TutRwMutexWriteLock.

See Also TutRwMutexCreate, TutRwMutexReadLock, TutRwMutexWriteLock

Examples This example unlocks a R/W mutex:

```
T_RW_MUTEX rw_mutex;
.
.
.
if (!TutRwMutexUnlock(rw_mutex)) {
  /* error */
}
```

TutRwMutexUpgradeLock

Name

TutRwMutexUpgradeLock — upgrade a read-lock on a R/W mutex to a write-lock

Synopsis

```
T_BOOL TutRwMutexUpgradeLock(rw_mutex, timeout)
T_RW_MUTEX rw_mutex;
T_REAL8 timeout;
```

Arguments

rw_mutex — R/W mutex object to update

timeout — maximum number of seconds to wait for write-lock

Return Values

`TRUE` if lock was upgraded, `FALSE` otherwise.

Diagnostics

If TutRwMutexUpgradeLock fails, it returns `FALSE` and sets the global SmartSockets error number to one of:

- T_ERR_OS — an operating system error has occurred. Call the function TutErrNumGetOs to obtain the related OS-specific error value.
- T_ERR_TIMEOUT_REACHED — the *timeout* period was reached
- T_ERR_TYPE_MISMATCH — the platform does not support threads
- T_ERR_VAL_INVALID — *rw_mutex* was not a valid R/W mutex object

Description

TutRwMutexUpgradeLock allows a thread to explicitly upgrade from a read-lock on a R/W mutex to a write-lock. The upgrade operation does not acquire a new nested locking level— it does not need to be balanced by a call to TutRwMutexUnlock. TutRwMutexUpgradeLock also takes a timeout, making it useful for acquiring write-locks in time-sensitive scenarios.

To complete this operation, the calling thread may be suspended until all other reader-threads have released their locks (or until a timeout is occurs). Once the R/W mutex is available (upgrades preempt normal write-lock acquisitions and are FIFO with respect to other upgraders), the thread is granted a write-lock. The thread is then a writer-thread and remains a writer-thread until all of its locks have been released.

Caution

None

See Also

TutRwMutexCreate, TutRwMutexReadLock, TutRwMutexWriteLock

Examples This example uses TutRwMutexReadLock, followed by TutRwMutexUpgradeLock, to try for five seconds to obtain a write-lock on the R/W mutex:

```
T_RW_MUTEX rw_mutex
.
.
.
if (!TutRwMutexReadLock(rw_mutex)) {
  /* error */
}
if (!TutRwMutexUpgradeLock(rw_mutex, 5.0)) {
  if (T_ERR_TIMEOUT_REACHED != TutErrNumGet()) {
    /* error */
  }
  /* timeout */
  .
  .
  .
}
.
.
.
```

TutRwMutexWriteLock

Name TutRwMutexWriteLock — write-lock a R/W mutex

Synopsis

```
T_BOOL TutRwMutexWriteLock(rw_mutex)
T_RW_MUTEX rw_mutex;
```

Arguments *rw_mutex* — R/W mutex object to lock

Return Values TRUE if lock was granted, FALSE otherwise.

Diagnostics If TutRwMutexWriteLock fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_TYPE_MISMATCH — the platform does not support threads
- T_ERR_VAL_INVALID — *rw_mutex* was not a valid R/W mutex object

Description TutRwMutexWriteLock write-locks a R/W mutex. A write-lock implies that the thread may safely examine and modify the state of the resource protected by the R/W mutex. Write-locks exclude all other threads from accessing the resource. Because there is no mechanism for taking a lock away from a thread once it has been granted, TutRwMutexWriteLock may need to suspend the calling thread until all of the read-locks have been released. If new read-locks were granted during this time, the thread calling TutRwMutexWriteLock might wait indefinitely, never getting a chance to update the protected resource. For that reason, waiting writer-threads preempt new reader threads.

Like normal mutexes, R/W mutexes can be recursively locked. A thread is never suspended by a recursive lock operation. This means that a thread already granted a read-lock always succeeds immediately in obtaining another read-lock, and a thread already granted a write-lock succeeds immediately in obtaining either a read-lock or a write-lock.

If a reader-thread attempts to write-lock the R/W mutex, it automatically has its read-lock upgraded. This means the thread is suspended (if necessary) until all other reader-threads have released their locks, and then is granted a write-lock (upgrades preempt normal write-lock acquisitions and are FIFO with respect to other upgraders). The thread is then a writer-thread and remains a writer-thread until all of its locks have been released. It is possible to explicitly upgrade to a write-lock with TutRwMutexUpgradeLock.

Caution Every call to TutRwMutexWriteLock should be balanced by a subsequent call to TutRwMutexUnlock.

See Also TutRwMutexCreate, TutRwMutexReadLock, TutRwMutexUnlock, TutRwMutexUpgradeLock

Examples This example write-locks a R/W mutex:

```
T_RW_MUTEX rw_mutex;
.
.
.
if (!TutRwMutexWriteLock(rw_mutex)) {
   /* error */
}
```

TutRwMutexWriteLocked

Name

TutRwMutexWriteLocked — check if a R/W mutex is write-locked by the current thread

Synopsis

```
T_BOOL TutRwMutexWriteLocked(rw_mutex)
T_RW_MUTEX rw_mutex;
```

Arguments

rw_mutex — R/W mutex object to query

Return Values

TRUE if calling thread has been granted a write-lock, FALSE otherwise.

Diagnostics

If TutRwMutexWriteLocked fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_TYPE_MISMATCH — the platform does not support threads
- T_ERR_VAL_INVALID — *rw_mutex* was not a valid R/W mutex object

Description

TutRwMutexWriteLocked checks if a R/W mutex is write-locked by the current thread.

Caution

None

See Also

TutRwMutexCreate, TutRwMutexReadLocked

Examples

This example prints the calling thread's current lock status:

```
void show_rw_lock(rw_mutex)
T_RW_MUTEX rw_mutex;
{
  if (TutRwMutexWriteLocked(rw_mutex)) {
    TutOut("WRITE-LOCK");
  }
  else if (TutRwMutexReadLocked(rw_mutex)) {
    TutOut("READ-LOCK");
  }
  else {
    TutOut("NONE");
  }
}
```

Chapter 10 TutSetCurrentTime - TutXmlSetStr

This chapter describes these SmartSockets utility functions:

- *TutSetCurrentTime*
- *TutSetCurrentTimeStruct*
- *TutSetFileCloseFunc*
- *TutSetFileOpenFunc*
- *TutSetFileReadFunc*
- *TutSetOutFlushFunc*
- *TutSetOutputFunc*
- *TutSleep*
- *TutSocketAccept*
- *TutSocketCheck*
- *TutSocketCreateClientLocal*
- *TutSocketCreateClientTcp*
- *TutSocketCreateServerLocal*
- *TutSocketCreateServerTcp*
- *TutSocketGetBlockMode*
- *TutSocketRecvAll*
- *TutSocketRecvTimeout*
- *TutSocketSetBlockMode*
- *TutStrDup*
- *TutStrLwr*
- *TutStrUpr*
- *TutSystem*
- *TutThreadCreate*
- *TutThreadCreateVa*
- *TutThreadDetach*

- *TutThreadEqual*
- *TutThreadExit*
- *TutThreadSelf*
- *TutThreadWait*
- *TutTimeChangeCbCreate*
- *TutTimeChangeCbLookup*
- *TutTimeCvtCreate*
- *TutTimeCvtDestroy*
- *TutTimeCvtLookup*
- *TutTimeCvtNumToStr*
- *TutTimeCvtStrToNum*
- *TutTimeCvtTraverse*
- *TutTimeNumToStr*
- *TutTimeStrToNum*
- *TutTsdGetValue*
- *TutTsdKeyCreate*
- *TutTsdKeyDestroy*
- *TutTsdSetValue*
- *TutWarning*
- *TutWinSetOutputListBox*
- *TutXmlClone*
- *TutXmlCreate*
- *TutXmlCreateStatic*
- *TutXmlDestroy*
- *TutXmlGetStr*
- *TutXmlSetStr*

TutSetCurrentTime

Name

TutSetCurrentTime — set the current data time for the process

Synopsis

```
T_BOOL TutSetCurrentTime(time_val)
T_REAL8 time_val;
```

Arguments

time_val — double-precision value to use for current time

Return Values

TRUE if successful, FALSE otherwise.

Diagnostics

If TutSetCurrentTime fails, it returns FALSE and sets the global SmartSockets error number to:

- T_ERR_VAL_TOO_SMALL — *time_val* was negative

Description

TutSetCurrentTime sets the current value of data time for the current process. This causes the process to take other actions based on the specific needs of that process. The effects of setting time are determined by the time change callbacks that exist in the process.

Caution

Most processes do not tolerate time reversals. Be careful not to set time backwards unless it is really intended. Negative time values are not allowed.

See Also

TutGetCurrentTime, TutSetCurrentTimeStruct, TutTimeChangeCbCreate

Examples

This example sets the value of data time to the current wall clock time:

```
T_REAL8 wall_time;

wall_time = TutGetWallTime();
if (!TutSetCurrentTime(wall_time)) {
  /* error */
}
```

TutSetCurrentTimeStruct

Name

TutSetCurrentTimeStruct — set the current data time for the process, using the most precise method

Synopsis

```
T_BOOL TutSetCurrentTimeStruct(timep)
T_TIME timep;
```

Arguments

timep — pointer to T_TIME_STRUCT with the time to be set

Return Values

TRUE if successful, FALSE otherwise.

Diagnostics

If TutSetCurrentTimeStruct fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *timep* was NULL
- T_ERR_VAL_TOO_SMALL — *timep* was negative

Description

TutSetCurrentTimeStruct sets the current value of data time for the current process. Whatever processing is required based on time values being set then occurs.

This function provides a more precise method to set the value of time. Depending on data rates and timing requirements, it may not be possible to provide the required accuracy using double-precision time values, so this function allows extra accuracy at the expense of some convenience. Time is maintained internally using both T_TIME_STRUCT and T_REAL8 so that either method can be used, depending on the requirements of the process.

Caution

Most processes do not tolerate time reversals. Be careful not to set time backwards unless it is really intended.

Negative time values are not permitted.

See Also

TutGetCurrentTimeStruct, TutSetCurrentTime, TutTimeChangeCbCreate

Examples

This example sets the value of data time to the current wall clock time:

```
T_TIME_STRUCT wall_time;

if (!TutGetWallTimeStruct(&wall_time)) {
  /* error */
}
if (!TutSetCurrentTimeStruct(&wall_time)) {
  /* error */
}
```

TutSetFileCloseFunc

Name TutSetFileCloseFunc — sets the function to use when closing a file

Synopsis
```
T_BOOL TutSetFileCloseFunc(close_func)
T_FILE_CLOSE_FUNC close_func;
```

Arguments *close_func* — function to use when closing a file

Return Values TRUE if successful, FALSE otherwise.

Diagnostics None

Description TutSetFileCloseFunc allows you to specify a function used to close a file when the process is done with it.

The default close function is fclose. If NULL is specified for *close_func*, the default is used for subsequent close file operations.

Caution None

See Also TutSetFileOpenFunc, TutSetFileReadFunc, TutFileTraverse

Examples This example uses TutSetFileCloseFunc to perform post-processing on a data file:

```
static T_PTR find_name(t_file, arg)
T_FILE t_file;
T_PTR arg;
{

  if (t_file->file == (FILE *)arg) {
    return t_file->fname;
  }

  return NULL;

} /* find_name */

static T_BOOL close_func(file)
FILE *file;
{
  T_STR fname;

  fname = TutFileTraverse(find_name, file);
  if (fname == NULL) {
    /* error */
  }
  fclose(file);
  TutOut("Closing file %s\n", fname);
  postprocess_file(fname);    /* user provided function */

}

.
.
.
TutSetFileCloseFunc(close_func);
.
.
.
```

TutSetFileOpenFunc

Name TutSetFileOpenFunc — set the function to use when opening a file

Synopsis

```
T_BOOL TutSetFileOpenFunc(open_func)
T_FILE_OPEN_FUNC open_func;
```

Arguments *open_func* — function to be used to open a file

Return Values TRUE if successful, FALSE otherwise.

Diagnostics None

Description TutSetFileOpenFunc allows you to specify a function used to open a file, instead of the default TutFOpen. This makes it possible to set up input data to come from a different place, or perform pre-processing on the file before it is read.

If NULL is specified for *open_func*, the default open function TutFOpen is used for subsequent file open operations.

Caution None

See Also TutSetFileCloseFunc, TutSetFileReadFunc, TutFOpen, TutFileTraverse

Examples This example performs pre-processing on a file before opening it:

```
static FILE *open_func(file_name, file_mode)
T_STR file_name;
T_STR file_mode;
{
  if (strcmp(file_mode, "r")) {
    return TutFOpen(file_name, file_mode);
  }
  preprocess_file(file_name);    /* User written pre-processing */
  return TutFOpen(file_name, file_mode);
} /* open_func */
.
.
.
TutSetFileOpenFunc(open_func);
.
.
.
```

TutSetFileReadFunc

Name TutSetFileReadFunc — set the function to use when reading from a file

Synopsis

```
T_BOOL TutSetFileReadFunc(read_func)
T_FILE_READ_FUNC read_func;
```

Arguments *read_func* — function to use when reading data from a file

Return Values TRUE if successful, FALSE otherwise.

Diagnostics None

Description TutSetFileReadFunc specifies a user-defined function (*read_func*) to be used for future file reading operations. The only file read operations where this applies are those that process data in a non-line oriented fashion.

This function make is possible to redirect the input to come from any desired location (such as a database), rather than requiring the data to be stored in a file.

If a user-specified read function has not been specified, the file is read using fread. If a user-defined read function is no longer required, passing NULL for *read_func* causes fread to be used for subsequent reads.

Caution None

See Also TutSetFileOpenFunc, TutSetFileCloseFunc, TutFileTraverse

TutSetOutFlushFunc

Name TutSetOutFlushFunc — set the function to use for TutOutFlush

Synopsis

```
void TutSetOutFlushFunc(flush_func)
T_OUT_VA_FLUSH_FUNC flush_func;
```

Arguments *flush_func* — function to be used by TutOutFlush

Return Values None

Diagnostics None

Description TutSetOutFlushFunc allows you to specify a function that is used by TutOutFlush to display and flush output, instead of the default function. This makes it possible to set up output data to go to a different place, or perform pre-processing before displaying the output.

Caution None

See Also TutGetOutFlushFunc, TutOutFlush

Examples This example pre-processes output before displaying it:

```
static void flush_func(format_str, arg_list)
T_STR format_str;
va_list arg_list;
{
  printf ("In custom flush function.\n");
  vprintf (format_str, arg_list);
  fflush (stdout);
} /* flush_func */
.
.
.
TutSetOutFlushFunc(flush_func);
TutOutFlush("test string\n");
```

TutSetOutputFunc

Name

TutSetOutputFunc — set the function to use for TutOut output

Synopsis

```
void TutSetOutputFunc(output_func)
T_OUT_VA_FUNC output_func;
```

Arguments

output_func — function to be used by TutOut

Return Values

None

Diagnostics

None

Description

TutSetOutputFunc allows you to specify a function that is used by TutOut to display output, instead of the default `vprintf`. This makes it possible to set up output data to go to a different place, or perform pre-processing on the output before it is displayed.

The function *output_func* is called each time with two arguments, the format string and the argument list, much like `vprintf`.

If `NULL` is specified for *output_func*, the default output function `vprintf` is used for subsequent TutOut operations.

For compatibility across platforms, cast using T_ENTRY_VA.

Caution

None

See Also

TutGetOutputFunc, TutOut

Examples

This example discards all TutOut output by using a dummy output function that does nothing:

```
static void T_ENTRY_VA dummy_func(format_str, arg_list)
T_STR format_str;
va_list arg_list;
{
   /* do nothing */
} /* dummy_func */
.
.
.
TutSetOutputFunc(dummy_func);
TutOut("This line will not appear!\n");
.
.
.
```

TutSleep

Name TutSleep — suspend execution for specified interval

Synopsis

```
T_BOOL TutSleep(interval)
T_REAL8 interval;
```

Arguments *interval* — number of seconds to pause

Return Values TRUE if process slept for full interval, FALSE otherwise.

Diagnostics If TutSleep fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_C — a C error has occurred. The function TutErrNumGetC should be called to obtain the related C error value.
- T_ERR_OS — an operating system error has occurred. The function TutErrNumGetOs should be called to obtain the related OS error value.
- T_ERR_SOCKET — a socket error has occurred. The function TutErrNumGetSocket should be called to obtain the related socket error value.
- T_ERR_VAL_TOO_SMALL — the interval was a negative number

Description TutSleep is a convenience function that causes the process to sleep for the specified interval (fractional seconds are legal). TutSleep is roughly equivalent to the UNIX usleep function. TutSleep can be used instead of usleep for portability reasons.

Caution TutSleep is implemented with the function select, which may return prematurely if it receives a signal. Therefore TutSleep may sometimes return early (TutSleep returns FALSE in this case). This is normally not a problem.

See Also None

Examples This example suspends a process for 3.45 seconds:

```
if (!TutSleep(3.45)) {
  /* error */
}
```

TutSocketAccept

Name

TutSocketAccept — accept a socket connection from client

Synopsis

```
T_INT4 TutSocketAccept(server_fd)
T_INT4 server_fd;
```

Arguments

server_fd — server side of socket

Return Values

New socket file descriptor to client if successful, -1 otherwise.

Diagnostics

If TutSocketAccept fails, it returns -1 and sets the global SmartSockets error number to one of:

- T_ERR_C — a C error has occurred. Call function TutErrNumGetC to obtain the related C error value.
- T_ERR_OS — an operating system error has occurred. Call function TutErrNumGetOs to obtain the related OS error value.
- T_ERR_SOCKET — a socket error has occurred. Call function TutErrNumGetSocket to obtain the related socket error value.
- T_ERR_VAL_TOO_SMALL — the server socket file descriptor was a negative number

Description

TutSocketAccept is a convenience function that provides a simplified interface to the accept function. TutSocketAccept accepts a socket connection from a new client and returns a new socket file descriptor that is connected to that client. The *server_fd* socket is left open to accept more clients. TutSocketAccept blocks if accept blocks. (Use TutSocketCheck to check if a new client has connected.)

Caution

None

See Also

TutSocketCheck

Examples

This code uses TutSocketCheck to wait for up to 10 seconds for a new client to connect to a server socket, then accepts the new client with TutSocketAccept:

```
if (!TutSocketCheck(server_fd, T_IO_CHECK_READ, 10.0)) {
  /* error */
}
else {
  client_fd = TutSocketAccept(server_fd);
  if (client_fd == -1) {
    /* error */
  }
}
```

TutSocketCheck

Name TutSocketCheck — check if data can be read from or written to a socket

Synopsis

```
T_BOOL TutSocketCheck(socket_fd, check_mode, timeout)
T_INT4 socket_fd;
T_IO_CHECK_MODE check_mode;
T_REAL8 timeout;
```

Arguments

socket_fd — socket file descriptor to check

check_mode — how to check the socket availability

timeout — maximum number of seconds to wait for availability

Return Values TRUE if data can be read / written, FALSE if timeout reached or error occurred.

Diagnostics If TutSocketCheck fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_C — a C error has occurred. The function TutErrNumGetC should be called to obtain the related C error value.
- T_ERR_OS — an operating system error has occurred. The function TutErrNumGetOs should be called to obtain the related OS error value.
- T_ERR_SOCKET — a socket error has occurred. The function TutErrNumGetSocket should be called to obtain the related socket error value.
- T_ERR_VAL_TOO_SMALL — the socket file descriptor was a negative number or the timeout was a negative number.
- T_ERR_TIMEOUT_REACHED — the timeout period was reached

Description TutSocketCheck is a convenience function that provides a simplified interface to the select function. If *check_mode* is T_IO_CHECK_READ, then TutSocketCheck checks if data is available to be read from the socket. If *check_mode* is T_IO_CHECK_WRITE, then TutSocketCheck checks if data can be written to the socket without blocking. TutSocketCheck returns TRUE if the socket is available and FALSE otherwise. The timeout controls how long TutSocketCheck waits for availability. To wait indefinitely, use the constant T_TIMEOUT_FOREVER for the timeout. To poll the socket without blocking, use a timeout of 0.

Server sockets may use TutSocketCheck to check if a client has connected.

Caution None

See Also TutSocketRecvTimeout

Examples This example uses TutSocketCheck to wait for up to 10 seconds for a new client to connect to a server socket, then accepts the new client with TutSocketAccept:

```
if (!TutSocketCheck(server_fd, T_IO_CHECK_READ, 10.0)) {
  /* error */
}
else {
  client_fd = TutSocketAccept(server_fd);
  if (client_fd == -1) {
  /* error */
  }
}
```

TutSocketCreateClientLocal

Name TutSocketCreateClientLocal — create the client side of a local socket

Synopsis

```
T_INT4 TutSocketCreateClientLocal(file_name)
T_STR file_name;
```

Arguments *file_name* — file name server socket is associated with

Return Values Non-negative socket file descriptor if successful, -1 otherwise.

Diagnostics If TutSocketCreateClientLocal fails, it returns -1 and sets the global SmartSockets error number to one of:

- T_ERR_C — a C error has occurred. The function TutErrNumGetC should be called to obtain the related C error value.
- T_ERR_OS — an operating system error has occurred. The function TutErrNumGetOs should be called to obtain the related OS error value.
- T_ERR_SOCKET — a socket error has occurred. The function TutErrNumGetSocket should be called to obtain the related socket error value.
- T_ERR_NULL_PTR — *file_name* was NULL

Description TutSocketCreateClientLocal is a convenience function that creates a local (UNIX-domain) socket and connects to a server socket that has already been created. Once both ends of the socket are connected, data may be read from the socket with recv and written to the socket with send. The socket can be checked for input with select.

Local sockets are bidirectional and have a file name associated with them. When creating a client/server socket connection, the server side must always be created first (with sockets the client side must have something to connect to).

Caution TutSocketCreateClientLocal arranges for SIGPIPE signals to be ignored.

On UNIX, TutSocketCreateClientLocal sets the close-on-exec bit on the file descriptor, so that any child processes do not share the file descriptor.

The socket file *file_name* is always placed in the directory returned by TutGetSocketDir.

On OpenVMS, TutSocketCreateClientLocal requires TMPMBX privilege.

On Windows, TutSocketCreateClientLocal is a stub function that always fails.

See Also TutGetSocketDir, TutSocketCreateServerLocal

Examples This example creates a socket connection to a server running on the same node, sends a random integer to the server, and receives an integer back from the server:

```
int main(argc, argv)
int argc;
char **argv;
{
  T_INT4 socket_fd;   /* socket connection to server */
  T_INT4 a_number;    /* number written to and read from server */
  T_INT4 status;      /* status code */

  socket_fd = TutSocketCreateClientLocal("file.sock");
  if (socket_fd == -1) {
    TutOut("TutSocketCreateClientLocal failed.\n");
    TutExit(T_EXIT_FAILURE);
  }

 /* send an integer to our server */
  a_number = time(0) % 10; /* nice pseudo-random number */
  TutOut("Sending %d to server process.\n", a_number);
  status = T_C_SEND(socket_fd, (char *)&a_number,
              sizeof(a_number), 0);
  if (status != sizeof(a_number)) {
    TutOut("only sent %d/%d bytes to server.\n",
            status, sizeof(a_number));
    TutExit(T_EXIT_FAILURE);
  }

 /* read an integer from our server */
  status = TutSocketRecvAll(socket_fd, (char *)&a_number,
                                  sizeof(a_number), 0);
  if (status != sizeof(a_number)) {
    TutOut("only read %d/%d bytes from server.\n",
            status, sizeof(a_number));
    TutExit(T_EXIT_FAILURE);
  }
  TutOut("Number from server is %d.\n", a_number);

  TutExit(T_EXIT_SUCCESS);
} /* main */
```

TutSocketCreateClientTcp

Name

TutSocketCreateClientTcp — create the client side of a TCP / IP socket

Synopsis

```
T_INT4 TutSocketCreateClientTcp(node_name, port_num)
T_STR node_name;
T_INT4 port_num;
```

Arguments

node_name — name of TCP / IP node the server socket is on

port_num — port number TCP / IP server socket is associated with

Return Values

Non-negative socket file descriptor if successful, `-1` otherwise.

Diagnostics

If TutSocketCreateClientTcp fails, it returns `-1` and sets the global SmartSockets error number to one of:

- T_ERR_C — a C error has occurred. The function TutErrNumGetC should be called to obtain the related C error value.
- T_ERR_OS — an operating system error has occurred. The function TutErrNumGetOs should be called to obtain the related OS error value.
- T_ERR_SOCKET — a socket error has occurred. The function TutErrNumGetSocket should be called to obtain the related socket error value.
- T_ERR_VAL_TOO_SMALL —*port_num* was less than one
- T_ERR_NULL_PTR — *node_name* was `NULL`

Description

TutSocketCreateClientTcp is a convenience function that creates a TCP / IP (Internet-domain) socket and connects to a server socket that has already been created. Once both ends of the socket are connected, data may be read from the socket with recv and written to the socket with send. The socket can be checked for input with select.

TCP / IP sockets are bidirectional and have a TCP / IP port number associated with them. *node_name* specifies the name of the TCP / IP node where the server socket exists, and *port_num* is used to specify this TCP / IP port number. *node_name* can be a string node name, such as `my_node` or `my_node.my_domain.com`, or an IP address of the form `d.d.d.d`, such as `192.9.200.33`. When creating a client / server socket connection, the server side must always be created first (with sockets the client side must have something to connect to).

When making a connection to the TCP/IP server socket, the connecting process normally blocks for approximately 75 seconds if the server computer is unavailable. To overcome this default 75 second timeout, TutSocketCreateClientTcp uses the option Socket_Connect_Timeout that defaults to 5.0 seconds to determine the maximum amount of time to wait for the connection to complete. If Socket_Connect_Timeout is 0, then TutSocketCreateClientTcp uses the default blocking behavior.

TutSocketCreateClientTcp sets the socket option SO_LINGER to ensure that no data is discarded when the socket is closed.

Caution TutSocketCreateClientTcp arranges for SIGPIPE signals to be ignored.

On UNIX, TutSocketCreateClientTcp sets the close-on-exec bit on the file descriptor, so that any child processes do not share the file descriptor.

If you are using Digital TCP/IP Services for OpenVMS (UCX), TutSocketCreateClientTcp requires NETMBX privilege. If you are using MultiNet TCP/IP on OpenVMS, TutSocketCreateClientTcp does not require any privileges.

On Windows NT, TutSocketCreateClientTcp calls SetHandleInformation with the HANDLE_FLAG_INHERIT flag set to FALSE, so that any child processes do not share the file descriptor.

See Also TutSocketCreateServerTcp

Examples This example creates a socket connection to a server running on the same node, sends a random integer to the server, and receives an integer back from the server:

```
int main(argc, argv)
int argc;
char **argv;
{
  T_STRING node_name;   /* our node name */
  T_INT4 socket_fd;     /* socket connection to server */
  T_INT4 a_number;      /* number written to and read from server */
  T_INT4 status;        /* status code */

 /* create connection to server */
  TutGetNodeName(node_name);
  socket_fd = TutSocketCreateClientTcp(node_name, 4242);
  if (socket_fd == -1) {
    TutOut("TutSocketCreateClientTcp failed.\n");
    TutExit(T_EXIT_FAILURE);
  }
 /* send an integer to our server */
  a_number = time(0) % 10; /* nice pseudo-random number */
  TutOut("Sending %d to server process.\n", a_number);
  status = T_C_SEND(socket_fd, (char *)&a_number,
              sizeof(a_number), 0);
  if (status != sizeof(a_number)) {
    TutOut("only sent %d/%d bytes to server.\n",
            status, sizeof(a_number));
    TutExit(T_EXIT_FAILURE);
  }
 /* read an integer from our server */
  status = TutSocketRecvAll(socket_fd, (char *)&a_number,
                                  sizeof(a_number), 0);
  if (status != sizeof(a_number)) {
    TutOut("only read %d/%d bytes from server.\n",
            status, sizeof(a_number));
    TutExit(T_EXIT_FAILURE);
  }
  TutOut("Number from server is %d.\n", a_number);
  TutExit(T_EXIT_SUCCESS);
} /* main */
```

TutSocketCreateServerLocal

Name

TutSocketCreateServerLocal — create the server side of a local socket

Synopsis

```
T_INT4 TutSocketCreateServerLocal(file_name)
T_STR file_name;
```

Arguments

file_name — name of file to associate with this socket

Return Values

Non-negative socket file descriptor if successful, -1 otherwise.

Diagnostics

If TutSocketCreateServerLocal fails, it returns -1 and sets the global SmartSockets error number to one of:

- T_ERR_C — a C error has occurred. The function TutErrNumGetC should be called to obtain the related C error value.
- T_ERR_OS — an operating system error has occurred. The function TutErrNumGetOs should be called to obtain the related OS error value.
- T_ERR_SOCKET — a socket error has occurred. The function TutErrNumGetSocket should be called to obtain the related socket error value.
- T_ERR_NULL_PTR — *file_name* was NULL

Description

TutSocketCreateServerLocal is a convenience function that creates a local (UNIX-domain) socket, binds an address to it, and listens for new connections on the socket. Once both ends of the socket are connected, data may be read from the socket with recv and written to the socket with send. The socket can be checked for input with select.

Local sockets are bidirectional and have a file name associated with them. When creating a client/server socket connection, the server side must always be created first (with sockets the client side must have something to connect to).

To ensure that the socket address (the file name) is not already in use, TutSocketCreateServerLocal first tries to connect to the local socket specified by *file_name*. If the connection succeeds, then the address is already in use, and TutSocketCreateServerLocal returns FALSE.

Caution

TutSocketCreateServerLocal arranges for SIGPIPE signals to be ignored.

On UNIX, TutSocketCreateServerLocal sets the close-on-exec bit on the file descriptor, so that any child processes do not share the file descriptor.

The socket file *file_name* is always placed in the directory returned by TutGetSocketDir.

On OpenVMS, TutSocketCreateServerLocal requires TMPMBX privilege.

On Windows, TutSocketCreateServerLocal is a stub function that always fails.

See Also TutGetSocketDir, TutSocketAccept, TutSocketCreateClientLocal

Examples This example creates a socket connection to a client, receives an integer from the client, adds one to that integer, and then sends the integer back to the client:

```
int main(argc, argv)
int argc;
char **argv;
{
  T_INT4 server_fd;      /* socket for accepting new connections */
  T_INT4 socket_fd;      /* socket connection to client */
  T_INT4 a_number;       /* number read from and written to client */
  T_INT4 status;         /* status code */

 /* create socket to accept connections on */
  server_fd = TutSocketCreateServerLocal("file.sock");
  if (server_fd == -1) {
    TutOut("TutSocketCreateServerLocal failed.\n");
    TutExit(T_EXIT_FAILURE);
  }

 /* make sure client connects within 30 seconds */
  if (!TutSocketCheck(server_fd, T_IO_CHECK_READ, 30.0)) {
    TutOut("Client did not connect in time.\n");
    TutExit(T_EXIT_FAILURE);
  }

 /* create connection to client */
  socket_fd = TutSocketAccept(server_fd);
  if (socket_fd == -1) {
    TutOut("TutSocketAccept failed.\n");
    TutExit(T_EXIT_FAILURE);
  }

 /* close original socket: we don't need it anymore */
  status = T_C_CLOSESOCKET(server_fd);
  if (status != 0) {
    TutPerror("close");
  }

 /* read an integer from our client */
  status = TutSocketRecvAll(socket_fd, (char *)&a_number,
                                  sizeof(a_number), 0);
  if (status != sizeof(a_number)) {
    TutOut("only read %d/%d bytes from client.\n",
           status, sizeof(a_number));
    TutExit(T_EXIT_FAILURE);
  }
  TutOut("Number from client is %d.\n", a_number);
```

```
  /* increment the integer and send it back to the client */
    a_number++;
    TutOut("Sending number %d back to client.\n", a_number);
    status = T_C_SEND(socket_fd, (char *)&a_number,
                          sizeof(a_number), 0);
    if (status != sizeof(a_number)) {
      TutOut("only wrote %d/%d bytes from client.\n",
               status, sizeof(a_number));
      TutExit(T_EXIT_FAILURE);
    }

    TutExit(T_EXIT_SUCCESS);
  } /* main */
```

TutSocketCreateServerTcp

Name

TutSocketCreateServerTcp — create the server side of a TCP / IP socket

Synopsis

```
T_INT4 TutSocketCreateServerTcp(port_num)
T_INT4 port_num;
```

Arguments

port_num — TCP / IP port number to associate with this socket

Return Values

Non-negative socket file descriptor if successful, `-1` otherwise.

Diagnostics

If TutSocketCreateServerTcp fails, it returns `-1` and sets the global SmartSockets error number to one of:

- T_ERR_C — a C error has occurred. The function TutErrNumGetC should be called to obtain the related C error value.
- T_ERR_OS — an operating system error has occurred. The function TutErrNumGetOs should be called to obtain the related OS error value.
- T_ERR_SOCKET — a socket error has occurred. The function TutErrNumGetSocket should be called to obtain the related socket error value.
- T_ERR_VAL_TOO_SMALL — the port number was less than zero

Description

TutSocketCreateServerTcp is a convenience function that creates a TCP / IP (Internet-domain) socket, binds an address to it, and listens for new connections on the socket. Once both ends of the socket are connected, data may be read from the socket with recv and written to the socket with send. The socket can be checked for input with select.

TCP / IP sockets are bidirectional and have a TCP / IP port number associated with them. *port_num* is used to specify this port number. When creating a client / server socket connection, the server side must always be created first (with sockets the client side must have something to connect to).

TutSocketCreateServerTcp uses the wildcard network address INADDR_ANY to ensure that the socket is visible on all network addresses.

TutSocketCreateServerTcp sets the socket option SO_REUSEADDR to ensure that the TCP / IP port number can be reused quickly if necessary.

TutSocketCreateServerTcp sets the socket option SO_LINGER to ensure that no data is discarded when the socket is closed.

Caution TutSocketCreateServerTcp arranges for SIGPIPE signals to be ignored.

On UNIX, TutSocketCreateServerTcp sets the close-on-exec bit on the file descriptor, so that any child processes do not share the file descriptor.

If you are using Digital TCP/IP Services for OpenVMS, also known as UCX, TutSocketCreateServerTcp requires NETMBX privilege. If you are using MultiNet TCP/IP on OpenVMS, TutSocketCreateServerTcp does not require any privileges.

On Windows NT, TutSocketCreateServerTcp calls SetHandleInformation with the HANDLE_FLAG_INHERIT flag set to FALSE, so that any child processes do not share the file descriptor.

See Also TutSocketAccept, TutSocketCreateClientTcp

Examples This example creates a socket connection to a client, receives an integer from the client, adds one to that integer, and then sends the integer back to the client:

```
int main(argc, argv)
int argc;
char **argv;
{
  T_INT4 server_fd;       /* socket for accepting new connections */
  T_INT4 socket_fd;       /* socket connection to client */
  T_INT4 a_number;        /* number read from and written to client */
  T_INT4 status;          /* status code */

 /* create socket to accept connections on */
  server_fd = TutSocketCreateServerTcp(4242);
  if (server_fd == -1) {
    TutOut("TutSocketCreateServerTcp failed.\n");
    TutExit(T_EXIT_FAILURE);
  }

 /* make sure client connects within 30 seconds */
  if (!TutSocketCheck(server_fd, T_IO_CHECK_READ, 30.0)) {
    TutOut("Client did not connect in time.\n");
    TutExit(T_EXIT_FAILURE);
  }

 /* create connection to client */
  socket_fd = TutSocketAccept(server_fd);
  if (socket_fd == -1) {
    TutOut("TutSocketAccept failed.\n");
    TutExit(T_EXIT_FAILURE);
  }

 /* close original socket: we don't need it anymore */
  status = T_C_CLOSESOCKET(server_fd);
  if (status != 0) {
    TutPerror("close");
  }
```

```
  /* read an integer from our client */
    status = TutSocketRecvAll(socket_fd, (char *)&a_number,
                                    sizeof(a_number), 0);
    if (status != sizeof(a_number)) {
      TutOut("only read %d/%d bytes from client.\n",
             status, sizeof(a_number));
      TutExit(T_EXIT_FAILURE);
    }
    TutOut("Number from client is %d.\n", a_number);

  /* increment the integer and send it back to the client */
    a_number++;
    TutOut("Sending number %d back to client.\n", a_number);
    status = T_C_SEND(socket_fd, (char *)&a_number,
                          sizeof(a_number), 0);
    if (status != sizeof(a_number)) {
      TutOut("only wrote %d/%d bytes from client.\n",
             status, sizeof(a_number));
      TutExit(T_EXIT_FAILURE);
    }

    TutExit(T_EXIT_SUCCESS);
  } /* main */
```

TutSocketGetBlockMode

Name

TutSocketGetBlockMode — get the block mode of a socket

Synopsis

```
T_BOOL TutSocketGetBlockMode(socket_fd, block_mode_return)
T_INT4 socket_fd;
T_BOOL *block_mode_return;
```

Arguments

socket_fd — socket file descriptor to get block mode for

block_mode_return — pointer to storage for block mode

Return Values

TRUE if block mode was successfully retrieved from socket, FALSE otherwise.

Diagnostics

If TutSocketGetBlockMode fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *block_mode_return* was NULL
- T_ERR_C — a C error has occurred. The function TutErrNumGetC should be called to obtain the related C error value.
- T_ERR_OS — an operating system error has occurred. The function TutErrNumGetOs should be called to obtain the related OS error value.
- T_ERR_SOCKET — a socket error has occurred. The function TutErrNumGetSocket should be called to obtain the related socket error value.
- T_ERR_VAL_TOO_SMALL — *socket_fd* was a negative number

Description

TutSocketGetBlockMode gets the block mode of a socket by calling fcntl to get the appropriate socket status flag. The status flag is system-dependent and is one of O_NONBLOCK, O_NDELAY, or FNDELAY. The block mode controls the behavior of I/O operations that cannot complete immediately. The most common operations are sending and receiving data.

The default block mode is TRUE. In blocking mode, an attempt to send data to a socket does not finish until all data has been sent or an error occurs. An attempt to receive data from a socket does not finish until data has been received or an error occurs.

A block mode of FALSE is also known as non-blocking mode. In non-blocking mode, an attempt to send data to a socket finishes when all data (that can be sent immediately) is sent. An attempt to receive data from a socket finishes when all data (that can be received immediately) is received.

Caution

If TutSocketGetBlockMode returns FALSE, it does not store a value in *block_mode_return*.

See Also TutSocketSetBlockMode; see the *TIBCO SmartSockets Application Programming Interface* reference for detailed information on TipcConnGetBlockMode.

Examples This example gets the block mode of a socket and prints it:

```
if (!TutSocketGetBlockMode(socket_fd, &block_mode)) {
   /* error */
}

TutOut("The socket block mode is %d.\n", block_mode);
```

TutSocketRecvAll

Name

TutSocketRecvAll — read multiple times from a socket until all data has been received

Synopsis

```
T_INT4 TutSocketRecvAll(fd, buf, size, flags)
T_INT4 fd;
T_PTR buf;
T_INT4 size;
T_INT4 flags;
```

Arguments

fd — socket file descriptor to read from

buf — buffer to put data into

size — number of bytes to read

flags — flags for recv

Return Values

Number of bytes read if successful, `-1` otherwise.

Diagnostics

If TutSocketRecvAll fails, it returns `-1` and sets the global SmartSockets error number to one of:

- T_ERR_C — a C error has occurred. The function TutErrNumGetC should be called to obtain the related C error value.
- T_ERR_OS — an operating system error has occurred. The function TutErrNumGetOs should be called to obtain the related OS error value.
- T_ERR_SOCKET — a socket error has occurred. The function TutErrNumGetSocket should be called to obtain the related socket error value.
- T_ERR_NULL_PTR — *buf* was `NULL`
- T_ERR_VAL_TOO_SMALL — *fd* was a negative number, or *size* was less than `1`

Description

TutSocketRecvAll is a convenience function that keeps reading data from a socket until the desired number of bytes have been read, an error occurs, or EOF is reached. Unlike files, sockets can have data broken up into smaller packets for transmission over a network, and thus multiple reads are often necessary to allow time for all packets to arrive at the destination.

This function call normally blocks until *size* bytes have been read from *socket*. This behavior can be modified by setting the block mode using TutSocketSetBlockMode. The function TutSocketCheck can be used to determine if data is available to be read before calling TutSocketRecvAll.

The *flags* argument can be any flags accepted by the recv function. Refer to your operating system manuals for full information on these flags. Specify 0 for *flags* to use the default behavior.

Caution None

See Also TutSocketRecvTimeout, TutSocketCheck, TutSocketSetBlockMode

Examples This example shows how to check if the desired number of bytes are read with TutSocketRecvAll:

```
bytes_read = TutSocketRecvAll(socket, buf, size, 0);
if (bytes_read != size) {
  /* error */
}
```

TutSocketRecvTimeout

Name

TutSocketRecvTimeout — read data from socket with a timeout in seconds

Synopsis

```
T_INT4 TutSocketRecvTimeout(socket_fd, buf, size, flags, timeout,
                            timeout_reached_return)
T_INT4 socket_fd;
T_PTR buf;
T_INT4 size;
T_INT4 flags;
T_REAL8 timeout;
T_BOOL *timeout_reached_return;
```

Arguments

socket_fd — socket file descriptor to read from

buf — buffer to put data into

size — number of bytes to read

flags — flags for recv

timeout — maximum number of seconds to wait for *size* bytes of data to arrive

timeout_reached_return — storage for whether or not timeout was reached

Return Values

Number of bytes read if successful, `-1` otherwise.

Diagnostics

If TutSocketRecvTimeout fails, it returns `-1` and sets the global SmartSockets error number to one of:

- T_ERR_C — a C error has occurred. The function TutErrNumGetC should be called to obtain the related C error value.
- T_ERR_OS — an operating system error has occurred. The function TutErrNumGetOs should be called to obtain the related OS error value.
- T_ERR_SOCKET — a socket error has occurred. The function TutErrNumGetSocket should be called to obtain the related socket error value.
- T_ERR_NULL_PTR — *timeout_reached_return* or *buf* was `NULL`
- T_ERR_VAL_TOO_SMALL — *socket_fd* was a negative number, or *size* was less than `1`

Description

TutSocketRecvTimeout is a convenience function that reads data from a socket, subject to a timeout period. TutSocketRecvTimeout is useful when a program needs to have an upper limit on how long it waits for data to arrive on a socket. RTserver, for example, cannot block forever waiting for a new client to send connection information. TutSocketRecvTimeout keeps reading from the socket until the specified number of bytes have been received, or *timeout* has been reached.

If TutSocketRecvTimeout returns `0`, then either *timeout* has been reached, or the socket peer has closed. The value TutSocketRecvTimeout stores in *timeout_reached_return* can be used to distinguish between these two cases.

The *flags* argument can be any flags accepted by the recv function. Refer to your operating system manuals for full information on these flags. Specify `0` for *flags* to use the default behavior.

Caution

TutSocketRecvTimeout is implemented with recv and select. If TutSocketRecvTimeout has to read multiple times from the socket, it reuses *timeout* each time it reads from the socket. For example, if TutSocketRecvTimeout has to read 5 times from the socket, up to 5 times *timeout* seconds could pass before TutSocketRecvTimeout would think *timeout* had been reached.

See Also

TutSocketRecvAll

Examples

This example distinguishes between `EOF` and the *timeout* period being reached:

```
bytes_read = TutSocketRecvTimeout(socket, buf, size, 0,
                                  5.0,  &timeout_reached);
TutOut("bytes read = %d, timeout_reached = %d\n",
       bytes_read, timeout_reached);
if (bytes_read == 0) {
  if (timeout_reached) {
    TutOut("Did not read any data in time.\n");
  }
  else {
    TutOut("The socket peer closed (EOF).\n");
  }
}
```

TutSocketSetBlockMode

Name

TutSocketSetBlockMode — set the block mode of a socket

Synopsis

```
T_BOOL TutSocketSetBlockMode(socket_fd, block_mode)
T_INT4 socket_fd;
T_BOOL block_mode;
```

Arguments

socket_fd — socket file descriptor to set block mode for

block_mode — block mode to apply to *socket_fd*

Return Values

TRUE if block mode was successfully set for socket, FALSE otherwise.

Diagnostics

If TutSocketSetBlockMode fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_C — a C error has occurred. The function TutErrNumGetC should be called to obtain the related C error value.
- T_ERR_OS — an operating system error has occurred. The function TutErrNumGetOs should be called to obtain the related OS error value.
- T_ERR_SOCKET — a socket error has occurred. The function TutErrNumGetSocket should be called to obtain the related socket error value.
- T_ERR_VAL_TOO_SMALL — the socket file descriptor was a negative number
- T_ERR_VAL_INVALID — the block mode was not TRUE or FALSE

Description

TutSocketSetBlockMode sets the block mode of *socket_fd*. See TutSocketGetBlockMode on page 268 for more information on socket block modes. A value of TRUE for *block_mode* makes I/O operations blocking, and FALSE makes I/O operations non-blocking.

Caution

None

See Also

TutSocketSetBlockMode, TutSocketGetBlockMode

Examples

This example sets the block mode of a socket to FALSE:

```
if (!TutSocketSetBlockMode(socket_fd, FALSE)) {
  /* error */
}
```

TutStrDup

Name TutStrDup — copies a string

Synopsis

```
T_STR TutStrDup(str)
T_STR str;
```

Arguments *str* — string to be duplicated

Return Values Pointer to the duplicate string.

Diagnostics None

Description TutStrDup makes a copy of the character string. It takes a pointer to a string as an argument and returns a pointer to the copy of that string. When the duplicate string is no longer needed, it should be disposed of with the T_FREE macro.

Caution Failure to properly dispose of the duplicate string creates a memory leak.

See Also TutStrUpr, TutStrLwr

Examples This example duplicates a string:

```
T_CHAR strptr[80],
T_STR dup_string;

strcpy(strptr, "This is a duped string");
dup_string = TutStrDup(strptr);
TutOut("%s\n", dup_string);
.
.
.
```

The TutOut statement produces this output:

```
This is a duped string
```

TutStrLwr

Name

TutStrLwr — convert a string to lowercase

Synopsis

```
T_STR TutStrLwr(dest, src)
T_STR dest;
T_STR src;
```

Arguments

dest — converted string

src — original (source) string

Return Values

Pointer to the converted string.

Diagnostics

None

Description

TutStrLwr converts a character string from uppercase to lowercase. It takes two character pointers as arguments. The first pointer points to the pre-allocated array that should be large enough to hold the converted string. The second pointer points to the string to be converted to lowercase.

Caution

Always ensure that an array has been allocated and is large enough to hold the converted string, before calling TutStrLwr.

See Also

TutStrUpr

Examples

This example converts a string to lowercase:

```
T_CHAR strptr[80];

strcpy(strptr, "ThiS Is A MixeD CasE StrinG");
TutStrLwr(strptr, strptr);
TutOut("%s\n", strptr);
.
.
.
```

The TutOut statement produces this output:

```
this is a mixed case string
```

TutStrUpr

Name TutStrUpr — convert a string to uppercase

Synopsis

```
T_STR TutStrUpr(dest, src)
T_STR dest;
T_STR src;
```

Arguments *dest* — converted string

src — original (source) string

Return Values Pointer to the converted string.

Diagnostics None

Description TutStrUpr converts a character string from lowercase to uppercase. It takes two character pointers as arguments. The first pointer, *dest*, points to the pre-allocated array that should be big enough to hold the converted string. The second pointer, *src*, points to the string to be converted to uppercase.

Caution Ensure that *dest* has been allocated and is large enough to hold the converted string before calling TutStrUpr.

See Also TutStrLwr

Examples This example converts a string to uppercase:

```
T_STR strptr;

strptr = "ThiS Is A MixeD CasE StrinG"
TutStrUpr(strptr, strptr);
TutOut("%s\n", strptr);
.
.
.
```

The TutOut statement produces this output:

```
THIS IS A MIXED CASE STRING
```

TutSystem

Name

TutSystem — execute a shell command

Synopsis

```
T_INT4 TutSystem(command)
T_STR command;
```

Arguments

command — command to be executed

Return Values

Exit status of *command*: `0` if successful, non-zero otherwise.

Diagnostics

None

Description

TutSystem passes a *command* to be executed by an OS command processor. The Bourne shell (`/bin/sh`) is used on UNIX, DCL is used on OpenVMS, and the Windows command prompt is used on Windows. On MVS, TSO or SAS execution services ultimately processes the request. TutSystem returns the exit status of *command* (on OpenVMS the exit status `1` is mapped to `0` to emulate UNIX semantics).

TutSystem on UNIX and MVS simply calls the function system. On OpenVMS, TutSystem calls system unless the command string ends with an ampersand (&) character. In this case, TutSystem emulates a UNIX shell by calling LIB$SPAWN with the NOWAIT bit set (on UNIX this is called running the command in the background). On Windows, TutSystem emulates UNIX to accommodate the limitations of COMMAND.COM on Windows. This is useful for writing code that is portable between OpenVMS, UNIX, and Windows.

Caution

If *command* is run in the background, TutSystem does not return the exit status of *command*, because doing so would require waiting for *command* to complete. In this case, TutSystem always returns `0` unless something goes wrong starting *command*.

If the command is not run in background, TutSystem does not return until the command completes.

See Also

TutCommandParseTypedStr

Examples

This example starts a run-time RTmon in the background:

```
T_INT4 status;
status = TutSystem("rtmon -runtime &");
if (status != 0) {
  TutOut("ERROR: could not start rtmon: errors was %d\n", status);
}
```

TutThreadCreate

Name

TutThreadCreate — create a thread

Synopsis

```
T_THREAD TutThreadCreate(thread_func, thread_arg, ...)
T_THREAD_FUNC thread_func;
T_PTR thread_arg;
```

Arguments

thread_func — function to call from the new thread

thread_arg — argument to pass to *thread_func*

additional arguments — NULL-terminated list of key-value argument pairs

Return Values

Thread handle if successful, T_INVALID_THREAD otherwise.

Diagnostics

If TutThreadCreate fails, it returns T_INVALID_THREAD and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *thread_func* was NULL
- T_ERR_OS — an operating system error has occurred. Call the function TutErrNumGetOs to obtain the related OS error value.
- T_ERR_TYPE_MISMATCH — the platform does not support threads

Description

TutThreadCreate is used to create a new operating system thread executing *thread_func* with the supplied argument. When *thread_func* returns, or explicitly calls TutThreadExit, the operating system thread exits and the function's return value, or the argument passed to TutThreadExit, is available to another thread with TutThreadWait.

The additional arguments accepted by this function allow key-value pairs to be defined that modify the platform-specific thread creation arguments passed to the operating system. At present, there are no such key-value pairs defined. Callers should simply pass NULL to indicate no additional arguments.

Caution

Threads created by TutThreadCreate or TutThreadCreateVa should always be exited with a call to TutThreadExit (or by returning from *thread_func*, which implicitly calls TutThreadExit for the thread).

Before you use any thread function, you must call TipcInitThreads to initialize SmartSockets thread synchronization.

The T_ENTRY declaration specifier is required in the definition of all thread functions as well as their prototypes.

See Also TutThreadCreateVa, TutThreadEqual, TutThreadExit, TutThreadSelf, TutThreadWait; see the *TIBCO SmartSockets Application Programming Interface* reference for detailed information on TipcInitThreads

Examples This example creates three threads, each executing a function that prints a simple message identifying the thread and then exits. It then waits until all three threads have finished.

```
T_PTR T_ENTRY my_thread_func(T_PTR arg)
{
  TutOut("This output comes from thread %d\n", *((T_INT4 *)arg);
  return NULL;
}

int main()
{
  T_THREAD threads[3];
  T_INT4 args[3];
  T_INT4 i;

  if (!TipcInitThreads())
    exit (T_EXIT_FAILURE);

  for (i = 0; i < 3; i++) {
    args[i] = i + 1;
    threads[i] = TutThreadCreate(my_thread_func, &args[i], NULL);
    if (TutThreadEqual(T_INVALID_THREAD, threads[i])) {
      /* error */
    }
  }
  for (i = 0; i < 3; i++) {
    T_PTR result;

    if (!TutThreadWait(threads[i], &result)) {
      /* error */
    }
  }
  return T_EXIT_SUCCESS;
}
```

TutThreadCreateVa

Name TutThreadCreateVa — create a thread (va_list version)

Synopsis

```
T_THREAD TutThreadCreateVa(thread_func, thread_arg, var_arg_list)
T_THREAD_FUNC thread_func;
T_PTR thread_arg;
va_list var_arg_list;
```

Arguments

thread_func — function to call from the new thread

thread_arg — argument to pass to *thread_func*

var_arg_list — argument list

Return Values Thread handle if successful, T_INVALID_THREAD otherwise.

Diagnostics If TutThreadCreateVa fails, it returns T_INVALID_THREAD and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *thread_func* was NULL
- T_ERR_OS — an operating system error has occurred. Call the function TutErrNumGetOs to obtain the related OS error value.
- T_ERR_TYPE_MISMATCH — the platform does not support threads

Description TutThreadCreateVa is the non-varargs version of TutThreadCreate. (TutThreadCreate actually calls TutThreadCreateVa.) TutThreadCreateVa can be used to implement a varargs function.

In the C language, a varargs function (one that takes a variable number of arguments) cannot call another varargs function with the original arguments. A varargs function usually calls a helper function that takes a fixed number of arguments, with the last argument usually of type va_list. TutThreadCreateVa is TutThreadCreate's helper function.

Caution Threads created by TutThreadCreate or TutThreadCreateVa should always be exited with a call to TutThreadExit or by returning from *thread_func*, which implicitly calls TutThreadExit for the thread.

See Also TutThreadCreate, TutThreadExit

Examples This example implements a minimal replacement for TutThreadCreate (primarily omitting error checking):

```
T_THREAD my_thread_create(T_THREAD_FUNC thread_func,
                  T_PTR thread_arg, ...)
{
  va_list var_arg_list;
  T_THREAD result;

  va_start(var_arg_list, thread_arg);
  result = TutThreadCreateVa(thread_func, thread_arg,
                             var_arg_list);
  va_end(var_arg_list);
  return result;
}
```

TutThreadDetach

Name TutThreadDetach — allow a thread to run to completion without requiring the operating system to remember its exit code

Synopsis `T_BOOL TutThreadDetach(T_THREAD` *thread)*

Arguments *thread* — the thread to detach

Return Values `TRUE` if successful, `FALSE` otherwise.

Diagnostics If TutThreadDetach fails, it returns `FALSE` and sets the global SmartSockets error number to one of:

- T_ERR_OS — an operating system error has occurred. Call the function TutErrNumGetOs to obtain the related OS error value
- T_ERR_TYPE_MISMATCH — the platform does not support threads

Description Normally, when a thread exits, the operating system must remember the thread's exit code, which can be retrieved with TutThreadWait. This requires that every thread created with TutThreadCreate be subsequently joined with TutThreadWait. If it is not, it leaks a small amount of system resources. TutThreadDetach instructs the operating system not to remember the thread's exit code. This means that if someone calls TutThreadDetach, there is no need to call TutThreadWait. When the detached thread exits, all system resources for that thread are cleaned up.

Caution Do not call TutThreadWait on a thread that has been detached using TutThreadDetach.

See Also TutThreadCreate, TutThreadWait, TutThreadDetach, TutThreadExit

Examples This example creates three threads and detaches them. Each thread executes a function that prints a simple message identifying the thread, then exits.

```
T_PTR T_ENTRY my_thread_func(T_PTR arg)
{
  TutOut("This output comes from thread %d\n", *((T_INT4 *)arg);
  return NULL;
}

int func()
{
  T_THREAD threads[3];
  T_INT4 args[3];
  T_INT4 i;

  if (!TipcInitThreads())
    exit (T_EXIT_FAILURE);

  for (i = 0; i < 3; i++) {
    args[i] = i + 1;
    threads[i] = TutThreadCreate(my_thread_func, &args[i], NULL);
    if (TutThreadEqual(T_INVALID_THREAD, threads[i])) {
      /* error */
    }
    if (!TutThreadDetach(threads[i])) {
      /* error */
    }
  }
}
```

TutThreadEqual

Name

TutThreadEqual — compare two thread handles for equality

Synopsis

```
T_BOOL TutThreadEqual(thread_1, thread_2)
T_THREAD thread_1;
T_THREAD thread_2;
```

Arguments

thread_1 — first thread handle to compare

thread_2 — second thread handle to compare

Return Values

TRUE if the handles are equal, FALSE otherwise.

Diagnostics

None

Description

TutThreadEqual compares two T_THREAD values for equality. This function is necessary because some platforms use non-scalar values, such as structures, as thread handles.

Caution

Comparing two thread handles with C's equality operator (==) is non-portable.

See Also

TutThreadCreate

Examples

This example checks for a successful return value from TutThreadCreate:

```
T_THREAD thread;

thread = TutThreadCreate(my_thread_func, NULL, NULL);
if (TutThreadEqual(T_INVALID_THREAD, thread)) {
  /* error */
}
```

TutThreadExit

Name TutThreadExit — exit from a thread

Synopsis

```
void TutThreadExit(exit_code)
T_PTR exit_code;
```

Arguments *exit_code* — thread exit status value

Return Values None

Diagnostics None

Description TutThreadExit exits the thread from which it is called (it does not return to the caller). The *exit_code* provided can be accessed by another thread with the TutThreadWait function.

Caution TutThreadExit must be called only from threads that are created by TutThreadCreate or TutThreadCreateVa.

See Also TutThreadCreate, TutThreadCreateVa, TutThreadWait

Examples This example returns a pointer to a heap-allocated two-element integer array as the calling thread's exit status:

```
T_INT4 *result_array = TutMalloc(sizeof(T_INT4) * 2);

result_array[0] = 10;
result_array[1] = 13;
TutThreadExit(result_array);
/* not reached */
```

TutThreadSelf

Name TutThreadSelf — return the T_THREAD of the current thread

Synopsis

```
T_THREAD TutThreadSelf()
```

Arguments None

Return Values T_THREAD value for the calling thread.

Diagnostics None

Description TutThreadSelf returns a T_THREAD value for the calling thread. For platforms that do not support threads, this is always equal to T_INVALID_THREAD. This value is useful primarily for comparison with other T_THREAD values as a means of establishing the identity of the calling thread.

Caution Comparing two thread handles with the C equality operator (==) is non-portable. TutThreadEqual compares two T_THREAD values for equality. This function is necessary because some platforms use non-scalar values, such as structures, as thread handles.

See Also TutThreadCreate, TutThreadDetach

Examples This example checks to see if the calling thread is the arbitrarily-designated privileged thread:

```
static T_THREAD privileged_thread;

T_BOOL check_privileged()
{
  T_THREAD caller_thread = TutThreadSelf();
  return TutThreadEqual(privileged_thread, caller_thread);
}
```

TutThreadWait

Name

TutThreadWait — wait for another thread to exit

Synopsis

```
T_BOOL TutThreadWait(thread, exit_return)
T_THREAD thread;
T_PTR *exit_return;
```

Arguments

thread — thread to join with

exit_return — location of a T_PTR into which that *thread*'s exit value is stored

Return Values

`TRUE` if the attempt to wait was successful, `FALSE` otherwise.

Diagnostics

If TutThreadWait fails, it returns `FALSE` and sets the global SmartSockets error number to one of:

- T_ERR_OS — an operating system error has occurred. Call the function TutErrNumGetOs to obtain the related OS error value.
- T_ERR_TYPE_MISMATCH — the platform does not support threads

Description

TutThreadWait attempts to suspend the calling thread until the (other) thread designated by *thread* has exited. It then retrieves *thread*'s exit value and stores it in the location designated by *exit_return*. Finally, all resources associated with (the now defunct) `thread` are released. This implies that only one calling thread can successfully do a TutThreadWait on a given (other) thread. The normal practice is to make any thread that creates child threads responsible for waiting for each of its children to exit.

If the thread designated by *thread* has already exited before TutThreadWait is first called, the function succeeds immediately, without suspending the calling thread, and stores the saved exit value in the location designated by *exit_return*.

Caution

A thread created by TutThreadCreate or TutThreadCreateVa that is not joined with TutThreadWait leaks a small amount of system resources. If this is done systematically by a long-lived process, eventually that process may exhaust an operating-system-dependent limit and no may longer be able to create new threads.

To create detached threads (for example, threads that free all of their associated resources immediately upon completion), call either TutThreadWait or TutThreadDetach.

See Also

TutThreadCreate, TutThreadCreateVa, TutThreadExit, TutThreadDetach

Examples This example expects a pointer to a heap-allocated two-element integer array as the calling thread's exit status:

```
T_THREAD thread;
T_INT4 *result_array;
.
.
.
if (!TutThreadWait(thread, &result_array)) {
  /* error */
}
TutOut("Result array => [ %d, %d ]\n",
       result_array[0], result_array[1]);
TutFree(result_array);
```

TutTimeChangeCbCreate

Name

TutTimeChangeCbCreate — create a time change callback

Synopsis

```
T_CB TutTimeChangeCbCreate(func, arg)
T_TIME_CHANGE_CB_FUNC func;
T_PTR arg;
```

Arguments

func — time change callback function to be called

arg — argument to be passed to the callback function

Return Values

Returns pointer to created callback. NULL is returned if callback could not be created.

Diagnostics

None

Description

TutTimeChangeCbCreate creates a time change callback that is called whenever the value of data time changes. The value of data time is changed with the functions TutSetCurrentTime and TutSetCurrentTimeStruct. Whenever the value of time is changed, *func* is called with *arg* as the callback argument.

Caution

None

See Also

TutTimeChangeCbLookup, TutSetCurrentTime, TutSetCurrentTimeStruct

Examples

This example creates a time change callback to print the value of time when it changes:

```
void T_ENTRY time_change_cb(dummy, data, arg)
T_PTR dummy; /* not used */
T_TIME_CHANGE_CB_DATA data;
T_PTR arg;
{
  TutOut("Data time is: %d.%06d\n",
        data->time->sec, data->time->usec);
}

if (TutTimeChangeCbCreate(time_change_cb, NULL) == NULL) {
  /* error */
}
```

TutTimeChangeCbLookup

Name TutTimeChangeCbLookup — look up a time change callback

Synopsis

```
T_CB TutTimeChangeCbLookup(func, arg)
T_TIME_CHANGE_CB_FUNC func;
T_PTR arg;
```

Arguments *func* — time change callback function to be looked up

arg — argument to be passed to the callback function

Return Values Returns pointer to callback if successful, NULL otherwise.

Diagnostics None

Description TutTimeChangeCbLookup looks up a time change callback based on *func* and *arg*. It returns the callback that exists using the specified callback function and argument.

Caution None

See Also TutTimeChangeCbCreate, TutSetCurrentTime, TutSetCurrentTimeStruct

Examples This example creates a time change callback and then looks it up to verify that the function is working properly:

```
void T_ENTRY time_change_cb(dummy, data, arg)
T_PTR dummy; /* not used */
T_TIME_CHANGE_CB_DATA data;
T_PTR arg;
{
  TutOut("Data time is: %d.%06d\n",
       data->time->sec, data->time->usec);
}

T_CB created_cb;
T_CB found_cb;
created_cb = TutTimeChangeCbCreate(time_change_cb, NULL);
if (created_cb == NULL) {
  /* error */
}

found_cb = TutTimeChangeCbLookup(time_change_cb, NULL);
if (found_cb != created_cb) {
  /* error */
}
```

TutTimeCvtCreate

Name

TutTimeCvtCreate — create a time converter

Synopsis

```
T_TIME_CVT TutTimeCvtCreate(name, num_to_str_func, str_to_num_func,
                            destroy_func)
T_STR name;
T_TIME_CVT_NUM_TO_STR_FUNC num_to_str_func;
T_TIME_CVT_STR_TO_NUM_FUNC str_to_num_func;
T_TIME_CVT_DESTROY_FUNC destroy_func;
```

Arguments

name — name of new time converter (not case sensitive)

num_to_str_func — function to convert numeric time value to a string

str_to_num_func — function to convert string time value to a number

destroy_func — optional function to be called if time converter is destroyed. This argument may be null.

Return Values

Returns pointer to new time converter, `NULL` if a converter with the same name already exists.

Diagnostics

None

Description

TutTimeCvtCreate creates a new SmartSockets time converter. A time converter consists of a name, a number-to-string function, a string-to-number function, and a destroy function.

Time converters are used by SmartSockets to convert time values that are stored as floating-point numbers into strings that can be more easily interpreted. For example, the current SmartSockets system time, as returned by TutGetCurrentTime, can be a large number such as 695520549.321. This number is almost useless unless it can be converted to a more meaningful string, such as Fri Mar 27 16:09:09.321 1998. Time converters provide this mechanism and also allow you to customize the printed format of the time value. Time converters are used throughout SmartSockets. For example, this conversion function is used when you are asked to enter a time, such as in the run until *time* command.

The number-to-string function *num_to_str_func* must be written to take two arguments: the numeric time value and an address where it places the string value. The *num_to_str_func* is responsible for converting the time value from a number to a string and storing a pointer to the string in *time_str_return*.

The *num_to_str_func* is also responsible for managing the storage for the string. A *num_to_str_func* should allocate enough string storage the first time it is called and then reuse that storage in subsequent calls. The *num_to_str_func* should not allocate a new string every time it is called. Alternatively, the *num_to_str_func* could declare a static string and choose to use that static storage every time.

The *num_to_str_func* returns TRUE if it successfully converted the number to a string, which is almost always the case, or FALSE if it could not do so. Some time converters may not treat all numeric values as valid time values.

The string to number function *str_to_num_func* is just the inverse of the number to string function. It must be written to take two arguments: the time string and an address where it places the numeric time value. Unlike the *num_to_str_func*, the *str_to_num_func* does not have to explicitly manage any storage. The *str_to_num_func* may choose to accept many different time string formats. The *str_to_num_func* returns TRUE if it successfully converted the string to a number, or FALSE if it could not do so. For example, a *str_to_num_func* expecting a string of the format HH:MM:SS is probably not going to understand the string `yesterday`. A *str_to_num_func* that understands DAY MMM DD HH:MM:SS.FF YYYY may also choose to understand MMM DDD HH:MM:SS.FF for ease of use. It is entirely up to the writer of the time converter to decide what string formats to accept.

Once a time converter has been created, it can be called with TutTimeCvtNumToStr to convert numeric time values to strings, and with TutTimeCvtStrToNum to convert string time values to numbers.

Write the optional destroy function *destroy_func* to do any cleanup necessary when the time converter is destroyed, such as freeing the string storage that was allocated. Time converters are normally not destroyed, but it is possible to destroy them if it is desirable to remove a time converter, making it unavailable for use. If a time converter is destroyed with TutTimeCvtDestroy, the *destroy_func* is called with no arguments. If *destroy_func* is not needed, specify it as NULL in the call to TutTimeCvtCreate.

SmartSockets has these standard converters:

- `full` — convert numeric time to string DAY MMM DD HH:MM:SS.FFF YYYY (for example, Thurs Mar 27 16:09:09.321 1998) and back. The number-to-string function treats the numeric time as the number of seconds occurring since January 1, 1970. The string-to-number function accepts the these formats:
  - MONTH DD HH:MM:SS.FF YYYY, for example, March 10 10:01:51.32 1998
  - MMM DD HH:MM:SS.FF YYYY, for example, Mar 10 10:01:51.32 1998
  - MONTH DD HH:MM:SS.FF YY, for example, March 10 10:01:51.32 98
  - MMM DD HH:MM:SS.FF YY, for example, Mar 10 10:01:51.32 98
  - MONTH DD HH:MM:SS.FF, for example, March 10 10:01:51.32
  - MMM DD HH:MM:SS.FF, for example, Mar 10 10:01:51.32
  - HH:MM:SS.FF, for example, 10:01:51.32

  In all of the above formats, the .FF fractional-second part can be any number of digits or can be omitted.
- `hms` — convert numeric time to string HH:MM:SS, such as 16:09:09. The number to string function treats the numeric time as the number of seconds occurring since January 1, 1970. The string to number function accepts the format HH:MM:SS.FF; for example, 10:01:51.32. The .FF fractional-second part can be any number of digits or can be completely omitted.
- `numeric` — leave time as a floating-point number, such as 695520549.321, printed with the format in the option Real_Number_Format. The string to number function accepts any valid ANSI C real number.

There is a close relationship between time converters and the SmartSockets option Time_Format. The convenience function TutTimeNumToStr uses the value of the Time_Format option to determine which time converter to use, and then calls TutTimeCvtNumToStr using the appropriate converter. Therefore it is normally unnecessary to call TutTimeCvtNumToStr. By using TutTimeNumToStr instead, you can customize the printing of time by simply setting the Time_Format option.

The same relationship is true for TutTimeCvtStrToNum and TutTimeStrToNum. TutTimeStrToNum is a convenience function that should almost always be used instead of TutTimeCvtStrToNum.

Caution SmartSockets time converter names are not case sensitive (`FULL`, `full`, and `Full` all refer to the same time converter).

See Also TutTimeCvtNumToStr, TutTimeCvtDestroy, TutTimeCvtLookup, TutTimeStrToNum, TutTimeNumToStr

Examples This example is a time converter named `percentf` that is similar to `numeric`, but always uses the `%f` `printf` conversion format (`numeric` uses the format specified in the option Real_Number_Format):

```
/* ============================================================== */
/*..utTimeCvtPercentfNumToStr -- convert time to plain number */
static T_BOOL T_ENTRY utTimeCvtPercentfNumToStr(time_val,
time_str_return)
T_REAL8 time_val;          /* time to be converted */
T_STR *time_str_return; /* address of string */
{
  static T_CHAR time_str[80];

  sprintf(time_str, "%f", time_val);
  *time_str_return = time_str;
  return TRUE;             /* number successfully converted to string */
} /* utTimeCvtPercentfNumToStr */

/* ============================================================== */
/*..utTimeCvtPercentfStrToNum -- convert string to numeric time */
static T_BOOL T_ENTRY utTimeCvtPercentfStrToNum(time_str,
time_val_return)
T_STR time_str;
T_REAL8 *time_val_return;
{
  if (sscanf(time_str, "%lf", time_val_return) == 1) {
    return TRUE; /* successfully converted to number */
  }
  else {
    return FALSE;   /* conversion failed */
  }
} /* utTimeCvtPercentfStrToNum */
```

Because this converter uses static storage for its string, it does not need a *destroy_func*.

This code registers the time converter:

```
if (T_NULL == TutTimeCvtCreate("percentf",
utTimeCvtPercentfNumToStr,
utTimeCvtPercentfStrToNum, T_NULL_FUNC)) {
  /* error */
}
```

TutTimeCvtDestroy

Name TutTimeCvtDestroy — destroy a time converter

Synopsis

```
T_BOOL TutTimeCvtDestroy(time_cvtr)
T_TIME_CVT time_cvtr;
```

Arguments *time_cvtr* — time converter to destroy

Return Values TRUE if the converter was successfully destroyed, FALSE if the converter is currently in use, such as the option Time_Format points to the converter.

Diagnostics None

Description TutTimeCvtDestroy destroys a time converter. If the time converter has a destroy function, the function is called.

Caution None

See Also TutTimeCvtCreate, TutTimeCvtLookup

Examples This example tries to destroy the time converter hms:

```
void destroy_hms()
{
  T_TIME_CVT hms = TutTimeCvtLookup("hms");
  if (hms != NULL) {
    if (!TutTimeCvtDestroy(hms)) {
      TutOut("time converter hms is in use.\n");
    }
  }
} /* destroy_hms */
```

TutTimeCvtLookup

Name TutTimeCvtLookup — look up a time converter by name

Synopsis

```
T_TIME_CVT TutTimeCvtLookup(name)
T_STR name;
```

Arguments *name* — name of time converter (not case sensitive)

Return Values Pointer to the specified time converter, `NULL` otherwise.

Diagnostics None

Description TutTimeCvtLookup returns a pointer to a time converter.

Caution SmartSockets time converter names are not case sensitive (`FULL`, `full`, and `Full` can all be used to look up the same time converter).

See Also TutTimeCvtCreate, TutTimeCvtDestroy

Examples This example looks up the time converter `hms`, gets the current wall clock time, and then converts the time to a string using the `hms` converter:

```
T_TIME_CVT time_cvtr;
T_REAL8 wall_time;

wall_time = TutGetWallTime();
TutOut("Raw wall clock time is %f.\n", wall_time);
time_cvtr = TutTimeCvtLookup("hms");
if (time_cvtr != NULL) {
  T_STR *time_str;
  if (TutTimeCvtNumToStr(time_cvtr, wall_time, time_str)) {
    TutOut("Formatted wall clock time is %s\n", *time_str);
  }
}
```

TutTimeCvtNumToStr

Name

TutTimeCvtNumToStr — convert a numeric time value to a string

Synopsis

```
T_BOOL TutTimeCvtNumToStr(time_cvtr, time_num, time_str_return)
T_TIME_CVT time_cvtr;
T_REAL8 time_num;
T_STR *time_str_return;
```

Arguments

time_cvtr — time converter to use

time_num — numeric time value to be converted to a string

time_str_return — pointer to store converted time string into

Return Values

`TRUE` if *time_num* successfully converted, `FALSE` otherwise.

Diagnostics

None

Description

TutTimeCvtNumToStr calls the *num_to_str* function of a time converter to convert a time number to a string. It is normally never necessary to use TutTimeCvtNumToStr directly; use the convenience function TutTimeNumToStr instead.

Caution

The string pointed to by *time_str_return* is managed by the time converter and should not be modified or deallocated.

See Also

TutTimeCvtLookup, TutTimeCvtStrToNum, TutTimeNumToStr

Examples

This example looks up the time converter `hms`, gets the current wall clock time, and then converts the time to a string using the `hms` converter:

```
T_TIME_CVT time_cvtr;
T_REAL8 wall_time;
wall_time = TutGetWallTime();
TutOut("Raw wall clock time is %f.\n", wall_time);
time_cvtr = TutTimeCvtLookup("hms");
if (time_cvtr != NULL) {
  T_STR time_str;
  if (TutTimeCvtNumToStr(time_cvtr, wall_time, &time_str)) {
    TutOut("Formatted wall clock time is %s\n", time_str);
  }
}
```

TutTimeCvtStrToNum

Name TutTimeCvtStrToNum — convert a string time value to a number

Synopsis

```
T_BOOL TutTimeCvtStrToNum(time_cvtr, time_str, time_num_return)
T_TIME_CVT time_cvtr;
T_STR time_str;
T_REAL8 *time_num_return;
```

Arguments

time_cvtr — time converter to use

time_str — string time value to be converted to a number

time_num_return — pointer to store converted time number into

Return Values `TRUE` if *time_str* successfully converted, `FALSE` otherwise.

Diagnostics None

Description TutTimeCvtStrToNum calls the convert function of a time converter to convert a time string to a number. It is normally never necessary to use TutTimeCvtStrToNum directly; use the convenience function TutTimeStrToNum instead.

Unlike TutTimeCvtNumToStr, which rarely fails (returns `FALSE`), it is more likely that TutTimeCvtStrToNum can fail, since it has the more difficult job of converting a time string to a number. If *time_str* is in a format that *time_cvtr* is not prepared to handle (such as the string `"last week"`), then *time_cvtr*'s string to number function fails and returns `FALSE`.

Caution If TutTimeCvtStrToNum returns `FALSE`, it does not put anything in *time_num_return*.

See Also TutTimeCvtLookup, TutTimeCvtNumToStr, TutTimeStrToNum

Examples This example looks up the time converter `hms` and uses it to convert the string `13:56:58` to a time number:

```
T_TIME_CVT time_cvtr;
T_STR time_str = "13:56:58";
T_REAL8 time_num;

time_cvtr = TutTimeCvtLookup("hms");
if (time_cvtr != NULL) {
  if (TutTimeCvtStrToNum(time_cvtr, time_str, &time_num)) {
    TutOut("Time: string %s, number %f\n", time_str, time_num);
  }
}
```

TutTimeCvtTraverse

Name

TutTimeCvtTraverse — traverse all time converters

Synopsis

```
T_PTR TutTimeCvtTraverse(traverse_func, traverse_arg)
T_TIME_CVT_TRAV_FUNC traverse_func;
T_PTR traverse_arg;
```

Arguments

traverse_func — function to call once for each converter

traverse_arg — argument to pass to *traverse_arg*

Return Values

The first non-null return value from *traverse_func*, or NULL if *traverse_func* always returns NULL.

Diagnostics

None

Description

TutTimeCvtTraverse traverses all time converters and calls *traverse_func* once for each converter. The *traverse_func* function is called with three arguments: the converter *name*, the *converter*, and the *traverse_arg* that TutTimeCvtTraverse was called with. When *traverse_func* returns a non-null value, then TutTimeCvtTraverse returns with that value.

There are two common uses for TutTimeCvtTraverse: to perform an operation on each converter and to search for a particular converter. In the first case, write *traverse_func* to always return NULL. In the second case, *traverse_func* should return NULL until it finds the desired converter and then return some kind of non-null value (such as the converter).

Caution

Be careful to always return a value from *traverse_func*.

The order that the converters traverse in is not controllable.

See Also

TutTimeCvtLookup

Examples

This example is a simple traversal function that prints the name of each time converter.

```
T_PTR print_name(name, converter, dummy_arg)
T_STR name;
T_TIME_CVT converter;
T_PTR dummy_arg; /* not used */
{
  TutOut("Converter name is %s.\n", name);
  return NULL;    /* continue traversal */
} /* print_name */
```

To call this function, this code could be used:

```
(void)TutTimeCvtTraverse(print_name, NULL); /* no real traverse_arg */
```

This example is a traversal function that counts the number of converters:

```
T_PTR count_converters(name, converter, counter_ptr)
T_STR name;
T_TIME_CVT converter;
T_PTR counter_ptr;
{
  T_INT4 *cnt_ptr = (T_INT4 *)counter_ptr;
  (*counter_ptr)++;    /* increment our counter */
   return NULL;        /* continue traversal */
} /* count_converters */
```

To call this function, this code could be used:

```
T_INT4 counter = 0;
(void)TutTimeCvtTraverse(count_converters, (T_PTR)&counter);
TutOut("There are %d time converters.\n", counter);
```

TutTimeNumToStr

Name

TutTimeNumToStr — convert a time number to string using the Time_Format option

Synopsis

```
T_STR TutTimeNumToStr(time_num)
T_REAL8 time_num;
```

Arguments

time_num — numeric time value to be converted to a string

Return Values

String representation of *time_num*.

Diagnostics

None

Description

TutTimeNumToStr is a convenience function that converts a numeric time value to its string representation. TutTimeNumToStr uses the value of the option Time_Format to determine which time converter to use and then calls TutTimeCvtNumToStr with the appropriate converter. If TutTimeCvtNumToStr returns `FALSE`, then TutTimeNumToStr returns the string `"UNKNOWN"`.

Caution

The string returned by TutTimeNumToStr is managed by the time converter and should not be modified or deallocated. If a more permanent copy of the string is needed, use T_STRDUP or TutStrDup to make one.

See Also

TutTimeCvtNumToStr, TutTimeStrToNum

Examples

This example prints the string representation of the current SmartSockets data time and the current wall clock computer time:

```
TutOut("SmartSockets time is %s.\n",
TutTimeNumToStr(TutGetCurrentTime()));
TutOut("Wall clock time is %s\n.",
TutTimeNumToStr(TutGetWallTime()));
```

Because the same string storage is reused by time converters, this does not work:

```
TutOut("SmartSockets time is %s.\nWall clock time is %s\n",
       TutTimeNumToStr(TutGetCurrentTime()),
       TutTimeNumToStr(TutGetWallTime()));
```

TutTimeStrToNum

Name

TutTimeStrToNum — convert a string time to a number using the Time_Format option

Synopsis

```
T_BOOL TutTimeStrToNum(time_str, time_num_return)
T_STR time_str;
T_REAL8 *time_num_return;
```

Arguments

time_str — string time value to be converted to a number

time_num_return — pointer to store converted time number

Return Values

TRUE if *time_str* successfully converted, FALSE otherwise.

Diagnostics

None

Description

TutTimeStrToNum is a convenience function that converts a string time value to its numeric representation. TutTimeStrToNum uses the value of the option Time_Format to determine which time converter to use and then calls TutTimeCvtStrToNum with the appropriate converter.

Caution

If TutTimeStrToNum returns FALSE, it does not put anything in *time_num_return*.

See Also

TutTimeCvtStrToNum, TutTimeNumToStr

Examples

This example gets a a time string and converts it to a time number:

```
T_STRING time_str;
T_REAL8 time_num;
if (gets(time_str) && TutTimeStrToNum(time_str, &time_num)) {
  TutOut("Time: string %s, number %f\n", time_str, time_num);
}
```

TutTsdGetValue

Name

TutTsdGetValue — get the value of a thread-specific data key

Synopsis

```
T_BOOL TutTsdGetValue(key, value_return)
T_TSD_KEY key;
T_PTR *value_return;
```

Arguments

key — a key created by TutTsdKeyCreate

value_return — location of a T_PTR into which the associated value is stored

Return Values

TRUE if *key* exists and is set to a non-NULL value, FALSE otherwise.

Diagnostics

If TutTsdGetValue fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *value_return* was NULL
- T_ERR_OS — an operating system error has occurred. Call the function TutErrNumGetOs to obtain the related OS error value.
- T_ERR_VAL_INVALID — *key* was not a valid TSD key

Description

TutTsdGetValue gets the value associated with a thread-specific data key.

Caution

A FALSE result is not necessarily an error — it may just be that the value of the TSD key requested is NULL. This is intended as a convenience for initializing TSD values.

See Also

TutTsdKeyCreate, TutTsdSetValue

Examples

This example retrieves a two-element integer array associated with a TSD key. If the key's value has not been initialized, it is set.

```
T_TSD_KEY my_array_key; /* created previously */
T_INT4 *my_array;
.
.
.
if (!TutTsdGetValue(my_array_key, &my_array)) {
  my_array = (T_INT4 *)TutMalloc(sizeof(T_INT4) * 2);

if (!TutTsdSetValue(my_array_key, my_array)) {
    /* error */
  }
}
```

TutTsdKeyCreate

Name

TutTsdKeyCreate — create a thread-specific data key

Synopsis

```
T_BOOL TutTsdKeyCreate(key_return, destructor_func)
T_TSD_KEY *key_return;
T_TSD_KEY_DESTRUCTOR_FUNC destructor_func;
```

Arguments

key_return — location of a T_TSD_KEY variable to initialize

destructor_func — cleanup function to call at thread exit (may be NULL)

Return Values

TRUE if a new TSD key was successfully created, FALSE otherwise.

Diagnostics

If TutTsdKeyCreate fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_NULL_PTR — *key_return* was NULL
- T_ERR_OS — an operating system error has occurred. Call the function TutErrNumGetOs to obtain the related OS error value.

Description

TutTsdKeyCreate creates a new thread-specific data key and initializes *key_return* with its value. Thread-specific data keys provide a thread-safe substitute for global and file-scope variables. Rather than directly storing a value that is shared by several functions in a global variable, a TSD key can be created. The value may then be placed in memory allocated from the heap and referenced by a unique T_PTR value for each thread. Each function that references the value must then use the TSD key to locate the shared value on the heap with the T_PTR value for the currently executing thread.

The *destructor_func* allows the strategy outlined above to be used without fear of introducing memory leaks as threads are created and exited. Whenever a utility library thread exits, it iterates through all of its TSD values and calls any cleanup functions specified when the keys for those values were created.

Caution

Not all platforms that support threads have native support for TSD. An even smaller subset supports TSD cleanup functions. The utility library implements these features across all platforms (using native support where it is available and internal support code where it is not), but necessarily only for utility library threads. Thus, code that calls utility library TSD functions from native threads may leak memory when the native thread exits and the TSD cleanup functions are not called.

See Also

TutTsdGetValue, TutTsdKeyDestroy, TutTsdSetValue

Examples This example uses TSD to provide a TutOut-like function that uses a different output file handle for each thread:

```
T_TSD_KEY my_output_key;
T_INT4 my_printf(T_STR fmt, ...)
{
  FILE *my_output;
  va_list var_arg_list;
  T_INT4 result;

  if (!TutTsdGetValue(my_output_key, &my_output)) {
    /* default to duplicate stdout */
    my_output = fdopen(dup(fileno(stdout)), "a");
    if (!TutTsdSetValue(my_output_key, my_output)) {
      /* error */
    }
  }
  va_start(var_arg_list, fmt);
  result = vfprintf(my_output, fmt, var_arg_list);
  va_end(var_arg_list);
  return result;
}
.
.
.
void my_output_cleanup(T_PTR my_output)
{
  if (my_output != NULL) {
    fclose((FILE *)my_output);
  }
}

int main()
{
  if (!TutTsdKeyCreate(&my_output_key, my_output_cleanup)) {
    /* error */
  }
  .
  .
  .
  if (!TutTsdKeyDestroy(my_output_key)) {
    /* error */
  }
  return T_EXIT_SUCCESS;
}
```

TutTsdKeyDestroy

Name TutTsdKeyDestroy — destroy a thread-specific data key

Synopsis

```
T_BOOL TutTsdKeyDestroy(key)
T_TSD_KEY key;
```

Arguments *key* — TSD key to destroy

Return Values TRUE if key is successfully destroyed, FALSE otherwise.

Diagnostics If TutTsdKeyDestroy fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_OS — an operating system error has occurred. Call the function TutErrNumGetOs to obtain the related OS error value.
- T_ERR_VAL_INVALID — *key* was not a valid TSD key

Description TutTsdKeyDestroy releases all resources associated with *key*.

Caution Destroying a TSD key does not cause the cleanup function specified when the key was created to be called, and subsequent thread exits do not call the cleanup function either. TutTsdKeyDestroy is normally called only during process shutdown, after all threads other than the main thread have exited.

See Also TutTsdKeyCreate

Examples This example creates and then destroys a TSD key:

```
T_TSD_KEY my_tsd_key;

if (!TutTsdKeyCreate(&my_tsd_key, T_NULL_FUNC)) {
  /* error */
}
.
.
.
if (!TutTsdKeyDestroy(my_tsd_key)) {
  /* error */
}
```

TutTsdSetValue

Name

TutTsdSetValue — set the value of a thread-specific data key

Synopsis

```
T_BOOL TutTsdSetValue(key, value)
T_TSD_KEY key;
T_PTR value;
```

Arguments

key — a key created by TutTsdKeyCreate

value — a T_PTR value to be associated with *key* in the current thread

Return Values

TRUE if the association is successful, FALSE otherwise.

Diagnostics

If TutTsdSetValue fails, it returns FALSE and sets the global SmartSockets error number to one of:

- T_ERR_OS — an operating system error has occurred. Call the function TutErrNumGetOs to obtain the related OS error value.
- T_ERR_VAL_INVALID — *key* was not a valid TSD key

Description

TutTsdSetValue sets the value associated with a thread-specific data key.

Caution

None

See Also

TutTsdGetValue, TutTsdKeyCreate

Examples

This example retrieves a two-element integer array associated with a TSD key. If the key's value has not been initialized, it is set.

```
T_TSD_KEY my_array_key; /* created previously */
T_INT4 *my_array;
.
.
.
if (!TutTsdGetValue(my_array_key, &my_array)) {
  my_array = (T_INT4 *)TutMalloc(sizeof(T_INT4) * 2);
  if (!TutTsdSetValue(my_array_key, my_array)) {
    /* error */
  }
}
```

TutWarning

Name TutWarning — unconditional `printf`-style warning to an output window

Synopsis

```
void TutWarning(format_str, ...)
T_STR format_str;
```

Arguments *format_str* — `printf`-style format string

additional args — `printf`-style arguments

Return Values None

Diagnostics None

Description TutWarning unconditionally prints a warning to the output window of the SmartSockets process that the call is being used with. The output generated is preceded with the word `WARNING`. This function is useful for printing error messages. It supports all the conversion specifications that `printf` does (%f, %g, %d, %s, and so on). See any C reference manual for more information on the conversion specifications.

Caution None

See Also TutOut

Examples This example uses TutWarning for important error messages:

```
/* perform some processing */
if (error_occurs) {
    TutWarning("Could not process data.\n");
    return FALSE;
  }
```

The code issues this output:

```
WARNING: Could not process data.
```

TutWinSetOutputListBox

Name

TutWinSetOutputListBox — set Windows output function

Synopsis

```
void TutWinSetOutputListBox(list_box_hwnd, max_num_lines)
HWND list_box_hwnd;
T_INT4 max_num_lines;
```

Arguments

list_box_hwnd — HWND of a list box for program output

max_num_lines — number of history lines to maintain

Return Values

None

Diagnostics

None

Description

TutWinSetOutputListBox is used to set program output in a Windows environment. The HWND of a list box control is passed to the function. All subsequent program output is displayed in the list box. Without this function, `printf` is used for all program output. `printf` works for Windows console applications but does not display anything in a Windows GUI application. *max_num_lines* is used to specify the number of history lines to maintain in the list box.

Caution

list_box_hwnd must be the HWND of a list box control.

max_num_lines must be greater than 0.

TutWinSetOutputListBox calls TutSetOutputFunc to set the function used by TutOut.

See Also

TutOut, TutSetOutputFunc

Examples This example sets program output in the Windows environment:

```
class CRTclientDlg : public CDialog
{
  public:
  CRTclientDlg(CWnd* pParent = NULL);
  CListBox m_programOutput;
};
BOOL CRTclientApp::InitInstance()
{
 //---------------------------------------------------------
 //create and display the dialog (our main window)
 //---------------------------------------------------------
  m_dlg = new CRTclientDlg;
  m_pMainWnd = m_dlg;
  m_dlg->Create(IDD_RTCLIENT_DIALOG);
 //---------------------------------------------------------
 // redirect TutOut
 // This redirects any TutOut output into the list box of
 // the program dialog box. We pass the HWND of the list box
 // along with the number of lines of history we want to store.
 //---------------------------------------------------------
  TutWinSetOutputListBox(m_dlg->m_programOutput.GetSafeHwnd(),
                         100);
```

TutXmlClone

Name TutXmlClone — make an identical copy of an XML object

Synopsis

```
T_XML TutXmlClone(xml_obj)
T_XML xml_obj;
```

Arguments *xml_obj* — XML object to clone

Return Values Copy of the original XML object if successful; NULL otherwise.

Diagnostics If TipcMsgClone fails, it returns NULL and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *xml_obj* was NULL

Description TutXmlClone makes a copy of an existing XML object. The copied XML object does not share any memory with the original XML object.

Caution Destroy the XML object returned by TutXmlClone when it is no longer needed by calling TutXmlDestroy.

See Also TutXmlCreate, TutXmlDestroy

Examples This example creates a new XML object, then makes a copy of that XML object:

```
T_STR xml_str = "<data><data1><temp=\"hot\"/></data1>"
                "<data2><temp=\"cold\"/></data2></data>";
T_XML xml;
T_XML xml_copy;
xml = TutXmlCreate(xml_str);
if (xml == NULL) {
 /* error */
}
xml_copy = TutXmlClone(xml);
if (xml_copy == NULL) {
 /* error */
}
```

TutXmlCreate

Name

TutXmlCreate — create a new XML object

Synopsis

```
T_XML T_ENTRY T_EXPORT TutXmlCreate(xml_str)
T_STR xml_str;
```

Arguments

xml_str — an XML string

Return Values

New XML object if successful, NULL otherwise.

Diagnostics

If TutXmlCreate fails, it returns NULL and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *xml_str* was NULL

Description

TutXmlCreate creates a new XML object that represents the XML string *xml_str*. The *xml_str* parameter is copied so it does not need to remain allocated by the caller.

Caution

The returned XML object must be destroyed when no longer needed by calling TutXmlDestroy.

See Also

TutXmlCreateStatic, TutXmlDestroy

Examples

This example creates a new XML object and initializes it to a valid XML string:

```
T_STR xml_str = "<data><data1><temp=\"hot\"/></data1>"
                "<data2><temp=\"cold\"/></data2></data>";
T_XML xml;

xml = TutXmlCreate(xml_str);
if (xml == NULL) {
  /* error */
}
```

TutXmlCreateStatic

Name TutXmlCreateStatic — create an XML object from a static buffer

Synopsis

```
T_XML TutXmlCreateStatic(xml_str)
T_XML xml_str;
```

Arguments *xml_str* — an XML string that must remain valid for the life of the XML object.

Return Values New XML object if successful, NULL otherwise.

Diagnostics If TutXmlCreateStatic fails, it returns NULL and sets the global SmartSockets error number to this:

- T_ERR_NULL_PTR — *xml_str* was NULL

Description TutXmlCreateStatic creates a new XML object from a static XML string. This function differs from TutXmlCreate because it does not copy the *xml_str*. The *xml_str* must remain valid (allocated and accessible) for the life of the newly created XML object.

Caution The returned XML object must be destroyed when no longer needed by calling TutXmlDestroy.

See Also TutXmlCreate, TutXmlDestroy

Examples This example creates a new XML object and initializes it to a valid XML string. The value of *xml_str* must remain valid for the duration of the XML object that is returned by TutXmlCreateStatic.

```
T_STR xml_str = "<data><data1><temp=\"hot\"/></data1>"
                "<data2><temp=\"cold\"/></data2></data>";
T_XML xml;
xml = TutXmlCreateStatic(xml_str);
if (xml == NULL) {
 /* error */
}
/* xml_str MUST remain valid until TutXmlDestroy is called */
```

TutXmlDestroy

Name

TutXmlDestroy — destroy an XML object

Synopsis

```
T_BOOL TutXmlDestroy(xml_obj)
T_XML xml_obj;
```

Arguments

xml_obj — XML object to destroy

Return Values

TRUE if the XML object was successfully destroyed, FALSE otherwise.

Diagnostics

If TutXmlDestroy fails, it returns NULL and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *xml_obj* was NULL

Description

TutXmlDestroy destroys the XML object. If the object is not static (that is, it was created with TutXmlCreate), the XML object and all the data contained by that object are destroyed. If the object is static (created with TutXmlCreateStatic), only the XML object itself is destroyed, not the data.

Caution

None

See Also

TutXmlCreate, TutXmlCreateStatic

Examples

This example creates a new XML object and initializes it to a valid XML string, then destroys the newly created XML object:

```
T_STR xml_str = "<data><data1><temp=\"hot\"/></data1>"
                "<data2><temp=\"cold\"/></data2></data>";
T_XML xml;

xml = TutXmlCreate(xml_str);
if (xml == NULL) {
 /* error */
}
if (!TutXmlDestroy(xml)) {
 /* error */
}
```

TutXmlGetStr

Name

TutXmlGetStr — return the XML string associated with the XML object

Synopsis

```
T_BOOL TutXmlGetStr(xml_obj, str_return)
T_XML xml_obj;
T_STR *str_return;
```

Arguments

xml_obj — XML object from which to get the string

str_return — storage for the string value from the XML object

Return Values

`TRUE` if successful, `FALSE` otherwise.

Diagnostics

If TutXmlGetStr fails, it returns `FALSE` and sets the global SmartSockets error number to:

- T_ERR_NULL_PTR — *xml_obj* or *str_return* was `NULL`

Description

TutXmlGetStr retrieves the string associated with the XML object.

Caution

If TutXmlGetStr returns `FALSE`, it does not store a value in *str_return*.

The value stored in *str_return* points directly into an internal data structure and should not be modified.

See Also

TutXmlSetStr

Examples

This example creates a new XML object and initializes it to a valid XML string, then gets that XML string:

```
T_STR xml_str = "<data><data1><temp=\"hot\"/></data1>"
                "<data2><temp=\"cold\"/></data2></data>";
T_XML xml;
T_STR tmp_str;
xml = TutXmlCreate(xml_str);
if (xml == NULL) {
 /* error */
}
if (!TutXmlGetStr(xml, &tmp_str)) {
 /* error */
}
```

TutXmlSetStr

Name

TutXmlSetStr — set the XML string

Synopsis

```
T_BOOL TutXmlSetStr(xml_obj, str)
T_XML xml_obj;
T_STR str;
```

Arguments

xml_obj — XML object into which to set the XML string

str — XML string to set

Return Values

TRUE if successful, FALSE otherwise.

Diagnostics

If TutXmlSetStr fails, it returns FALSE and sets the global SmartSockets error number to one of these:

- T_ERR_NULL_PTR — either *xml_obj* or *str* was NULL
- T_ERR_READ_ONLY — the XML object was created by TutXmlCreateStatic

Description

TutXmlSetStr sets the string into the XML object. This function makes a copy of the string data. Any existing XML string is destroyed (freed).

Caution

None

See Also

TutXmlGetStr

Examples

This example creates a new XML object and initializes it to a valid XML string. It then sets the XML string in the new XML object to a new valid XML string.

```
T_STR xml_str = "<data><data1><temp=\"hot\"/></data1>"
                "<data2><temp=\"cold\"/></data2></data>";
T_STR xml_str2 = "<data><data1><temp=\"warm\"/></data1></data>";
T_XML xml;
xml = TutXmlCreate(xml_str);
if (xml == NULL) {
  /* error */
}
if (!TutXmlSetStr(xml, xml_str2)) {
  /* error */
}
```

Chapter 11 Shell Scripts

This chapter describes the SmartSockets shell scripts. All of the SmartSockets processes can be started by shell scripts. There are also shell scripts to link and create your own custom executables.

Topics

- *Shell Script Summary, page 320*
- *Shell Script Descriptions, page 321*

Shell Script Summary

These lists present the filename and a brief description of the SmartSockets shell scripts available on the supported platform. A reference page with detailed information on each shell script appears later in the chapter, in alphabetical order.

This shell script is available on OpenVMS only:

- `rtinit.com` — initialize OpenVMS DCL to run SmartSockets

These shell scripts are available on UNIX only:

- `rtinit.csh` — initialize UNIX C shell (csh) or extended C shell (tcsh) to run SmartSockets
- `rtinit.sh` — initialize UNIX Bourne shell (sh), Korn Shell (ksh), or GNU Bourne Again Shell (bash) to run SmartSockets

These shell scripts are available on OpenVMS and UNIX:

- `rtlink` — link user-defined executable with SmartSockets libraries
- `rtmove` — reconfigure SmartSockets after moving it

These shell scripts are available on all platforms, including UNIX, OpenVMS and Windows:

- `rtlic` — display SmartSockets license information, including your customer ID
- `rtmon` — start RTmon
- `rtmongdi` — start RTmon graphical development interface (deprecated)
- `rtserver` — start RTserver

This shell script is available on Windows, and on UNIX platforms that have both 32-bit and 64-bit libraries:

- `rtserver64` — start a 64-bit RTserver

Shell Script Descriptions

The reference page for each SmartSockets shell script, presented in alphabetical order, consists of these parts:

- **Name** — the name of the shell script
- **Synopsis** — the shell script syntax and arguments
- **Arguments** — valid arguments
- **Return Values** — valid return values
- **Description** — what the shell script does
- **Caution** — possible side effects or things to watch out for
- **See Also** — related scripts
- **Examples** — at least one example of how the shell script can be used

rtinit.com

Name `rtinit.com` — initialize OpenVMS DCL to run SmartSockets

Synopsis `@`*INSTALL_DEV*`:[`*INSTALL_DIR*`.SS68.COM]RTINIT.COM`

Arguments None

Return Values None

Description The `rtinit.com` script sets up DCL users to run SmartSockets. This should be done in the startup script `SYS$LOGIN:LOGIN.COM` for each user account that is using SmartSockets. It is difficult to automate the process of adding `rtinit.com` to `SYS$LOGIN:LOGIN.COM`, as `SYS$LOGIN:LOGIN.COM` is often very complicated. Add this line to `SYS$LOGIN:LOGIN.COM`:

```
$ @INSTALL_DEV:[INSTALL_DIR.SS68.COM]RTINIT.COM
```

Add the line towards the end of a user's `SYS$LOGIN:LOGIN.COM` file, so that the settings in `rtinit.com` are applied last.

Caution None

See Also `rtinit.csh`, `rtinit.sh`

Examples None

rtinit.csh

Name

`rtinit.csh` — initialize UNIX C shell (csh) or extended C shell (tcsh) to run SmartSockets

Synopsis

`source` *install_dir*`/ss68/bin/rtinit.csh`

Arguments

None

Return Values

None

Description

The `rtinit.csh` script sets up C shell (`csh`) and extended C shell (`tcsh`) users to run SmartSockets. This should be done in the startup script `$HOME/.cshrc` for each user account that is using SmartSockets. It is difficult to automate the process of adding `rtinit.csh` to `$HOME/.cshrc`, as `$HOME/.cshrc` is often very complicated. Add this line to `$HOME/.cshrc`:

`source` *install_dir*`/ss68/bin/rtinit.csh`

Add this line toward the end of a user's `$HOME/.cshrc` file, so that the settings in `rtinit.csh` are applied last.

The `rtinit.csh` script sets or modifies these environment variables:

- `RTHOME` is set to the name of the directory where SmartSockets is installed (the *`install_dir`*`/ss68` mentioned above).
- `RTARCH` is set to the SmartSockets architecture; for example, `sun4_solaris`.
- `PATH` has `$RTHOME/bin` added to the end.
- `XUSERFILESEARCHPATH` has `$RTHOME/app-defaults/$RTARCH` and `$RTHOME/app-defaults` added to the end.
- `XKEYSYMDB` is set to `$RTHOME/lib/$RTARCH/XKeysymDB`. This is needed for Motif 1.2 and X11R5.
- `XNLSPATH` is set to `$RTHOME/lib/$RTARCH/nls`. This is needed for Motif 1.2 and X11R5.
- `LD_LIBRARY_PATH` (UNIX platforms other then HP-UX / AIX) has `$RTHOME/lib/$RTARCH` added to the front. On platforms that support both 32-bit and 64-bit, `$RTHOME/lib/$RTARCH/`*`64dir`* is added before `$RTHOME/lib/$RTARCH`, where *64dir* is the directory containing 64-bit libraries.
- `SHLIB_PATH` (HP-UX only) has `$RTHOME/lib/$RTARCH:$RTHOME/lib/$RTARCH/`*`64Dir`* added to the end, where *64Dir* is the directory containing 64-bit libraries.
- `LIBPATH` (AIX only) has `$RTHOME/lib/$RTARCH` added to the front.

Caution Many users have a conditional exit in their `$HOME/.cshrc` that prevents certain commands from being executed if the shell is not interactive, such as when an executable script is being run. The standard C shell test for a non-interactive login is:

```
if (! $?prompt) then
  exit
endif
```

The line that sources the `rtinit.csh` script should appear before this conditional exit; otherwise, the SmartSockets environment will not be properly set up.

See Also `rtinit.com, rtinit.sh`

Examples None

rtinit.sh

Name

`rtinit.sh` — initialize UNIX Bourne shell (sh), Korn Shell (ksh), or GNU Born Again Shell (bash) to run SmartSockets

Synopsis

. *install_dir*/ss68/bin/rtinit.sh

Arguments

None

Return Values

None

Description

The `rtinit.sh` script sets up Bourne shell (sh), Korn shell (ksh), and GNU Born Again shell (bash) to run SmartSockets. This should be done in the startup script `$HOME/.profile` for each user account that is using SmartSockets. It is difficult to automate the process of adding `rtinit.sh` to `$HOME/.profile`, as `$HOME/.profile` is often very complicated. Add this line to `$HOME/.profile`.

. *install_dir*/ss68/bin/rtinit.sh

Add this line toward the end of a user's `$HOME/.profile` file, so that the settings in `rtinit.sh` are applied last.

The `rtinit.sh` script sets or modifies these environment variables:

- `RTHOME` is set to the name of the directory where SmartSockets is installed (the *install_dir*/ss68 mentioned above).
- `RTARCH` is set to the SmartSockets architecture; for example, `sun4_solaris`.
- `PATH` has `$RTHOME/bin` added to the end.
- `XUSERFILESEARCHPATH` has `$RTHOME/app-defaults/$RTARCH` and `$RTHOME/app-defaults` added to the end.
- `XKEYSYMDB` is set to `$RTHOME/lib/$RTARCH/XKeysymDB`. This is needed for Motif 1.2 and X11R5.
- `XNLSPATH` is set to `$RTHOME/lib/$RTARCH/nls`. This is needed for Motif 1.2 and X11R5.
- `LD_LIBRARY_PATH` (UNIX platforms other then HP-UX/AIX) has `$RTHOME/lib/$RTARCH` added to the front. On platforms that support both 32-bit and 64-bit, `$RTHOME/lib/$RTARCH/`*64dir* is added before `$RTHOME/lib/$RTARCH`, where *64dir* is the directory containing 64-bit libraries.
- `SHLIB_PATH` (HP-UX only) has `$RTHOME/lib/$RTARCH:$RTHOME/lib/$RTARCH/`*64Dir* added to the end, where *64Dir* is the directory containing 64-bit libraries.
- `LIBPATH` (AIX only) has `$RTHOME/lib/$RTARCH` added to the front.

Caution None

See Also `rtinit.csh`, `rtinit.com`

rtlic

Name `rtlic` — display the license information found in the `/standard/talarian.lic` file

Synopsis `rtlic`

Arguments None

Return Values None

Description `rtlic` displays the license information found in the SmartSockets license file, which is in `/standard/talarian.lic`. Note that SmartSockets was originally a product of Talarian Corporation, before Talarian was acquired by TIBCO Software Incorporated.

`rtlic` expands each of the serial numbers found in the file and displays the resulting information. The serial numbers are used to license the various SmartSockets processes, such as RTserver and RTgms.

The display created by `rtlic` shows your customer ID, customer name, and provides other information, such as the number of RTservers licensed. This information is useful if you are contacting TIBCO Product Support to report a problem.

Caution None

See Also None

Examples This example displays the standard license information:

```
$ rtlic

product              = SmartSockets
customer check sum   = <11>
server_count         = <20>
client_count         = <200>
customer_id          = <1234>
product_version      = <62>
reference_date       = <Tue Apr 23 13:11:42.000 2002>
customer_name        = <Superior Messaging, Inc.>
eval                 = <false>
capabilities         = <14>
                     = "Advanced Usage"
                     = "Multi-Threaded"
                     = "Group Connections"
```

rtlink

Name

`rtlink` — link a user-defined process with the SmartSockets rtipc, rtipcl, and rtutil libraries.

Synopsis

On UNIX:

```
rtlink [ -echo ] [ -help ] [ -64bit ] [ -verbose ] [ -static_link ]
       [ -check ] [ -optimized ] [ -Olevel ] [ -cxx | -cpp ]
       [ compiler_options ] files
```

On OpenVMS:

```
rtlink  [ -echo ] [ -help ] [ -verbose ] [ -cxx | -cpp ]
        [ link_options ] files
```

Arguments

`-echo` — show what the compile and link look like but do not actually do it

`-help` — print a brief synopsis of the valid command-line arguments and exit

`-64bit` — link with 64 bit client libraries and compile with 64 bit compiler flags on supported platforms

`-verbose` — show the compile and link as they are being performed

`-static_link` — link with SmartSockets static archive libraries instead of shared libraries

`-check` — link with SmartSockets debug libraries

`-optimized` — Link with SmartSockets optimized libraries and compile with compiler optimized flags

`-O`*level* — compile using optimization; link with SmartSockets optimized libraries

`-cxx | -cpp` — also link with one of the SmartSockets C++ class libraries, either `-cxx` for the `cxxipc` library or `-cpp` for the `sscpp` library on supported platforms

compiler_options — any valid compiler option

files — list of source, object, executable, and library files

link_options — any valid options to the OpenVMS linker

Return Values

On UNIX, `rtlink` returns `0` if successful, and a non-zero value otherwise. On OpenVMS, `rtlink` returns an odd value (usually, but not always `1`) if successful, and an even value otherwise.

Description

The `rtlink` script builds an executable image by linking source and object files with the SmartSockets rtipc, rtipcl, and rtutil libraries. On UNIX platforms, `rtlink` compiles any source files and links the resulting object files with any specified object files. When compiling on UNIX, `rtlink` uses the compiler specified in the `CC` environment variable or the `cc` command if the environment variable `CC` is not set. On OpenVMS platforms, `rtlink` performs a link only.

When supported, use of the shared libraries is the default. To disable the use of shared libraries on UNIX specify the `-static_link` option. To disable the use of shared libraries on OpenVMS, define the `RTSHARE` logical to anything but `YES` (note upper case). Use of shared libraries usually results in faster link times and smaller executable image sizes, but adds additional run-time dependencies and may decrease performance if only a few SmartSockets processes are running on a node.

Debug libraries are supplied on all UNIX platforms. These are SmartSockets libraries that have increased internal error checking. In many cases, the use of the debug libraries can provide increased diagnostic output helpful for troubleshooting problems. To link with these libraries, specify the -check argument.

Specifying the `-cxx` option adds the `rtcxxipc` SmartSockets C++ class library to the list of libraries. Specifying the `-cpp` option adds the `rtsscpp` SmartSockets C++ class library to the list of libraries. When compiling on UNIX, `rtlink -cxx | -cpp` uses the compiler specified in the `CC` environment variable, or the vendor's C++ compiler if the environment variable `CC` is not set. For example, on Solaris 2.8 `rtlink` executes `CC`; however, on AIX `rtlink` executes `xlC_r`.

Caution

On OpenVMS, `rtlink` only performs a link; there is no compilation done.

If the `-cxx`, `-cpp`, `-echo`, `-help`, `-static_link`, or `-verbose` arguments are not the first arguments on the command line, an error is generated.

If the `-O` argument is specified, the SmartSockets optimized libraries must be installed.

See Also

None

Examples UNIX examples:

```
/* This compiles the C program "myprog.c" and produces the custom executable "a.out" */
$ rtlink myprog.c
$ echo $?
0

/* This produces the custom executable "myprog.x" */
$ rtlink myprog.c -o myprog.x
$ echo $?
0

/* A bad file name is passed to rtlink */
$ rtlink foo.c
cpp: foo.c: No such file or directory
$ echo $?
2

/* This shows the result of a bad command-line argument */
$ rtlink -junk myprog.c
cc: Warning: Option -junk passed to ld
ld: -j: bad flag
$ echo $?
4

/* This compiles myprog.c and myfunc.c with the default flag set and creates the custom executable
myprog.x */
$ rtlink -check myprog.c myfunc.c -o myprog.x
myprog.c:
myfunc.c:
Linking:
$ echo $?
0
```

OpenVMS examples:

```
/* On OpenVMS the compilation needs to be performed as a separate step */
$ cc myprog
$ rtlink myprog
$ show sym $status
$STATUS == "%X10000001"

/* Arguments must be separated by commas; white spaces do not work */
$ cc myfunc.c
$ rtlink myprog myfunc
%DCL-W-MAXPARM, too many parameters - reenter command with fewer
parameters \MYFUNC\
$ rtlink myprog, myfunc

/* If rtshare defined to "no", it does not use the shared libraries */
$ define rtshare "no"
%DCL-I-SUPERSEDE, previous value of RTSHARE has been superseded
$ rtlink myprog
/* This example shows how command-line arguments can be passed to the linker */
$ rtlink myprog /exe=ss_app.exe
```

rtmon

Name

`rtmon` — start the SmartSockets monitoring process RTmon

Synopsis

`rtmon [ -command` *pathname* `] [ -help ] [ -nofork ] [ -runtime ]`

Arguments

`-command` *pathname* — alternate startup command file to use (default is `rtmon.cm` in current directory).

`-help` — print a brief synopsis of the valid command-line arguments and exit.

`-nofork` — do not automatically fork `rtmongdi` process.

`-runtime` — do not attempt to start the GDI version of RTmon (on platforms that support it); instead just start the command-line version.

Return Values

On UNIX and Windows, `rtmon` returns `0` if successful, and a non-zero value otherwise. On OpenVMS, `rtmon` returns an odd value (usually but not always `1`) if successful, and an even value otherwise.

Description

The `rtmon` script starts the SmartSockets monitoring process RTmon. For a complete description of RTmon, see the *TIBCO SmartSockets User's Guide*.

The default behavior of RTmon is to attempt to start the GUI version of itself. This interface is implemented as a separate process (rtmongdi). When RTmon starts, it forks (starts) the `rtmongdi` process and attempts to connect to it. If RTmon is unable to connect to the GDI process, it will switch to command-line mode instead. If started with `-runtime`, RTmon goes directly to command-line mode and does not start the `mongdi` process.

The GUI version of RTmon (`rtmongdi`) has been deprecated and may be removed in a future release.

When first invoked, RTmon looks for the license startup command file. This file contains settings for license-related options used by RTmon to obtain a license during the startup process. The filename and directory location of the startup command file is:

UNIX:

```
$RTHOME/standard/license.cm
```

OpenVMS:

```
RTHOME:[STANDARD]LICENSE.CM
```

Windows:

```
%RTHOME%\standard\license.cm
```

The behavior of RTmon is determined by the values set for the various RTmon options, which are read from up to three levels of `rtmon.cm` startup command files. For more information about the startup command file, refer to the *TIBCO SmartSockets User's Guide*.

The `-help` argument causes RTmon to print a brief synopsis and exit.

The `-nofork` argument prevents RTmon from automatically forking (starting) the `rtmongdi` script when RTmon is in development mode. This is useful for debugging `rtmon` and `rtmongdi` separately (usually only useful for source code licenses).

The `-runtime` argument starts RTmon in command-line mode and does not attempt to start or use the `rtmongdi` graphical interface.

The `rtmon` script runs the RTmon executable. On UNIX, `rtmon` is a symbolic link to the generic SmartSockets startup script `rtprocess`. On OpenVMS `rtmon` is not a script but a symbol.

The `rtmon` script starts the standard RTmon from the SmartSockets installation directory.

Caution None

See Also `rtmongdi, rtserver`

Examples

```
/* start rtmon with a different startup command file */
$ rtmon -command file.cm
/* start run-time RTmon */
$ rtmon -runtime
/* start development RTmon, but do not start rtmongdi */
$ rtmon -nofork
```

rtmongdi

This script has been deprecated and may be removed in a future release.

Name

`rtmongdi`— start the SmartSockets RTmon GDI process

Synopsis

`rtmongdi [ -help ] [ -pid` *ProcessID* `]`

Arguments

`-help` — print a brief synopsis of the valid command-line arguments and exit

`-pid` *ProcessID* — specify process ID of RTmon process to communicate with

Return Values

On UNIX and Windows, `rtmongdi` returns `0` if successful, and a non-zero value otherwise. On OpenVMS, `rtmongdi` returns an odd value (usually `1`) if successful, and an even value otherwise.

Description

The `rtmongdi` implements the RTmon GDI. It is normally not necessary to run `rtmongdi` by itself, as RTmon normally starts `rtmongdi` as a child process, automatically.

When `rtmongdi` starts, it normally uses the process ID of its parent process (the RTmon process) to create a connection to the RTmon process (the process ID is used to build a local connection name).

The `-help` argument causes `rtmongdi` to print a brief synopsis and exit.

The `-pid` argument is needed when `rtmongdi` is started manually (such as when both `rtmon` and `rtmongdi` are being run under the control of a debugger). The *ProcessId* should be specified as the process ID of the RTmon process. It is also necessary to start the RTmon process first with the `-nofork` argument (so that the RTmon process is ready to accept a connection from the `rtmongdi` process).

See Also

`rtmon`

Examples

To start `rtmon` and `rtmongdi` separately, use this method. In one terminal emulator window, start RTmon with the `-nofork` option.

```
$ rtmon -nofork
```

SmartSockets banner information

```
.
.
.
My process ID is 9356.
Starting development environment. Please wait.
Waiting for up to 20 seconds for GDI process to connect.
```

The RTmon process is waiting for an `rtmongdi` process to connect to it.

The RTmon only waits for a limited amount of time. If the time-out period is reached and no `rtmongdi` process has connected, the RTmon process automatically switches to run-time mode.

In another terminal emulator window, start `rtmongdi` (using the PID printed by the RTmon process).

```
$ rtmongdi -pid 9356
..Using pid <9356>.
....done.
```

When `rtmon` and `rtmongdi` are started separately, the output that normally appears in the scrollable text window appears instead in the window where the RTmon process is started. This is to allow easier debugging of the two processes.

rtmove

Name `rtmove` — reconfigure SmartSockets after moving it

Synopsis `rtmove`

Arguments None

Return Values On UNIX, `rtmove` returns `0` if successful, and a non-zero value otherwise. On OpenVMS, `rtmove` returns an odd value (usually but not always `1`) if successful, and an even value otherwise.

Description It is possible to move SmartSockets from its initial installation location (see the *TIBCO SmartSockets Installation Guide*). After moving SmartSockets, `rtmove` reconfigures it for the new value of `$RTHOME`. This script must be executed from the `ss68` directory.

Caution Physically moving SmartSockets without running `rtmove` causes unpredictable results.

See Also None

Examples This example moves SmartSockets with `rtmove`:

```
$ install/rtmove              /* UNIX example */
$ @[.install]rtmove           /* OpenVMS example */


Welcome to rtmove, which reinitializes SmartSockets once it has
been moved to a new directory.

When this script asks you questions, a default answer
is shown in brackets next to the question. To use the
default answer, press <Return> in response to the question.
Otherwise, type one of the other suggested responses and
then press <Return>.

This script will ask all its questions first, and then
execute the actions you select.

Would you like to continue? [y]: y

This script changes several SmartSockets files to reference this
directory. Therefore it needs to know the correct name of
this directory. If you will be accessing this directory via
NFS on several hosts, make sure the pathname is valid on all
those hosts. Do not use shell metacharacters (such as * or
~) when answering this question.
```

```
Please enter the full pathname that you would like to use to refer
to this directory [/home/user/ss68]:
Modifying SmartSockets text files to reference /home/user/ss68.
Updating libraries.

Done moving SmartSockets Version 6.8 to /home/user/ss68.
$
```

rtserver

Name

`rtserver` — start or stop the RTserver process

Synopsis

```
rtserver [-check] [ -command filename] [ -help ] [ -install ]
         [ -license ] [-no_console] [ -no_daemon ]
         [ -password pword] [ -reconfigure pid ]
         [ -server_names names_list]  [ -stop ]
         [ -stop_all ] [ -stop_clients ] [ -stop_servers ]
         [ -threads n ] [ -trace_file filename ]
         [ -trace_level level ]  [ -uninstall ] [ -username name]
         [ -verbose ] [ -version ]
```

Arguments

`-check` — start the non-optimized version of the RTserver

`-command` *filename* — specify the startup command file for the RTserver

`-help` — print a summary of the arguments for the `rtserver` command to the console

`-install` — install the RTserver as a Windows service. Use `-demandstart` or `-autostart` with this argument to specify the startup mode of the service. This option is supported only on Windows systems.

`-license` — display license information about your RTserver

`-no_console` — do not display a Windows console associated with the detached process

`-no_daemon` — run the RTserver as a foreground process

`-password` *pword* — specify a password to use when connecting to RTservers with Basic Security enabled

`-reconfigure` *pid* — instruct RTserver to re-read its .cm files by sending SIGHUP to the specified RTserver process. (Supported on UNIX platforms only.)

`-server_names` *names_list* — specify a list of server names for the Server_Names option

`-stop` — stop one RTserver

`-stop_all` — stop all RTserver processes and all RTclient processes

`-stop_clients` — stop one RTserver and all of its RTclient processes

`-stop_servers` — stop all RTserver processes

`-threads` *n* — start the RTserver in MP multithread mode

`-trace_file` *filename* — specify the name of the file where the RTserver writes debugging information

`-trace_level` *level* — specify the amount of information the RTserver writes to the debug file

`-uninstall` — remove the RTserver as a Windows service. This option is supported only on Windows systems.

`-username` *name* — specify a username to use when connecting to RTservers with Basic Security enabled

`-verbose` — is the same as using `-trace_level verbose_1`

`-version` — print version and revision levels of the RTserver to the console

Return Values

On UNIX and Windows, `rtserver` returns `0` if successful, and a non-zero value otherwise. On OpenVMS, `rtserver` returns an odd (usually but not always `1`) value if successful, and an even value otherwise.

Description

The `rtserver` script starts an RTserver. For a complete description of RTserver, see the *TIBCO SmartSockets User's Guide*. RTserver by default runs until initialization is complete, then starts a background process to continue. (On OpenVMS and Windows, this is known as a detached process.) The parent process then exits.

The `rtserver` script runs the RTserver executable. On UNIX, `rtserver` is a symbolic link to the generic SmartSockets startup script, `rtprocess`. On OpenVMS, `rtserver` is not a script, but rather a symbol.

When started, RTserver looks for the license startup command file. This file contains settings for license-related options used by RTserver to obtain a license during the startup process.

The arguments for `rtserver` are:

`-check` — starts the non-optimized version of RTserver, which performs additional validations and checking. If you do not specify `-check`, by default the optimized version of RTserver is started. The optimized version is faster because validation and checking is turned off. However, without the validation and checking, it is harder to debug any problems. When developing your applications or testing RTserver in your development or production environments, always start RTserver with the `-check` argument.

`-command` *filename* — causes RTserver to look for a file named *filename* in the current directory (the directory from which RTserver is being run) and use that as a startup command file. Values for options specified in that file override values for the same options specified in the system-level or user-level startup command file.

`-help`	causes RTserver to print a brief summary of the arguments for the `rtserver` script and exit.
`-install -demandstart -autostart`	installs the RTserver process as a Windows service. You can specify `-demandstart` or `-autostart` as the startup mode of the service. This option is supported only on Windows systems.
`-license`	displays license information about your RTserver. This is useful for finding out your license number when you need to contact TIBCO Product Support. Or you can display license information by using the `rtlic` shell script. See page 327 for more information.
`-no_console`	specifies not to display a Windows console associated with the detached process. Because there is no Windows console, to save output from the RTserver in this mode, you must set the Trace_File option or the `-trace_file` argument. This argument is ignored if you specify `-no_daemon`.
`-no_daemon`	specifies not to start RTserver as a background process and instead run in the foreground.
`-node` *node_name*	starts RTserver on a remote node. A remote shell command is used with `rsh` (`remsh` on HP-UX) to start RTserver on the remote node.
`-password` *pword*	provides the password used by RTserver, along with a username, to connect to other RTservers when Basic Security is enabled. Use with the `-username` argument. *pword* size is unlimited. To specify no password, use empty quotation marks (`""`)
`-reconfigure` *pid*	Instructs RTserver to re-read its .cm files by sending SIGHUP to the specified RTserver process. (Supported on UNIX platforms only.)
`-server_names` *names_list*	specifies the value or values to use for the option Server_Names. *names_list* is a list of logical connection names separated by commas. It allows you to override the value or values specified in a startup command file. The values you specify here must use the same syntax as any value set for the Server_Names option.
`-stop`	specifies that a single RTserver should be stopped. The RTclients that are connected to that RTserver continue to run, eventually detect that RTserver has stopped, and try to find or start a new RTserver.

`-stop_all`	specifies that one RTserver, all RTservers connected to that RTserver, and all RTclients connected to all of the above RTservers should all be stopped. The RTclients are stopped by sending them a CONTROL message with a destination subject of `_all` and containing the command `quit force`.
`-stop_clients`	specifies that one RTserver and all RTclients connected to that RTserver should be stopped. The RTclients are stopped by sending them a CONTROL message with a destination subject of `_all` and containing the command `quit force`.
`-stop_servers`	specifies that one RTserver as well as all RTservers connected to that RTserver should be stopped. The RTclients that are connected to those RTservers continue to run, eventually detect that RTserver has stopped, and try to find or start a new RTserver.
`-threads` *n*	specifies whether the RTserver should start in MP multithread mode, and the number of threads to use. If you do not specify this argument, the default is 1 and RTserver starts in normal mode. If you specify a value greater than 1, the process checks to see if this RTserver was licensed for the MP option. If yes, the RTserver is started in multi-thread mode with the number of threads you specified for *n*. If no, you receive a warning message and the RTserver is started in single-thread mode. We do not recommend setting *n* to a large number. See the *TIBCO SmartSockets User's Guide* for more information. The value you specify for `-threads` overrides the value specified for the Server_Num_Threads option in any of the RTserver startup command files.
`-trace_file` *filename*	specifies the name of the file where the RTserver puts debug information. The default, if you do not specify this argument, is the standard output (`stdout`) which is printed to the console. You can change the value of `-trace_file` after RTserver is started by issuing a CONTROL message to override the values specified at startup time.

`-trace_level` *level*

specifies the amount of information the RTserver puts in the debug file (the file you specified in `-trace_file`). You can change the value of `-trace_level` after RTserver is started by issuing a CONTROL message to override the values specified at startup time.

These are the values you can specify for level:

- `never`, no information is put in a debug file
- `error`, only error messages put in debug file
- `warning`, error and warning messages are put in debug file. This is the default setting.
- `info`
- `info_1`, provides thread information for multi-threading
- `info_2`
- `verbose`
- `verbose_1`
- `verbose_2`
- `debug`, provides the maximum amount of information

`-uninstall`

removes RTserver as a Windows service. This option is only supported on Windows systems.

`-username` *name*

provides the username used by RTserver, along with a password, to connect to other RTservers when Basic Security is enabled. Use with the `-password` argument.

name size is restricted to 64 characters.

`-verbose`

is the same as specifying `-trace_level info`. The `-verbose` argument is provided for backwards compatibility. The preferred method for specifying the amount of information for RTserver to provide is to use the `-trace_level` argument.

`-version`

causes RTserver to print version and revision levels.

The `-stop` arguments have precedence over all other command-line arguments. For example, if both `-node` and `-stop` are specified on the command line, only `-stop` is used. The Enable_Stop_Msgs option in the startup command file specifies whether or not RTserver can be stopped with `rtserver -stop`. By

default this option is set to TRUE, allowing you and all users to stop RTserver. When it is disabled, no variation of the `rtserver -stop` command is permitted (including `rtserver -stop_all`). Security-conscious sites should set this option to FALSE to prevent accidental shutdown of an entire SmartSockets system.

You can change the value of the Enable_Stop_Msgs option after RTserver is started by issuing a CONTROL message. The new value overrides the value specified at startup.

Caution To start RTserver on a remote node, use the command `rtserver -node` *remote_node* instead of executing `rtserver` with a remote shell command such as `rsh` *remote_node* `rtserver` (`remsh` *remote_node* `rtserver` on HP-UX), otherwise your local shell hangs until the remote RTserver exits.

To use a space, tab, comma, or semicolon in the `-server_names` name list, double quotes must be included in the name list, as shown in these examples:

UNIX:

```
-server_names '"local:_node:two words"'
```

OpenVMS:

```
-server_names """local:_node:two words"""
```

Windows:

```
-server_names "local:_node:two words"
```

See Also None

Examples To start an RTserver using the default values of the arguments, enter:

```
rtserver
```

On UNIX systems, you should see SmartSockets banner information and then a line saying that RTserver started successfully.

On Windows systems, if RTserver started successfully, the window is automatically minimized. Check your bottom bar for the RTserver process and click on it to expand the window. In the window, you should see a message saying that the RTserver was started successfully.

The next example shows the output from `rtserver` when run with the `-stop` command-line argument:

```
$ rtserver -stop

SmartSockets banner information
.
.
.
Connecting to project <rtworks> on <_node> RTserver.
Using local protocol.
Connecting to project <rtworks> on <_node> RTserver.
Using local protocol.
Message from RTserver: Connection established.
Start subscribing to subject </_workstation1_13597>.
Sending RTserver stop message.
Message from RTserver: RTserver stopping.
```

If RTserver is not running when `rtserver -stop` is executed, this output appears:

```
$ rtserver -stop

SmartSockets banner information
.
.
.
TAL-SS-00088-I Connecting to project <rtworks> on <_node> RTserver
TAL-SS-00089-I Using tcp protocol
TAL-SS-00090-I Could not connect to <_node> RTserver
TAL-SS-00093-I Skipping starting <_node> RTserver
TAL-SS-50001-I RTserver is not currently running.
```

This example shows how to use the `rtserver` command to stop an RTserver on node `mynode` that is using a custom TCP/IP port number:

```
$ rtserver -stop -server_names tcp:mynode:2001
```

rtserver64

Name `rtserver64` — start or stop the 64-bit RTserver process

Synopsis

```
rtserver64 [ -check ] [ -command filename] [ -help ] [ -install ]
           [ -license ] [-no_console] [ -no_daemon ]
           [ -reconfigure pid ] [ -password pword]
           [ -server_names names_list]  [ -stop ]
           [ -stop_all ] [ -stop_clients ] [ -stop_servers ]
           [ -threads n ] [ -trace_file filename ]
           [ -trace_level level ]  [ -uninstall ] [ -username name]
           [ -verbose ] [ -version ]
```

See `rtserver` on page 337 for usage information.

`rtserver64` is only supported on platforms that provide both 32-bit and 64-bit versions of the SmartSockets libraries.

Index

Numerics

-64bit argument 328

A

access functions 7, 11

B

Bourne shell 325

C

C shell setup 323
callback list 16
callbacks 4
  global vs object-specific 5
case sensitivity xv
  on UNIX and Windows xv
-check argument 337
-command argument 331, 337
commands
  parsing 69, 70
compiling and linking 7
condition variable 13
conventions used in manual 8
conventions used in this manual xiii
customer support xvi

D

data structures 11
data time 24, 25
data type
  T_BOOL 3
  T_CHAR 3
  T_INT1 3
  T_INT2 3
  T_INT4 3
  T_INT8 3
  T_PTR 3
  T_REAL16 3
  T_REAL4 3
  T_STR 3
  T_UCHAR 3
  T_UINT1 3
  T_UINT2 3
  T_UINT4 3
DCL
  setup 322
-default argument 328
diagnosing problems
  starting debug RTserver 337
displaying
  license information 327

E

-echo argument 328
error handling 28
error number 28
external processes
  compiling and linking 328

F

file names
  specifying xv
files
  SmartSockets license 327
full time converter 294
functions
  case-sensitivity xv

H

hash tables 19
  traversing 19
-help argument 328, 331, 333, 337
hms time converter 294

I

identifiers
  case sensitivity xv
-install argument 337

L

license
  displaying 327
-license argument 337
lookup functions 6

M

macros
  T_ASSERT 30
  T_CALLOC 32
  T_ENTRY 33
  T_FREE 34
  T_MALLOC 35
  T_REALLOC 36
  T_STRDUP 38
messages
  case sensitivity xv
multithreading
  See also threads 11
mutex 13

N

-no_console argument 337
-no_daemon argument 337
-nofork argument 331
numeric time converter 294

O

-O argument 328
opaque data type 11
-optimized argument 328
option change callbacks 23
  data structure 24
  function 24
option types
  Boolean 23
  Enumerated 23
  Enumerated List 23
  Numeric 23
  Numeric List 23
  String 23
  String List 23

options
  case sensitivity xv
  Real_Number_Format 228, 295
  Time_Format 294, 302, 303
  verify function 23

P

-password argument 337
perror 195
-pid argument 333
pointer array 21
  compare function 21
  user function 21
pointer array change callbacks 21
  data structure 21
  function 21

R

read / write mutex 13
Real_Number_Format option 228, 295
-reconfigure argument 337, 339
rtinit.com script 322
rtinit.csh script 323
rtinit.sh script 325
rtlic script 327
rtlink script 328
RTmon
  rtmon script 331
  rtmongdi script 333
  starting 331
rtmove script 335
RTserver
  debugging 337
  rtserver script 337, 344
  starting 337, 344
rtserver64 command 344
-runtime argument 331

S

-server_names argument 337
Server_Num_Threads option
  overriding value 340
setup 325
shell commands
  specifying xv
shell scripts 319
shell setups
  Bourne shell 325
  C shell 323
  DCL 322
SmartSockets license
  displaying 327
starting processes
  RTmon 331
  RTmon GDI 333
  RTserver 337, 344
-static_link argument 328
-stop argument 337
-stop_all argument 337
-stop_clients argument 337
-stop_servers argument 337
structures
  T_CB_DATA 17
  T_FILE 27
  T_HASH_BUCKET 11, 20
  T_HASH_TABLE 11
  T_OPTION 11
  T_OPTION_CHANGE_CB_DATA 24
  T_PTR_ARY 11
  T_PTR_ARY_CHANGE_CB_DATA 21
  T_TIME 24
  T_TIME_CHANGE_CB_DATA 25
  T_TIME_CVT 11
support
  providing license information 327
support, contacting xvi

T

T_ASSERT macro 30
T_BOOL data type 3
T_CALLOC macro 32
T_CB_DATA structure 17
T_CB_DATA typedef 17
T_CB_FUNC typedef 17
T_CHAR data type 3
T_COND typedef 14
T_ENTRY macro 33
T_FILE structure 27
T_FILE_CLOSE_FUNC typedef 27
T_FILE_OPEN_FUNC typedef 27
T_FILE_READ_FUNC typedef 27
T_FILE_TRAV_FUNC typedef 27
T_FREE macro 34
T_HASH_BUCKET structure 11, 20
T_HASH_BUCKET typedef 20
T_HASH_CALC_FUNC typedef 19
T_HASH_DESTROY_FUNC typedef 20
T_HASH_SORT_CMP_FUNC typedef 20
T_HASH_SORT_SELECT_FUNC typedef 20
T_HASH_TABLE structure 11
T_HASH_TEST_FUNC typedef 19
T_HASH_TRAV_FUNC typedef 19
T_INT_BIG_ENDIAN 26
T_INT_FORMAT_INVALID 26
T_INT_LITTLE_ENDIAN 26
T_INT1 data type 3
T_INT2 data type 3
T_INT4 data type 3
T_INT8 data type 3
T_MALLOC macro 35
T_MUTEX typedef 13
T_OPT_TYPE_BOOLEAN 23
T_OPT_TYPE_ENUM 23
T_OPT_TYPE_LIST_ENUM 23
T_OPT_TYPE_LIST_NUMERIC 23
T_OPT_TYPE_LIST_STRING 23
T_OPT_TYPE_NUMERIC 23
T_OPT_TYPE_STRING 23
T_OPTION structure 11
T_OPTION_CHANGE_CB_DATA structure 24
T_OPTION_CHANGE_CB_DATA typedef 24
T_OPTION_CHANGE_CB_FUNC typedef 23
T_OPTION_VERIFY_FUNC typedef 23
T_PA_CLEARED 22
T_PA_ITEM_INSERTED 22
T_PA_ITEM_REMOVED 22
T_PA_ITEM_REPLACED 22
T_PA_ITEMS_REMOVED 22
T_PA_ITEMS_SWAPPED 22
T_PA_SORTED 22
T_PTR data type 3
T_PTR_ARY structure 11
T_PTR_ARY_CHANGE_CB_DATA structure 21
T_PTR_ARY_CHANGE_CB_DATA typedef 21
T_PTR_ARY_CMP_FUNC typedef 21
T_PTR_ARY_USER_FUNC typedef 21
T_REAL_DEC_D 26
T_REAL_DEC_G 26
T_REAL_FORMAT_INVALID 26
T_REAL_IBM_370 26
T_REAL_IEEE 26
T_REAL16 data type 3
T_REAL4 data type 3
T_REALLOC macro 36
T_RW_MUTEX typedef 14
T_STR data type 3
T_STRDUP macro 38
T_THREAD typedef 12
T_THREAD_FUNC typedef 12
T_TIME structure 24
T_TIME typedef 24
T_TIME_CHANGE_CB_DATA structure 25
T_TIME_CHANGE_CB_DATA typedef 25
T_TIME_CHANGE_CB_FUNC typedef 25
T_TIME_CVT structure 11
T_TIME_CVT_DESTROY_FUNC typedef 18
T_TIME_CVT_NUM_TO_STR_FUNC typedef 18
T_TIME_CVT_STR_TO_NUM_FUNC typedef 18
T_TIME_CVT_TRAV_FUNC typedef 18
T_TSD_KEY typedef 15
T_TSD_KEY_DESTRUCTOR_FUNC typedef 15
T_UCHAR data type 3
T_UINT1 data type 3
T_UINT2 data type 3
T_UINT4 data type 3

technical support xvi
  providing license information 327
threads 11
  condition variable 13
  cross-platform support 12
  fast mutexes 144, 149
  initialization 12
  mutex 13
  read / write mutex 13
  thread-specific data 15
-threads argument 337
thread-specific data 15
time change callbacks 25
time converters 18
  full 294
  hms 294
  numeric 294
Time_Format option 294, 302, 303
TipcInitThreads function 12
-trace_file argument 337
-trace_level argument 338
traversing
  overview 6
TutBufAllocAligned function 40
TutBufAppendBuf function 41
TutBufAppendRange function 42
TutBufCloneRange function 43
TutBufCreate function 44
TutBufCreateStatic function 45
TutBufDeleteRange function 46
TutBufDestroy function 47
TutBufGetSize function 48
TutBufGrow function 49
TutBufNextAligned function 50
TutBufReset function 51
TutBufRewind function 52
TutBufSetSize function 53
TutCalloc function 56
TutCbDestroy function 57
TutCbGetArgument function 58
TutCbGetFunction function 59
TutCbGetPriority function 61
TutCbSetArgument function 62
TutCbSetFunction function 64
TutCbSetPriority function 66
TutCommandParseFile function 68
TutCommandParseStr function 69
TutCommandParseTypedStr function 70
TutCondCreate function 14, 71
TutCondDestroy function 14, 73
TutCondWait function 14, 74
TutCondWakeAll function 14, 76
TutErrNumGet function 28, 80
TutErrNumGetC function 81
TutErrNumGetOs function 82
TutErrNumGetSocket function 83
TutErrNumSet function 28, 84, 85, 86, 87
TutErrNumToStr function 28
TutErrStrGet function 28
TutExit 88
TutFileTraverse function 27, 90
TutFOpen function 27, 91
TutFree function 93
TutGetCurrentTime function 24, 97
TutGetCurrentTimeStruct function 24, 98
TutGetenv function 99
TutGetIntFormat function 26, 100
TutGetNodeName function 101
TutGetOutFlushFunc function 102
TutGetOutputFunc function 103
TutGetRealFormat function 26, 104
TutGetSocketDir function 105
TutGetVersionName function 106
TutGetVersionNumber function 107
TutGetWallTime function 108
TutGetWallTimeStruct function 109
TutHashBucketGetValue function 110
TutHashBucketSetValue function 111
TutHashCreate function 19, 112
TutHashCreateInt function 19, 115
TutHashCreatePtr function 19, 117
TutHashCreateStr function 19, 119
TutHashCreateStri function 19, 120
TutHashDelete function 121
TutHashDestroy function 20, 123
TutHashDump function 125
TutHashGetSize function 127
TutHashInsert function 128
TutHashLookup function 19, 130
TutHashLookupBucket function 131

TutHashReplace function 132
TutHashSort function 20, 135
TutHashTraverse function 19, 137
TutMalloc function 141
TutMutexCreate function 13, 142
TutMutexCreateFast function 144
TutMutexDestroy function 13, 147
TutMutexLock function 13, 148
TutMutexLockFast function 149
TutMutexUnlock function 13, 150
TutOptionChangeCbCreate function 24, 151, 153
TutOptionChangeCbLookup function 24
TutOptionCreate function 155
TutOptionDestroy function 157
TutOptionGetBool function 158
TutOptionGetEnum function 159
TutOptionGetEnumList function 160
TutOptionGetKnown function 162
TutOptionGetName function 163
TutOptionGetNum function 164
TutOptionGetReadOnly function 165
TutOptionGetRequired function 166
TutOptionGetStr function 167
TutOptionGetStrList function 168
TutOptionGetType function 170
TutOptionGetVerifyFunc function 23, 172
TutOptionLegValAdd function 174
TutOptionLookup function 175
TutOptionNamedLookup function 176
TutOptionSetBool function 177
TutOptionSetEnum function 178
TutOptionSetEnumList function 179
TutOptionSetNum function 181
TutOptionSetReadOnly function 182
TutOptionSetRequired function 183
TutOptionSetStr function 184
TutOptionSetStrList function 185
TutOptionSetUnknown function 187
TutOptionSetVerifyFunc function 23, 188
TutOut function 190
TutPerror function 195
TutProcInfo function 196
TutPtrAryAppend function 197
TutPtrAryChangeCbCreate function 22, 198, 200
TutPtrAryChangeCbLookup function 22
TutPtrAryClear function 21, 201
TutPtrAryCreate function 203
TutPtrAryDestroy function 21, 204
TutPtrAryGetItemAtIndex function 206
TutPtrAryGetItemCount function 207
TutPtrAryGetItemIndex function 208
TutPtrAryInsertAfter function 210
TutPtrAryInsertBefore function 212
TutPtrAryInsertItemAtIndex function 214
TutPtrAryItemExists function 216
TutPtrAryRemove function 217
TutPtrAryRemoveAll function 219
TutPtrAryReplaceItemAtIndex function 221
TutPtrArySort function 21, 223
TutPtrArySwapItems function 225
TutPutenv function 226
TutRealloc function 227
TutRealToStr function 228
TutRwMutexCreate function 14, 230
TutRwMutexDestroy function 14, 232
TutRwMutexGetReadQuota function 14, 233
TutRwMutexReadLock function 14, 234
TutRwMutexReadLocked function 14, 235
TutRwMutexSetReadQuota function 14, 236
TutRwMutexUnlock function 14, 237
TutRwMutexUpgradeLock function 14, 238
TutRwMutexWriteLock function 14, 240
TutRwMutexWriteLocked function 14, 242
TutSetCurrentTime function 24, 245
TutSetCurrentTimeStruct function 24, 246
TutSetFileCloseFunc function 27, 247
TutSetFileOpenFunc function 27, 249
TutSetFileReadFunc function 27, 250
TutSetOutFlushFunc function 191, 251
TutSetOutputFunc function 252
TutSleep function 253
TutSocketAccept function 254
TutSocketCheck function 255
TutSocketCreateClientLocal function 257
TutSocketCreateClientTcp function 259
TutSocketCreateServerLocal function 262
TutSocketCreateServerTcp function 265
TutSocketGetBlockMode function 268
TutSocketRecvAll function 270
TutSocketRecvTimeout function 272

TutSocketSetBlockMode function 274
TutStrDup function 275
TutStrLwr function 276
TutStrUpr function 277
TutSystem function 278
TutThreadCreate function 13, 279
TutThreadCreateVa function 13, 281
TutThreadDetach function 13
TutThreadEqual function 13, 283, 285
TutThreadExit function 13, 286
TutThreadSelf function 13, 287
TutThreadWait function 13, 288
TutTimeChangeCbCreate function 290
TutTimeChangeCbLookup function 291
TutTimeCvtCreate function 18, 292
TutTimeCvtDestroy function 296
TutTimeCvtLookup function 297
TutTimeCvtNumToStr function 298
TutTimeCvtStrToNum function 299
TutTimeCvtTraverse function 18, 300
TutTimeNumToStr function 302
TutTimeStrToNum function 303
TutTsdGetValue function 15, 304
TutTsdKeyCreate function 15, 305
TutTsdKeyDestroy function 15, 307
TutTsdSetValue function 15, 308
TutWarning function 309
TutWinSetOutputListBox function 310
Tutxmlclone function 312
Tutxmlclone ulitity 28
TutXmlCreate function 313
TutXmlCreate utility 28
TutXmlCreateStatic function 314
TutXmlCreateStatic utility 28
TutXmlDestroy function 315
TutXmlDestroy utility 28
TutXmlGetStr function 316
TutXmlGetStr utility 28
TutXmlSetStr function 317
TutXmlSetStr utility 28

typedefs
T_CB_DATA 17
T_CB_FUNC 17
T_COND 14
T_FILE_CLOSE_FUNC 27
T_FILE_OPEN_FUNC 27
T_FILE_READ_FUNC 27
T_FILE_TRAV_FUNC 27
T_HASH_BUCKET 20
T_HASH_CALC_FUNC 19
T_HASH_DESTROY_FUNC 20
T_HASH_SORT_CMP_FUNC 20
T_HASH_SORT_SELECT_FUNC 20
T_HASH_TEST_FUNC 19
T_HASH_TRAV_FUNC 19
T_MUTEX 13
T_OPTION_CHANGE_CB_DATA 24
T_OPTION_CHANGE_CB_FUNC 23
T_OPTION_VERIFY_FUNC 23
T_PTR_ARY_CHANGE_CB_DATA 21
T_PTR_ARY_CMP_FUNC 21
T_PTR_ARY_USER_FUNC 21
T_RW_MUTEX 14
T_THREAD 12
T_THREAD_FUNC 12
T_TIME 24
T_TIME_CHANGE_CB_DATA 25
T_TIME_CHANGE_CB_FUNC 25
T_TIME_CVT_DESTROY_FUNC 18
T_TIME_CVT_NUM_TO_STR_FUNC 18
T_TIME_CVT_STR_TO_NUM_FUNC 18
T_TIME_CVT_TRAV_FUNC 18
T_TSD_KEY 15
T_TSD_KEY_DESTRUCTOR_FUNC 15

U

-uninstall argument 338
-version argument 338
user-defined processes
  compiling and linking 328
-username argument 338
utilities
  Tutxmlclone 28
  TutXmlCreate 28
  TutXmlCreateStatic 28
  TutXmlDestroy 28
  TutXmlGetStr 28
  TutXmlSetStr 28

V

-verbose argument 328, 338
-version argument 338