



Runtime Developer's Guide

Version 3.01

Published by

S2 Technologies, Inc.
2037 San Elijo Avenue
Cardiff, CA 92007 USA

The information in this document is subject to change without notice.
Copyright © 2001 – 2008 S2 Technologies, Inc. All rights reserved.

S2 Technologies, the S2 Technologies logo, STRIDE, and the STRIDE logo are trademarks of S2 Technologies, Inc. Microsoft, Windows, and Visual C++ are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are the property of their respective owners.

Contents

Contents	1
About this Guide.....	6
Purpose	6
Document Conventions.....	6
Standard Data Types	6
Standard Defines	7
Hungarian Notation for Variables.....	7
Naming Conventions.....	8
Grammar	9
Terms	10
Related Documents.....	12
1. Using the STRIDE Runtime.....	13
1.1. Overview	13
1.1.1. STRIDE Message ID (SMID).....	14
1.1.2. STRIDE Unique ID (SUID)	16
1.1.3. STRIDE Transactor ID (STID).....	16
1.1.4. STRIDE Trace Point ID (STPID)	16
1.1.5. STRIDE Response ID (SRID)	17
1.1.6. Notification of Traffic ID (NID)	17
1.2. Memory Requirements.....	18
1.2.1. Messaging Memory	19
1.2.2. Tracing Memory.....	21
1.2.3. Transport Settings	22
1.2.4. Memory Management.....	22
1.2.5. Multi-Process Target	23
1.3. Using the API	24
1.3.1. Creating an STID.....	24
1.3.2. Creating a STRIDE Message	24
1.3.3. Registering Messages	26
1.3.4. Overriding Registration.....	26
1.3.5. Subscribing to Messages	26
1.3.6. Reading and Sending Messages	26
1.3.7. Using Pointers	28
1.3.8. Returning Message Memory	29
1.3.9. Pointer Memory Policies.....	30

1.3.10. Trace Points	31
1.3.11. Data Format Conformance	31
1.4. Routing with Access Class Registration	32
1.4.1. Remote Messaging (RM) Overview	32
1.4.2. Access Class Intercept Module	32
1.5. Connecting to the Host	33
1.5.1. Connection Settings	33
2. Remote Messaging (RM).....	34
2.1. Implementing a Remote Messaging Service	35
2.1.1. How It Works	35
2.1.2. Issues to Consider.....	36
2.1.3. Translating between STRIDE and Native Message IDs	37
2.2. Using a Remote Message Stub (RMS).....	37
2.2.1. Remote Message Stub Thread Setup	38
2.2.2. Remote Message Stub Thread Messages	38
2.2.3. How to Handle Responses.....	39
2.2.4. Binding Native Commands with Native Responses	39
2.2.5. Wait Event	40
2.2.6. Subscriptions and Broadcasts.....	40
2.2.7. Pointers.....	40
2.2.8. Message Tracing	41
2.2.9. Remote Message Stub Thread Example	41
2.3. Using a Remote Message Proxy (RMP).....	43
2.3.1. Remote Message Proxy Routing.....	44
2.3.2. Binding Native Commands with Native Responses	44
2.3.3. Receiving STRIDE Responses – Sending Native Responses.....	44
3. Runtime API Services	45
3.1. Setup and Shutdown.....	46
srInit().....	47
srUninit()	48
srCreateSTID()	49
srDeleteSTID().....	51
3.2. Messaging.....	52
srRegister().....	53
srRegisterAccessClass()	55
srRead().....	57
srReadComplete()	60

srSendCmd()	62
srSendRsp()	65
srBroadcast()	69
srSubscribe()	72
srSetAuxData()	75
srGetAuxData()	77
3.3. Pointers	79
srPtrSetup()	80
srPtrSetupChild()	83
srPtrTeardown()	86
srPtrGetHandle()	88
srPtrSize()	90
srPtrCreateCmdInst()	93
srPtrCreateRspInst()	94
srPtrDeleteInst()	95
3.4. Tracing	96
srTracePoint()	97
srTraceStr()	99
srTraceInterface()	101
3.5. Printing	102
srPrintInfo()	103
srPrintError()	104
3.6. Query	105
srQueryAccessClass()	106
srQueryNID()	107
srQueryName()	108
srQuerySMID()	109
srQueryBox()	111
3.7. Access Class (Remote Messaging) Routines	113
3.8. I-block	114
srIBlockOutReady()	114
3.9. Runtime Thread Entry and Exit Points	116
srThread()	117
srThreadInit()	118
srThreadUninit()	119
srThreadProc()	120
3.10. Host Override Routines	121

3.10.1. srHostShutdownIM().....	122
3.11. Connecting	124
srCONNECT_OPEN_T_SMID	125
srCONNECT_CLOSE_T_SMID	127
srCONNECT_STATUS_B_SMID	129
srCONNECT_STATUS_T_SMID	131
3.12. Database Loading Routines.....	133
srHOST_LOAD_DB_O_SMID	134
srHOST_LOAD_DB_STATUS_B_SMID	135
srHOST_LOAD_DB_STATUS_T_SMID	137
3.13. Trace Buffers.....	139
srTRACE_BUFFER_B_SMID.....	140
3.13.1. Trace Filtering.....	142
srTRACE_FILTER_STID_O_SMID	143
3.14. Subscriber Information.....	145
srSUBSCRIBERS_LOCAL_B_SMID	146
srSUBSCRIBERS_REMOTE_B_SMID	148
3.15.	149
3.16. Marshaling Errors.....	150
srERROR_MARSHAL_B_SMID.....	151
4. Runtime Test Services (RTS)	153
C Test Functions	153
C++ Test Classes	153
4.1. Runtime Test Services APIs	155
srTestSuiteAddSuite	155
srTestSuiteSetName	157
srTestSuiteSetDescription	158
srTestSuiteAddTest	160
srTestCaseSetName	162
srTestCaseSetDescription	164
srTestCaseAddComment	166
srTestCaseSetStatus.....	168
4.2.	168
srTestCaseSetStatusEx	169
4.3.	169
srTestSuiteAddAnnotation.....	171
srTestAnnotationAddComment	173

4.3.1. Messages to Support Runtime Test Services	175
4.4. Runtime Base Class (srTest)	186
testSuite	186
testCase	186
4.4.1. Methods of Member Object testSuite	187
4.4.2. Methods of Member Object testCase	196
5. STRIDE Runtime Internals	201
5.1. STRIDE Runtime Thread and Procedure	201
5.2. STRIDE Runtime Modules	201
5.3. STRIDE Runtime Files	203
Appendix A: STRIDE Runtime API (sr.h)	205
Appendix B: STRIDE Runtime Messages (srmsg.h)	213
Appendix C: STRIDE Runtime Configuration (srcfg.h)	216
Appendix D: STRIDE Runtime Test Services (srtest.h)	219




About this Guide

Purpose

The STRIDE Runtime Developer's Guide provides information you need to use the STRIDE Runtime. It also includes the API services, message services, and header files.

Document Conventions

The following symbols indicate specific activities, events or notes for the developer:

...	Time passing or activity
	Take special care to avoid errors
	Notes, remarks or additional information that could affect performance
	Interface through use of messaging

Standard Data Types

The STRIDE Runtime uses the following standard basic data types as shown in Standard Data Types below. These types are provided to help self-document the interfaces. The sizes of these types are based on the Platform Abstraction Layer (PAL) standard types found in pal.h. (See the *STRIDE Platform Abstraction Layer Specification*.)

Figure 1: Standard Data Types

```
typedef palCHAR      srCHAR;
typedef palBYTE      srBYTE;
typedef palSHORT     srSHORT;
typedef palWORD      srWORD;
typedef palLONG      srLONG;
typedef palDWORD     srDWORD;
typedef palBOOL      srBOOL;
```


Standard Defines

The STRIDE Runtime also defines TRUE, FALSE and NULL based on standard definitions.

Figure 2: Standard Defines

```
#define srFALSE      palFALSE
#define srTRUE       palTRUE
#define srNULL       palNULL
```

Hungarian Notation for Variables

The naming convention for variables used by the runtime API follows a modified version of the Hungarian notation. Each variable name begins with one or more lowercase characters identifying the type of the variable. Of special note are the enumeration type and the more general typedef. The “_e” notation indicates that an enum typedef is being used. The “_t” indicates a general typedef. Variables declared as an enumeration or a general typedef will use “e” or “t” in the prefix.

Table 1. Hungarian Notation

Prefix	Meaning	Example	
c	char	palCHAR	cMyChar;
y	unsigned char	palBYTE	yMyByte;
n	short	palSHORT	nMyShort;
w	unsigned short	palWORD	wMyWord;
l	long	palLONG	lMyLong;
dw	unsigned long	palDWORD	dMyDWord;
b	Boolean	palBOOL	bMyBool;
e	enumeration	<Name>_e	eMyEnum;
t	typedef	<Name>_t	tMyTypedef;
p	pointer	palBOOL	*pbMyPtrBool;
sz	zero terminated string	palCHAR	*szMyString;

Naming Conventions

All *public* header files, prototypes, data types, constants, and variables use the *component tag* (i.e., lower case “sr”) as a prefix. The following naming conventions are used:

Item	Convention	Example
Files	<tag><name>.h .c	srCfg.h
Prototypes	<tag><Name>(…)	srMyFunction(..)
Typedefs	<tag><Name>_t	srMyType_t
Constants	<tag><NAME>{_ <NAME>}	srMY_CONSTANT
Enumeration	<tag><Name>_e	srMyEnum_e
Enumerator	<tag><NAME>{_ <NAME>}	srMY_ENUMERATOR

All *private* files also use the component tag as well as the module name. Private prototypes and variables whose scope is global insert an additional underscore (“_”) in front of the *component tag* as a prefix. Static variables defined within a module and local constants and typedefs do not follow any specific convention.

Item	Convention	Example
Files	<tag><module-name>.h	Example: srmod.h
	<tag><module-name>.c	Example: srmod.c
Prototypes	_ <tag><Module- name>_ <Name>(…)	Example: _srMod_Func(..)
Variable	_ <tag><Module-name>_ <Name>	Example: _srMod_Variable
Typedefs	<Module-name>_ <Name>_t	Example: Mod_Type_t
Constants	<Module- name>_ <NAME>{_ <NAME>}	Example: MOD_CONSTANT
Enumeration	<Module-name>_ <Name>_e	Example: Mod_Enum_e
Enumerator	<Module- name>_ <NAME>{_ <NAME>}	Example: MOD_ENUMERATOR

Grammar

Item	Grammar
<name>	<alpha-char> {<alpha-char>}
<alpha-char>	a b c .. z
<NAME>	<ALPHA-CHAR> {<ALPHA-CHAR>}
<ALPHA-CHAR>	A B C .. Z
<Name>	<ALPHA-CHAR> {<string>}
<string>	<Alpha-Char> {<Alpha-Char>}
<Alpha-Char>	<ALPHA-CHAR> <alpha-char>
<Module-name>	<ALPHA-CHAR> {<string>}
<module-name>	<alpha-char> {<alpha-char>}
<tag>	<alpha-char> {<alpha-char>}

Terms

broadcast	To send a response to one or more subscribers. A unidirectional message in which one or more Owners sends a response to one or more Users. The Owner independently sends the message and the runtime is responsible for routing the message based on a subscriber list.
command	A message sent from a User to an Owner
component tag	Letters in the name of an API or source file that identifies a group of related functionality (example: “err” is used for the component tag for the files and APIs related to error routines in the source code)
Pointer Entry	A STRIDE Runtime data structure used for holding a single pointer's information.
EPE	Pointer Entry
I-block	STRIDE Communication Model (SCM) term for a packet of data transferred between platforms
mailbox	Logical ID for a runtime message queue
MCB	Message Control Block
message	A communication mechanism between two threads.
message type	One-way, two-way, or broadcast.
MID	Message Identifier
MQE	Message Queue Entry
MSE	Message Subscribe Entry
NID	Notification Identifier – a unique 32-bit value used by the pal notify routine to notify an STID of a pending message; used by the native platform when performing synchronization services.
Owner	A thread that reads a command and or sends a response
PAL	Platform Abstraction Layer

payload	The data portion of a message or TracePoint
pool memory	Memory allocated from a common pool used by application threads
private memory	Non-pool memory that is owned by a sending application thread
process	Implies a separate address space which typically does not apply to a task or thread
public memory	Memory that is accessible and usable by all threads in a system
Reader	The receiver of a message
response	A message sent from an Owner to a User
RTOS	Real-Time Operating System
SCB	STRIDE Transact Identifier (STID) Control Block
SCIP	STRIDE Communication Inter-platform Protocol
SMID	STRIDE Message Identifier. The SMID is a 32-bit structure consisting of a unique ID and a set of message attributes. The SCM defines the SUID portion as the first 24 bits of the SMID (bits 0 — 23), allowing unique identifiers up to 16,777,215. The message attributes use the highest 8 bits of the SMID (bits 24 — 31).
STID	STRIDE Transactor Identifier. The STID is a unique 8-bit value used to represent messaging and tracing operations associated with a native thread as defined by an operating system. STIDs provide a link between the STRIDE transactor and application native transactor IDs.
STPID	STRIDE Trace Point Identifier. The STPID consists of a unique 32-bit ID and an optional payload, and is used for trace points. The STPID value of zero (0) is reserved for the system. There are no constraints on different application threads using the same trace point.
SUID	STRIDE Unique Identifier. The SUID is a unique 24-bit value used to identify interfaces such as function calls and messages.

Related Documents

Other documents available through STRIDE Online Help include the following:

- STRIDE Host Transport Specification
- STRIDE Platform Abstraction Layer (PAL) Specification

1. Using the STRIDE Runtime

1.1. Overview

STRIDE Runtime is a software package that provides services for messaging, remote function calls, and tracing while providing seamless connectivity between the target application and host operating system. STRIDE Runtime standardizes how threads and applications communicate with each other independent of the platform on which they are executing. This eliminates the need to integrate new software on the target hardware all at one time. Developers can incrementally integrate embedded software on a combination of the desktop environment and the target hardware, providing more control over the integration phase. Threads can be divided up between the two platforms without requiring changes to the software. New software functionality currently under development can be easily simulated on the desktop environment while the software using this new functionality can run on the target hardware. The flexibility of choosing how to integrate different pieces of software and the platform it should run on enables developers to detect integration problems earlier in the development process and to correct defects while the impact of the defects is still minimal.

STRIDE Runtime is written in ANSI C and is partitioned into two functional groups, the *STRIDE APIs* and the *STRIDE Runtime Thread*. The STRIDE APIs are a set of public and private routines that run out of the context of the calling thread. These APIs support the public APIs defined in the *sr.h* header file. The STRIDE Runtime Thread is an independent thread that has its own context when executing. The Runtime Thread supports the public mail-based API defined in the *srmsg.h* header file.

Before STRIDE Runtime can be used on the target platform the messaging and tracing configurable memory must be set up and a Platform Abstraction Layer (PAL) must be implemented. The configurable memory is statically defined by the constants found in the *srcfg.h* file. The configurable memory is allocated at compile time and the memory resource requirements can be calculated beforehand. The *srcfg.h* header file allows customization of this memory based on individual projects.

The PAL defines the set of OS functionality required by the platform to support the STRIDE Runtime. The *pal.h* header file defines the PAL functionality. The PAL also defines functionality required by the STRIDE Runtime to transmit and receive packets of data (called I-blocks) using the platform's transport mechanism. These PAL routines enable the STRIDE Runtime to be installed on diverse environments without changing its internal design.

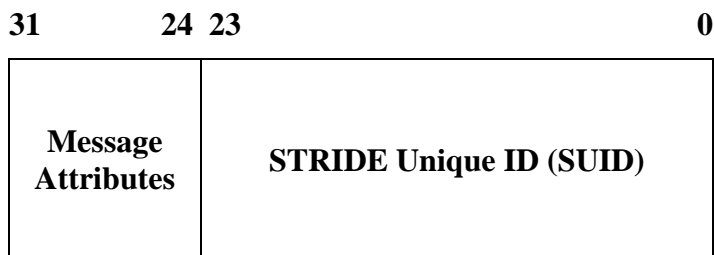
Once the memory has been configured and the PAL functionality implemented, the STRIDE Runtime services can be used by the application threads. For projects that do not want their application threads to use the API directly, a wrapper can be used or a remote message stub thread can be written. A remote message stub thread involves the target environment implementation of a STRIDE Runtime-specific thread whose main objective is to map SCL-compliant messages from the host to their native target environment. The application threads do not change how they currently use messaging, rather the remote message stub thread maps to their environment. Refer to section 2.2 *Using a Remote Message Stub (RMS)* for more details.

The following unique identifiers facilitate a variety of tasks performed by the STRIDE Runtime, such as message ID handling, memory management, payload routing, and data format conformance:

1.1.1. STRIDE Message ID (SMID)

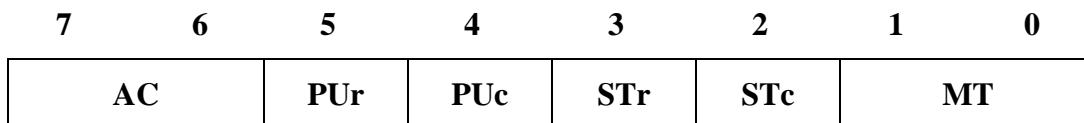
The STRIDE Message ID (SMID) is a 32-bit structure consisting of a unique ID and a set of message attributes. The STRIDE Communication Model (SCM) defines the STRIDE Unique ID (SUID) portion as the first 24 bits of the SMID (bits 0 — 23), allowing unique identifiers up to 16,777,215. The Message Attributes use the highest 8 bits of the SMID (bits 24 — 31).

Each SCL-compliant message within the system must be assigned a unique message ID.



Message Attributes

The attributes are used to declare the message type, describe how to send the option payload and indicate, if applicable, the pointer memory usage attribute for the payload. The attributes, shown below, are the message type (MT), send type for command (STc), send type for response (STr), pointer usage for command (PUc), pointer usage for response (PUr), and Access Class (AC).



Message Type (MT) Attribute

The Message Type (MT) attribute defines the type of message being used for communication between the owner and user. The following values are used for different message types:

- 00 One-way command
- 01 One-way response
- 10 Two-way command-response
- 11 Broadcast message

Send Type (ST) Attribute

The Send Type (ST) attribute is used to indicate how to transmit the payload. There are two ways to send the payload: by **value** or by **pointer**. The STRIDE Runtime is required to use the ST attribute when determining whether to route locally or remotely across platform boundaries. The ST attribute settings are listed below:

Send Type for Command (STc) Attributes

- 0 Pointer
- 1 Value

Send Type for Response (STr) Attributes

- 0 Pointer
- 1 Value

Note: When a command or response does not contain a payload, the ST attribute is set to zero (0).

Pointer Usage (PU) Attribute

When a payload is passed by pointer, a Pointer Usage (PU) attribute is required. The PU attribute indicates if the payload is using **pool memory** or **private memory**. When the PU attribute indicates **pool**, the SCM requires that the memory be allocated from the common pool being used by the system. When the PU attribute indicates **private**, the STRIDE Runtime environment makes no assumptions on how the payload memory is being managed between the Owner and User when they are executing on the same target platform. If the payload crosses platform boundaries, however, the Runtime is required to dynamically allocate memory from the common pool. The temporary memory that is allocated is used to hold the payload, and the address of the memory is passed to the reader. Once the reader returns the message memory to the Runtime, the temporary memory is automatically freed. The original memory from the sender is not affected or synchronized with the other platform. The PU attributes are listed below:

Pointer Usage for Command (PUc) Attributes

- 0 Pool
- 1 Private

Pointer Usage for Response (PUr) Attributes

- 0 Pool
- 1 Private

Access Class (AC) Attribute

The Access Class (AC) attribute ensures that the intercept module will not register each SUID, and function calls with no registered owner will be routed to the intercept module STID. The AC attributes are listed below:

- 00 Message
- 01 Function
- 10 System/Application
- 11 Reserved

1.1.2. STRIDE Unique ID (SUID)

The STRIDE Unique ID (SUID) is a unique 24-bit value used to identify interfaces such as remote function calls and messages. A SUID is required when using a two-way message in order to identify the message user and allow the response payload to be routed correctly. The message user provides this unique ID to the STRIDE Runtime, which in turn sends it to the message owner. The message owner is then required to supply this same unique ID when sending a response back to the STRIDE Runtime, so that the response payload can be routed correctly.

1.1.3. STRIDE Transactor ID (STID)

The STRIDE Transactor ID (STID) is a unique 8-bit value used to represent messaging and tracing operations associated with a native thread as defined by an operating system. STIDs provide a link between the STRIDE transactor and application native transactor IDs.

The STRIDE Runtime allocates resources for each STID created within the system, as well as configures the maximum size of a STID name (via `#define srCFG_STID_NAME_SIZE` in `srcfg.h`) for the target.

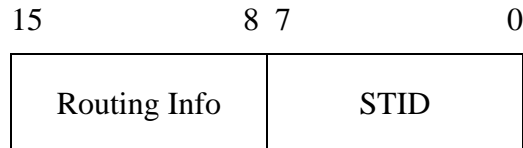
Generally, each thread will have its own STID.

1.1.4. STRIDE Trace Point ID (STPID)

The STRIDE Trace Point ID (STPID) consists of a unique 32-bit ID and an optional payload, and is used for trace points. The STPID value of zero (0) is reserved for the system. There are no constraints on different application threads using the same trace point. The trace point payload is used to define the format of the data associated with it. The only additional constraint unique to trace point payloads versus message payloads is that there is no support for embedded pointers. The memory block pointed to is not transferred over to the host Runtime environment. Trace points also use payloads to represent debug information that is being provided to the host Runtime environment.

1.1.5. STRIDE Response ID (SRID)

The STRIDE Response ID (SRID) is defined as a unique 16-bit value required when using a two-way message. It is not required for any other type of message. It identifies the message User and enables routing of the response payload. The User then provides this unique ID to the Runtime, which in turns provides it to the message Owner. The Owner is required to supply this same SRID when sending a response back to the Runtime, in order to enable proper routing of the response payload. The format of the SRID is shown below; the first 8 bits are reserved for routing information that is vendor-specified.



1.1.6. Notification of Traffic ID (NID)

The Notification of Traffic ID (NID) is defined as a unique 32-bit value used to represent traffic pending from the STRIDE Runtime. Each STID created within the system is required to store an associated NID used by the Runtime when notifying the native platform. The Runtime passes back the stored NID when signaling the platform of pending traffic for a specific STID.

1.2. Memory Requirements

STRIDE Runtime requires four types of memory to support its messaging and tracing services:

Program	For most applications, the program size of STRIDE Runtime ranges from 20 to 45 Kbytes depending on the compiler and underlying target processor.
Internal	<p>The STRIDE Runtime requires static memory for internal use. This memory is not configurable and typically ranges from 1 Kbytes to 10 Kbytes depending on the compiler and underlying target processor. The STRIDE Runtime Thread also requires memory for stack usage. This memory used by the STRIDE Runtime is requested through the <i>palMemSegmentOpen()</i> and <i>palMemSegmentClose()</i> routines.</p> <p>Note: This does not include space required for interrupt services routines that use the same stack as the interrupted thread.</p> <p>In case of multi-process target is enabled, this memory should be shared among applications.</p>
Dynamic	<p>All memory requirements for payload storage are requested through the <i>palMemAlloc(..)</i> and <i>palMemFree(..)</i> routines. There are no limits imposed by the STRIDE Runtime on how much memory is available at one time to support transferring message payloads through the system.</p> <p>If the STRIDE Runtime Memory Management is enabled, dynamic memory will be allocated from the <i>configurable memory</i>.</p>
Configurable	<p>The configurable memory used by the STRIDE Runtime is requested through the <i>palMemSegmentOpen()</i> and <i>palMemSegmentClose()</i> routines. The blocks of memory that are configurable are defined in the <i>srcfg.h</i> file. The constants contained in the file should be customized for each project based on its specific requirements.</p> <p>Note: In case of multi-process target is enabled, this memory should be shared among applications.</p>

Most of the memory usage required by the STRIDE Runtime is configurable. The configurable memory is divided into two functional groups: (1) Messaging, and (2) Tracing.

1.2.1. Messaging Memory

The STRIDE Runtime requires configurable messaging memory to be configured at compile time. Each project can use the constants contained in the *srcfg.h* file to customize their own memory requirements. Figure 3 contains the constants and their default values related to messaging.

```
#define srCFG_TOTAL_STIDS      16
#define srCFG_TOTAL_SUBCS     20
#define srCFG_TOTAL_PTRS      30

#define srCFG_SUID_TABLE_TYPE  1
#define srCFG_SUID_TABLE_SIZE 225

#define srCFG_TOTAL_SUIDS_QUED 50

#define srCFG_STID_NAME_SIZE   15

#define srCFG_SUID_HASH_FUNC(x)
(srWORD)((x)%(srCFG_SUID_TABLE_SIZE))
```

Figure 3: Configurable Defines for Messaging

The number of *STRIDE Transact IDs (STIDs)* supported by STRIDE Runtime is defined by the constant *srCFG_TOTAL_STIDS*. The STRIDE Runtime will pre-allocate an internal *STID Control Block (SCB)* for each of the potential STIDs that can be created. A single *STID Control Block* requires 278 bytes. The SCB contains the Notification Identifier (NID), tracing control flags, pre-allocated mailboxes, and the name of the STID. Thus, the total memory required for STIDs is equal to $278 * srCFG_TOTAL_STIDS$. The maximum number of STIDs that can be supported by the STRIDE Runtime is 255.

You can configure the STRIDE Runtime to use either an index-based or search-based SUID table by setting the constant *srCFG_SUID_TABLE_TYPE*. By default, the Runtime uses a search-based SUID table. A value of zero (0) will configure the Runtime to use the index-based SUID table. A value of one (1), the default value, will cause the Runtime to use a search-based SUID table.

The index-based SUID table is the most efficient for the Runtime to access, given the highest of the range of SUIDs is a lesser number. The search-based table is most efficient for memory usage. The number of SUID entries for either SUID table type is defined by *srCFG_SUID_TABLE_SIZE*. This constant is used to statically allocate an internal *Message Control Block (MCB)* for each SUID entry. The STRIDE Runtime can support up to 65534 unique SUIDs. However, the value of a SUID can be up to 24 bits (16777215). Note that it is highly recommended to use a search-based SUID table if high values are used for SUIDs. A single *Message Control Block* requires 4 bytes for an index-based SUID table and 10 bytes for a search-based SUID table.

If you use an index-based SUID table, the SUIDs numbering starts at zero and ends at the table size minus one (1). The SUID is the actual index into the SUID table. This makes for a very efficient way to access SUID information. If you use a search-based SUID table, the SUID table uses the first available SUID entry to store the SUID information. When accessing SUID information, the Runtime searches the SUID entries to find the correct SUID. The Runtime uses a Hashing algorithm to make the searching more efficient. The Hashing function can be altered by setting the `srCFG_SUID_HASH_FUNC` macro.

When a message is sent to a specific STID, it is placed in a *Message Queue Entry (MQE)* that corresponds to the receiving STID. A queue entry contains a SMID, a pointer to an optional payload, the size of the payload and the index to the next entry, if it exists. The STRIDE Runtime pre-allocates a number of these queue entries at compile time. The total number of messages that can be queued at one time is configurable and defined by the `srCFG_TOTAL_SUIDS_QUED` constant. This number represents the total number of messages that can be pending at any one time. STRIDE Runtime can support up to 3000 *Message Queue Entries*. A single *Message Queue Entry* requires 16 bytes.



The STRIDE Runtime also uses *Message Queue Entries* from this pre-allocated pool for its own internal messaging.

For broadcast message types any number of users, up to 200, can subscribe to the response payload. The STRIDE Runtime places the subscriber information into a *Message Subscribe Entry (MSE)*. A *Message Subscribe Entry* contains the routing information for the subscriber and an optional pointer to another entry. The total number of subscribers available for the system at any one time is configurable and defined by the `srCFG_TOTAL_SUBCS` constant. The STRIDE Runtime can support up to 200 subscribers. A single *Message Subscriber Entry* requires 8 bytes.

For messages that contain pointers within a payload, the `srEPtrSetup(..)` routine must be called by the sender. This provides the Runtime with the bookkeeping information required for marshaling. There is no limit to the number of pointers a single payload can contain. For every pointer the STRIDE Runtime pre-allocates a *Pointer Entry (EPE)*. The *Pointer Entry* contains the pointer address, size, offset, type and directional attribute of the pointer. The total number of pointers available for the system at any time is configurable and defined by the `srCFG_TOTAL_PTRS` constant. The STRIDE Runtime can support up to 4000 pointers. A single *Pointer Entry* requires 14 bytes.

Table 2 below is an example of the memory required based on the values found in Figure 3 .

Table 2: Messaging Memory Allocation Example

Item	Maximum	Example	Entry Size	Memory
STIDs	255	16	278	4448
SUIDs	65534	225	4	900
Queued Entries	3000	50	16	800
Subscribers	200	20	8	160
Pointers	4000	30	14	420
Totals				6728



Entry sizes are calculated using bytes. It is assumed that no structure padding is taking place.

1.2.2. Tracing Memory

The amount of memory and processing time used for tracing can be controlled by adjusting the parameters of the tracing constants. Figure contains the constants and their default values related to tracing.

```
#define srCFG_TOTAL_TRACING_MEMORY      4096
#define srCFG_TRACEBUFFER_MAX_SIZE     1000
#define srCFG_TRACEBUFFER_WAKEUP_TIME  100
```

Figure 4: Configurable Defines for Tracing

The amount of memory allocated for tracing is determined by the *srCFG_TOTAL_TRACING_MEMORY* define. This is the amount of memory allocated for the buffer that holds the trace entries before they are broadcast. Maximum tracing memory STRIDE Runtime can support is 65000.

The *srCFG_TRACEBUFFER_MAX_SIZE* define establishes the maximum size of each trace broadcast. Maximum trace buffer size STRIDE Runtime can support is 64000.

The *srCFG_TRACEBUFFER_WAKEUP_TIME* define determines the fastest rate at which the STRIDE Runtime will broadcast trace buffers.

1.2.3. Transport Settings

The maximum size of an I-block sent over the remote link can be defined by setting the `srCFG_MAX_TRANSPORT_UNIT` define. This value is used to determine when the Runtime will fragment an I-block. I-blocks larger than the max transport unit are broken up into smaller I-blocks, each not larger than the define value. A value of zero will disable any fragmentation. Your dynamic memory allocation system must be able to allocate a memory block at least as big as the size of this define (if it's not zero). If no fragmentation is used then you must be able to allocate a memory block as large as the largest I-block. The I-block size is determined by the size of your messages being sent over the link and your settings for the STRIDE tracing system.

```
#define srCFG_MAX_TRANSPORT_UNIT 2048
```

Figure 5: Configurable Defines for Transport

The initial state of the target transport is set through the `srCFG_DEFAULT_TRANSPORT_STATE` define. A value of one (1) indicates that the transport is initially ready transmit data, and if the target Runtime needs to, an I-block to the host it will do so. If a value of zero (0) is used, the Runtime will not send out an I-block until the state is changed by using the routine registered with the PAL through the `palOutRdyReg()` routine (for more information, refer to the STRIDE Platform Abstraction Layer (PAL) Specification).

```
#define srCFG_DEFAULT_TRANSPORT_STATE 1
```

Figure 6: Configurable Defines for Transport

1.2.4. Memory Management

The STRIDE Runtime can manage dynamic and configurable memory required. This is optional for single-process target but is a required for multi-process target. Each project can use the constants contained in the `srcfg.h` file to customize memory requirements. If memory management is set to enable, the block sizes and the maximum limits of memory segments for dynamic and configurable memory should be configured.

In case of multi-process target is enabled, all dynamic, configurable and internal static memory should be shared among applications and will be allocated and managed by the STRIDE Runtime's memory management module `srMem`.

Figure 7 contains the constants and their default values related to memory management.

```
#define srCFG_MEMORY_MANAGEMENT          0

#if srCFG_MEMORY_MANAGEMENT
#define srCFG_MEMORY_BLOCK_SIZE_SMALL   30
#define srCFG_MEMORY_BLOCK_SIZE_MEDIUM 100
#define srCFG_MEMORY_BLOCK_SIZE_LARGE   500
#define srCFG_MEMORY_BLOCK_SIZE_LARGE2 1000
#define srCFG_MEMORY_BLOCK_SIZE_LARGE3 10000
#define srCFG_MEMORY_BLOCK_SIZE_HUGE    0xFFFF

#define srCFG_MEMORY_BLOCK_MAX_SMALL     5000
#define srCFG_MEMORY_BLOCK_MAX_MEDIUM    250
#define srCFG_MEMORY_BLOCK_MAX_LARGE     250
#define srCFG_MEMORY_BLOCK_MAX_LARGE2    100
#define srCFG_MEMORY_BLOCK_MAX_LARGE3    50
#define srCFG_MEMORY_BLOCK_MAX_HUGE      50
```

Figure 7: Configurable Defines for Memory Management

The enabling and disabling of memory management in the STRIDE Runtime is defined by the constant *srCFG_MEMORY_MANAGEMENT*. The STRIDE Runtime's memory management module *srMem* determines the sizes and maximum number of memory blocks based on the parameters in *srcfg.h*. The size of dynamic memory allocated from shared memory is dependent on the dynamic memory requirements such as the size of the STRIDE messages defined in your system, the number of messages outstanding, and the memory size defined for your trace logs. The size of internal static memory, used by the STRIDE Runtime for bookkeeping purposes, depends on each of the STRIDE Runtime module data.

1.2.5. Multi-Process Target

The STRIDE Runtime can support multiple-process Target that runs several applications. The enabling and disabling of multi-process target in the STRIDE Runtime is defined by the constant *srCFG_MULTI_PROC_TARGET*.

In case of multi-process target is enabled, all dynamic, configurable and internal static memory will be shared among applications and allocated and managed by the STRIDE Runtime's memory management module *srMem*.

```
#define srCFG_MULTI_PROC_TARGET          0
```

Figure 8: Configurable Defines for Multi-Process Target

1.3. Using the API

After configuring the parameters related to memory usage, and completing the PAL layer (see the *STRIDE Platform Abstraction Layer Specification*), the STRIDE Runtime can be downloaded to the target or included in the target build. The STRIDE Runtime must first be initialized by calling *srInit()*. Following initialization, the STRIDE Runtime Thread, with the entry point *srThread()*, must be started in the RTOS. At this point the STRIDE Runtime is available for general use.

1.3.1. Creating an STID

A *STRIDE Transact ID (STID)* is required before the services from the STRIDE Runtime are used. The STID is the logical identifier representing resources allocated by the STRIDE Runtime. This unique identifier is required as an input parameter to nearly all of the STRIDE Runtime APIs. The *srCreateSTID(..)* routine is used to create the STID and enables you to provide the required input parameter *Notification Identifier (NID)*. The STRIDE Runtime stores the NID associated with the STID and only uses it when the *PalNotify(..)* routine is called.

For best performance using the STRIDE Runtime, keep a copy of the STID for use when invoking services. The STRIDE Runtime uses the STID to directly access (without searching) the associated resources. Using the *srQueryNID(..)* routine to look up the STID before each STRIDE Runtime call can adversely affect system performance.

Note: An STID alphanumeric name can be from 0 to a configurable length (`srCFG_STID_NAME_SIZE`). A empty string (zero-length) may assigned if it is desired that transaction originating or terminating on this STID are not to be traced on.

1.3.2. Creating a STRIDE Message

A STRIDE Message can be defined by adding STRIDE message attributes to your message IDs. These attributes define the type of message you are using. STRIDE message types are defined as one-way, one-way command, one-way response, two way and broadcast messages.

You also need to define whether the payload is sent by value or pointer. Payloads sent by pointer require that you also define which type of memory is being used, private or pool.

1.3.2.1. The One-way Message

The STRIDE one-way message is used as a generic message with no inherent sense of command or response. It is simply a message sent in one direction. The one-way command is the same as a one-way but is expected to be sent as a command. The one-way response is not directly connected to the one-way command but can be sent as a response to a received one-way command. A one-way and a one-way command must be registered in order for the message to be received. The one-way response does not need to be registered but will be sent to the recipient specified in the message instance provided when the one-way command was read.



The one-way command and one-way response do not share the same SUID, or

STRIDE Unique Identifier.

1.3.2.2. The STRIDE Two-way Message

The STRIDE two-way message is similar to the one-way command and one-way response. However, with the two-way message the command and response portions share the same SUID with different payloads defined. The SUID is registered by the “owner”, and as with the one-way response, the response is not registered but is instead sent back to the recipient in the message instance.



A service *owner* provides a given service to others (i.e., it calculates a checksum, polls a semaphore, or performs some other activity that can be classified as a service). The owner of the messaging interface receives a command from the users to obtain the service.

1.3.2.3. The STRIDE Broadcast Message

The STRIDE broadcast message is similar to the one-way response message and the response portion of the two-way message. However, with the broadcast message the recipient is determined through the use of the subscriber concept. Delivery is determined by subscription. A broadcast message is used to send a response to a set of Users based on the Runtime's subscriber list. To send a broadcast message, the `srBroadcast(..)` routine is used. A broadcast message's Send Type (ST) attribute must be set up as a response.

```

/* Message Types (MT) */
#define srMT_ONE_CMD    0x00000000
#define srMT_ONE_RSP    0x01000000
#define srMT_TWO        0x02000000
#define srMT_BRD        0x03000000

/* Abbreviations MT */
#define srMT_ONE        srMT_ONE_CMD
#define srMT_ONEc       srMT_ONE_CMD
#define srMT_ONEr       srMT_ONE_RSP

/* Send Type for Command (ST_CMD) */
#define srST_CMD_PTR    0x00000000
#define srST_CMD_VAL    0x04000000

/* Send Type for Response (ST_RSP) */
#define srST_RSP_PTR    0x00000000
#define srST_RSP_VAL    0x08000000

/* Pointer Usage for Command (PU_CMD) */
#define srPU_CMD_POL    0x00000000
#define srPU_CMD_PRI    0x10000000

/* Pointer Usage for Response (PU_RSP) */
#define srPU_RSP_POL    0x00000000
#define srPU_RSP_PRI    0x20000000

/* Access Class (AC) */

```

```
#define srAC_MSG          0x00000000
#define srAC_FUNCTION    0x40000000
#define srAC_SYS         0x80000000
```

Figure 9: STRIDE Message Types

1.3.3. Registering Messages

Owners of one-way command messages, two-way messages and functions are required to register ownership with the STRIDE Runtime. The *srRegister(..)* routine is used to establish this ownership. The STID and associated SMID (for messages) or SUID (for functions) are required for this operation. A STID owns the message and only one Owner of one-way and two-way messages can exist at one time. The STRIDE Runtime uses the registration information to properly route these message types between Owners and Users. If the message is not locally registered the STRIDE Runtime can potentially route the message to another platform.



Broadcast message types cannot be registered.

1.3.4. Overriding Registration

Any message that is registered with the STRIDE Runtime can be overridden by the host Runtime environment. This overrides and takes the ownership of any currently registered one-way command message, two-way message or function. Any future calls will be routed to the local override owner based on override registration.

1.3.5. Subscribing to Messages

Any STID can subscribe to a broadcast message type. The *srSubscribe(..)* routine is used to place a STID into the runtime's subscription list for a specific message. There are no limits on the number of subscribers for a broadcast message type.

When a payload is broadcast that uses pool memory, either for the payload (e.g., *srPU_RSP_POL*) or pointers, the STRIDE Runtime will create a copy of each memory block allocated from the pool for the second and subsequent subscribers. This enforces the rule that the sender always allocates and the reader always frees. If there are no subscribers, the STRIDE Runtime handles the release of any pool memory associated with the payload.

When a payload that uses private memory is broadcast, a copy is made for each subscriber then the memory is returned by calling the *srReadComplete* routine.



It is not possible to subscribe to one-way and two-way messages.

1.3.6. Reading and Sending Messages

The STRIDE Runtime supports one API for reading messages and three APIs for sending messages. These routines are:

- `srRead(..)`
- `srSendCmd(..)`
- `srSendRsp(..)`
- `srBroadcast(..)`

The `srRead(..)` routine allows the reading of commands and responses. Both a command and a response can have a payload associated with it. Payloads are sent “By Value” or “By Pointer”, which is determined by the attributes associated with the message.

When reading a message, the caller must allocate enough memory to hold the biggest “By Value” payload that it might receive. “By Pointer” payloads store only the address of the payload, which is typically 4 bytes.

When sending commands or responses, there are three different routines to choose from depending on the context.

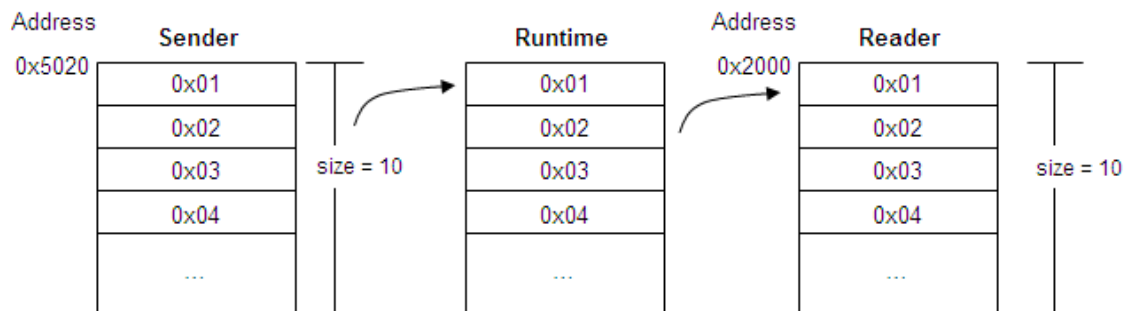
A User can send a one-way or two-way command using the `srSendCmd(..)` routine.

An Owner can send a two-way response using the `srSendRsp(..)` routine. (One-way messages do not have responses.)

A broadcast message is used to send a response to a set of Users based on the Runtime’s *subscriber list*. To send a broadcast message, the `srBroadcast(..)` routine is used. A broadcast message’s *Send Type (ST)* attribute must be set up as a response.

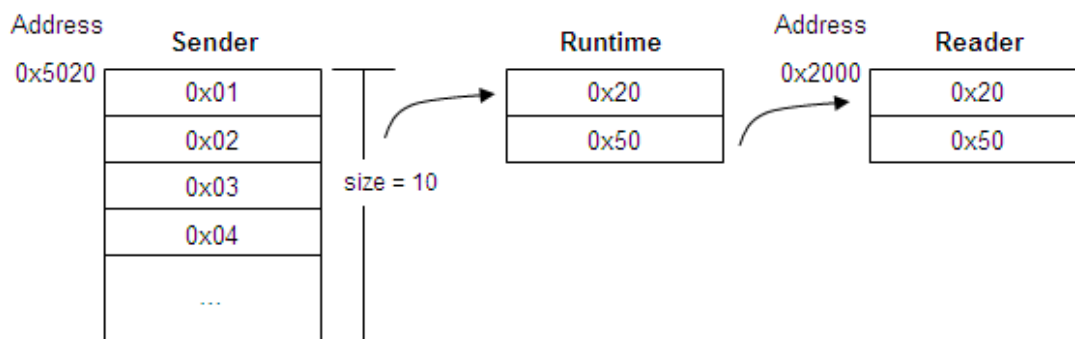
1.3.6.1. Send Type by Value

When the Send Type attribute is set by Value (Send Type by Value), the entire payload is copied by the Runtime from the payload sender's memory space to the receiver's memory space. The example below shows a sender transferring a payload of 10 bytes using 16-bit addresses. The sender makes a call to the Runtime, passing the address and size of the payload. The Runtime makes a local copy of the data, using common pool memory, and returns to the caller. The sender can now reuse the memory that was holding the original payload. When the reader calls the Runtime, an image of the payload is copied into its own address space and any temporary storage used to hold the payload by the Runtime is released.



1.3.6.2. Send Type by Pointer

When the Send Type attribute is set by Pointer (Send Type by Pointer), only the address of the payload is copied from the sender's memory space to the reader's memory space. The example below shows a sender transferring a payload of 10 bytes via a pointer using 16-bit addresses and Little Endian byte ordering. The sender makes a call to the Runtime, passing the size and address of the payload. The Runtime makes a local copy of the address and size and returns to the caller. The sender cannot reuse the memory holding the payload at this time. The address of the payload is copied into the reader's address space when the reader calls the Runtime. This address points to the original payload data created by the sender. The reader also receives the size of the payload when issuing the read call. If the payload's Pointer Usage attribute is set to pool, it is the responsibility of the reader of the payload to free the associated memory. A pool memory setting implies that the pointer memory has originated from a common area to which all application threads have access. If the Pointer Usage attribute is set to private, the reader does not free the associated memory; it is freed by the STRIDE Runtime.



1.3.7. Using Pointers

When a payload contains a pointer, there are special requirements that must be followed. The STRIDE Runtime supports pointers as defined in the *STRIDE Communication Model* (refer to Chapter 3). The STRIDE Runtime requires **each** pointer defined in a payload to be both set up and attached to before it can be successfully marshaled across platform boundaries. The setup process is used to inform the Runtime of a pointer's existence and provide information required for marshaling. The *srEPtrSetup(..)* routine is required to be called once for each pointer. The routine returns a pointer handle, which is a required input parameter to the attach routine. The setup routine is passed information concerning the STID, the SMID, the offset of the pointer, the message direction (command or response), the pointer direction (input or output) and whether freeing is required by the reader. If a nested pointer is used, the *srNestedEPtrSetup* routine needs to be called. The *srEPtrAttach(..)* routine is used to inform the Runtime of the actual pointer address and the corresponding size. The attachment routine must be called for each pointer, initially. Subsequent calls are only required if the size or address changes.

Listing 1: Example Using Setup and Attach Routines

```
srEPtrSetup(.., &MyEPtrHandle ); // Only required once

MyPayloadCmd.field1 = 10;
..

MyPayloadCmd.Ptr = Allocate( sizeof( MyType ) );

srEPtrAttach( MyEPtrHandle, MyPayloadCmd.Ptr, sizeof( MyType ) );

srSendCmd(..);
```

Each payload that contains a pointer requires a unique setup and attachment by each sender. The STRIDE Runtime maintains the bookkeeping information for each User and Owner attaching to a pointer. The only constraint regarding the number of unique pointers defined within a single payload is determined by the overall configuration. The total number of pointers supported by the STRIDE Runtime is configurable and defined by `srCFG_TOTAL_PTRS`. This total includes all of the unique pointers defined within the payloads multiplied by the total number of senders.

1.3.8. Returning Message Memory

The STRIDE Runtime supports the concept of a Reader of a message not freeing pointer memory associated with the payload. This type of memory is defined as *private*. Pointer memory in this context refers to both the payload (sent “By Pointer”) and a pointer. When a reader of a message frees any pointer memory, the STRIDE Runtime is required to initially allocate memory from the same pool of memory available to all applications. This type of memory is defined as *pool*. Pool memory is used (either directly or indirectly) by the *palMemAlloc(..)* and *palMemFree(..)* routines. To support passing private memory, the STRIDE Runtime requires each application that reads a payload using private memory to “return” the message memory back to the Runtime environment. This enables the STRIDE Runtime to release any resources required to support the marshaling of the private memory across platform boundaries. Resources are only allocated when using private memory and sending a message across platforms.

The *srReadComplete(..)* routine requires the Message Instance ID as an input parameter. The Message Instance ID represents the unique instances of the message corresponding to the sending STID.

Listing 2: Example Using the Read Complete Routine

```
for (;;) {  
    // Wait for new message  
  
    CustomerWait (...);  
  
    // Reading a message Payload  
  
    srRead(..);  
  
    ..  
  
    // Respond to any 2-ways using EPptr with OUT/INOUT before returning  
    ..  
  
    srReadComplete( wMySTID, dwMsgInstFromRead );  
}
```

1.3.9. Pointer Memory Policies

Sending payloads by pointer and using pointers provides an efficient method for passing data between threads — no data is copied. Pointers can also be used to construct more advanced data types. It is important to note that using these methods creates more responsibility for the application threads using them. There are a number of constraints that must be adhered to in order to support "transparent interfacing" when using pointers and to enable successful marshaling of data between platforms. The memory management requirements described below apply to application threads that send payloads by pointer and/or use pointers:

- There can only be one User of a pointer at any given time. Because interfaces can cross platform boundaries, only one application thread can have access to the pointer at a time (ownership of memory passes between the User and the Owner when exchanging data). It is recommended that a two-way message type always be used when sending a payload by pointer whose Pointer Usage attribute is set to private. This ensures that only one User uses a pointer at a time.
- When transferring a pointer that will require freeing, the sender always allocates the memory from a common pool and the reader always frees it. This applies to both a User sending a command payload and an Owner sending a response payload.

- The reader returns to the Runtime any pointer memory used for a payload that does not require freeing. This is required in order to support the marshaling of payloads across platform boundaries whose pointers are not being freed by the reader (Pointer Usage attribute set to private). The SCM requires that the Runtime dynamically allocate temporary memory from the common pool to hold pointer memory when crossing platform boundaries. If the reader is not required to free the memory, then it is the responsibility of the Runtime to free the temporary memory once it has been returned. Once a pointer has been returned by the reader to the Runtime, the memory cannot be dereferenced.
- Only a command's payload in two-way messages and ParameterLists can contain pointers with the OUT or INOUT directional attribute. Owners of these types of messages are required to respond to the message User. When crossing platform boundaries, temporary resources are allocated by the Runtime and are not released until a response to the message is received.
- Embedded pointers with OUT or INOUT directional attributes can only be used with two-way message types. This enables the Runtime to update the sender's original pointer memory.

1.3.10. Trace Points

Trace points are used to instrument the source code at a more detailed level than the standard message tracing. The STRIDE Runtime itself uses trace points to indicate errors and provide detailed error information. The STRIDE Trace Point ID (STPID) uses the most significant bit as a reserved bit to differentiate a system trace point from a user-defined trace point. When a connection exists, the Runtime will insert all system trace points into the trace buffer. If no connection exists, and no target filters are set, then no trace points will be placed in the trace buffer. This is done to minimize the system impact.

1.3.11. Data Format Conformance

Platform-dependent characteristics include items such as the Endian ordering, enumeration sizes, pointer sizes, integer size, and structure alignment boundaries. The SCM requires that the target platform's native data type formats must be conformed to by the host platform when exchanging messages. The host Runtime marshals payload data when communicating with the target platform. The burden is always on the host platform to map the differences between it and the target's payload data type formats, which reduces processing and memory requirements for the target platform. Messages are always exchanged according to the target memory format.

1.4. Routing with Access Class Registration

Access Class Registration is used to route non-registered messages and functions between the host and target platforms. There are two types of Access Classes:

- Access Class Messages – for Remote Messaging (RM) that includes Remote Message Stub (RMS) and Remote Message Proxy (RMP).
- Access Class Functions – for non-registered functions in Intercept Module (IM).

The routing of messages is described as Remote Messaging (RM) while that of functions is described as Access Class Intercept Module.

1.4.1. Remote Messaging (RM) Overview

A Remote Messaging (RM) service provides the glue between the native messaging system and STRIDE. It generally runs as a separate task on the Target to act as a forwarding agent between the Target and STRIDE by translating the message between STRIDE and Target message formats, often by wrapping the Target message datagram inside an equivalent STRIDE message.

Remote Messaging has two concepts, namely, Remote Message Stub (RMS) and Remote Message Proxy (RMP). The RMS can be used to receive STRIDE messages from the Host and translate them to native messages. The RMP can be used to receive native commands from native threads and translate them into STRIDE messages, which can be routed to the Host.

See section 2. *Remote Messaging (RM)* for more details.

1.4.2. Access Class Intercept Module

In some resource constrained environments, you might not want to register all your functions owned on the device with the STRIDE Runtime. By registering the Intercept Module (IM) as an Access Class (AC), all the non-registered functions could be routed to the STID associated with the IM.

This is useful when the search-based SUID table is used and the total number of functions is large because the SUIDs do not have to be registered regardless of they are in use or not. Remember that registering SUIDs (functions) will cause the Runtime to allocate an entry in the SUID table for each function registering.

1.4.2.1. Access Class IM Setup

The Intercept Module (IM) can register itself as an Access Class by calling once at startup the STRIDE Runtime API *srRegisterAccessClass* with the Access Class type *srAC_REG_FUNCTIONS* along with the STID and the mail box to which you want the unregistered functions routed.

To generate an Access Class IM, you can simply enable the Access Class registration check box in the IM Wizard GUI or use the STRIDE Studio automation functionality.

1.4.2.2. How It Works

If you call a non-registered function and the IM is registered as an Access Class, the function call will be routed to the IM. IM can be registered as an Access Class either on the host or the target. Routing of a function call is handled by the STRIDE Runtime according to the following sequence of order.

1. Routes to the local override owner based on override registration
2. If not, routes to the local owner base on SUID registration
3. If not, routes to the local IM based on Access Class registration
4. If not, routes to the remote platform
5. The remote platform would go through the steps 1 through 3 again

1.5. Connecting to the Host

Host Transport should initiate to a connection between the host and target platforms. Once there is a connection, inter-platform messaging and tracing will be available.

The connection method involves using a mail service found in the *srmsg.h* file through which the *srCONNECT_OPEN_T_SMID* message can be used to establish a connection.

When instructed to establish a connection, the STRIDE Runtime executes the connection protocol as defined by SCIP. The routing policy comes into place if a message is not registered locally and there are messages registered on the remote platform. The message is then automatically routed across the link to the other platform. This allows for Owners and Users to be located on either platform without changing any code. If a connection fails to open, the STRIDE Runtime will continuously listen for a connection request from the host.

To get updates on the current state of the connection a user can subscribe to the *srCONNECT_STATUS_B_SMID*. If the target needs to disconnect from the host the *srCONNECT_CLOSE_T_SMID* system message is used.

1.5.1. Connection Settings

Maximum allowed duration for a connect request to establish a connection with the Host can be configured using the define *srCFG_CONNECTION_TIMEOUT* in the *srcfg.h*.

```
#define srCFG_CONNECTION_TIMEOUT 5000
```

Figure 10: Configurable Defines for Connection

2. Remote Messaging (RM)

In some cases a native messaging system may not allow messages to be sent between your target platform and your Desktop. Using Remote Messaging, transparent messaging can be achieved between the STRIDE Host environment and the native target platform. A remote message is defined as any message not registered with the STRIDE Runtime. By using remote messages you can support your native messaging without having to use STRIDE SUIDs or subscribe resources.

Remote Messaging requires that the STRIDE Runtime be enabled on your target in order to allow communication between the STRIDE desktop environment and the native messaging environment on the target. The STRIDE Runtime provides several features which allow the STRIDE messaging model to interact with the native messaging system.

Remote Messaging has two concepts, namely, Remote Message Stub (RMS) and Remote Message Proxy (RMP). The RMS can be used to receive STRIDE messages from the Host and translate them to native messages. The RMP can be used to receive native commands from native threads and translate them into STRIDE messages, which can be routed to the Host.

Implementing the RMS and RMP requires a thorough understanding of the STRIDE system.

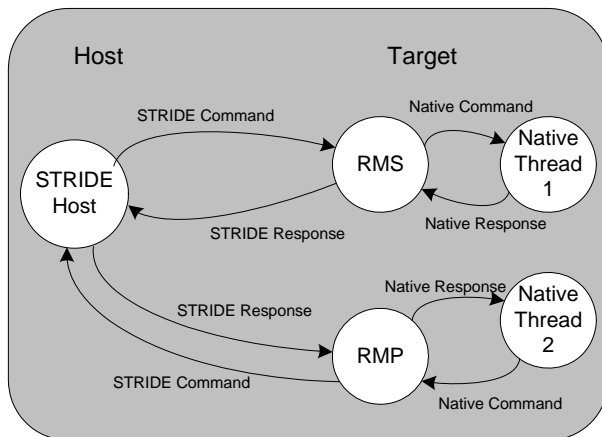


Figure 9: Remote Message Overview

2.1. Implementing a Remote Messaging Service

2.1.1. How It Works

If you send a non-registered message from the host to the target and the target does not have the message registered, the message will be routed to the Remote Message Stub (RMS) on the target. The RMS will identify itself by calling once at startup the STRIDE Runtime API *srRegisterAccessClass* with the Access Class type *srAC_REG_MESSAGES* along with the STID and the mail box to which you want the unregistered messages delivered. Once the RMS receives the remote message, it is free to translate the message into its native message format and deliver it to the native threads.

The remote messages received from the STRIDE Runtime will be accompanied by a “message instance” value, which can be used to route a response back to the sender without knowing the identity of the sender. If you need to match a response to a command received, you can save the message instance received and return it to the Runtime when you send the response. The message instance must be used to route one-way responses and two-way responses.



Responses are always routed to the sender regardless of the SUID value, as long as the proper message instance is sent with the response.

If coordinating responses with the message instance is difficult, you will have to use one-way messages for your responses. If these one-way messages are registered on the host, your responses will be routed correctly without using a message instance.

The RMS can also subscribe to a STRIDE system message that will broadcast remote subscriber information received by the Runtime. Any subscribe request that comes over the link will be forwarded to the RMS. Even if the SUID for which the subscribe request is intended has no SUID entry, the RMS will still receive the subscribe info. This is useful when subscribing to non-STRIDE messages that the RMS and native target support. This is also useful when using the target without any subscribe entries. The RMS can handle all the subscriber lists for its native subscribes.

Broadcasts by the RMS that have no subscribers locally will automatically be forced to the remote platform. This can be useful when no SUID entries are used on the target but broadcasts need to be sent remotely.

The Remote Message Proxy (RMP) can be used to receive native commands from the native threads and translate them into STRIDE messages which can be routed to the Host platform. Responses sent from the host will be routed back to the RMP and then can be re-translated into native messages that can be sent back to the native threads.

The RMP will also receive all broadcasts sent from the Host to the Target that are not subscribed to on the Target. This allows the native platform to use Broadcasts without having to allocate any subscribe or SUID resources.

2.1.2. Issues to Consider

There are many issues to consider when using Remote Messages. Your decisions will depend on your individual platform's requirements. Your RMS and RMP threads can also function exactly like normal STRIDE threads. You can register, broadcast, and send and receive STRIDE messages. You are free to construct your STRIDE environment to meet your project's requirements.

Allocated STRIDE Resources

- Do you need to allocate a SUID table?
- Is your SUID table search- or index-based?
- How many SUID entries do you need?
- What do you need SUID entries for?

SUID Value Organization

- Which messages do you need to register?
- How will you translate between your native messages and your STRIDE SUIDs?
- How will you create a STRIDE SMID from your native message ID? You will have to add specific STRIDE message attributes to route the message properly.

Broadcasting

- Will you use broadcasting?
- Who will broadcast? The RMS thread? Your native threads?
- If your native threads broadcast, will the RMS thread receive the broadcasts and forward them to the STRIDE Runtime?
- If your RMS thread receives native broadcasts to be forwarded to the host platform, how will the RMS thread identify the broadcasts?

Subscriber Lists

- Do you need a subscriber list, or will you force your broadcasts remotely?
- Will you maintain your own subscriber list or let the Runtime maintain it for you?
- If you use an index-based SUID table, is the SUID for your subscription within the range identified in your SUID table? If you use a search-based SUID table, you do not have this issue. However, did you allocate enough SUID entries?

Notification

- How does the RMS thread determine who notified it (the STRIDE Runtime or one of your native threads)? If your RMS thread reads mail from both systems, you need to be able to tell the difference.

Response Messages

- Do you need to send responses to messages received?
- How will you route responses? Will you use a registered one-way, or a one-way response with message instance? Or, will you use two-way messages?
- How will you differentiate between commands and responses?

Pointer Usage

- How many SUIDs will you need to support your messages with pointers?
- If you use an index-based SUID table, are your pointer SUIDs within the identified range? This ensures they have a valid SUID entry.

Tracing

- You must create an STID for tracing.

2.1.3. Translating between STRIDE and Native Message IDs

There are many ways to translate between STRIDE messages and your native messages. You can use static lookup tables (or switch statements) that map one message directly to another. Or you can have some type of algorithm for doing a generic transformation. Since the STRIDE Runtime routes un-registered messages to the RMS, you can use your own native message IDs for the STRIDE messages, but leave them unregistered. Then mask off the STRIDE Message ID (SMID) attributes (the high 8 bits) which leaves you with your native message ID. When formatting a response, add in the STRIDE attributes to match the message type you are sending (e.g., one-way response, two-way response, broadcast). If you can identify the type of native message you are sending back to the host you can easily add in the proper attributes.



Without the proper attributes on the STRIDE messages you could have invalid data or undelivered messages.

The benefit to doing a generic transformation algorithm is you would not have to maintain any kind of table or static data structure. By using ranges of message IDs you can identify different types of native messages and translate them to the proper STRIDE message types. For example, if you use message IDs 1 to 100 for one-way commands and IDs 200 to 300 for broadcasts you would be able to determine that an ID of 205 could be converted from your native broadcasts into the proper STRIDE messages by adding the broadcast message type to your ID.

2.2. Using a Remote Message Stub (RMS)

A Remote Message Stub (RMS) thread provides a simple means to integrate the STRIDE messaging environment with your target's native messaging. The RMS thread automatically receives all STRIDE messages that are not registered with the STRIDE Runtime.

You can take advantage of this feature in a number of ways. All your native messages can be left un-registered with the Runtime to allow the RMS thread to receive all these messages. You are then free to translate these received STRIDE messages into your “native” environment. Messages received from your native environment can also be translated into STRIDE messages to be sent back over to the remote platform using the STRIDE Runtime. The key is to take advantage of the STRIDE Runtime to accomplish the message routing that you need to do. You have many different options to choose from when organizing your system. This chapter can help you to understand the choices you need to make.

2.2.1. Remote Message Stub Thread Setup

The RMS creates a STRIDE Transact ID (STID). It then uses this STID along with one of its mail boxes and the RMS Access Class type *srAC_REG_MESSAGES* to identify itself as the RMS by calling *srRegisterAccessClass*. Then, all unregistered messages are routed to the RMS. The RMS can also subscribe to *srSUBSCRIBERS_REMOTE_B_SMID* to receive subscriber information received from the remote platform.

2.2.2. Remote Message Stub Thread Messages

The RMS thread translates the non-registered STRIDE message received into a native message that it will deliver to its native threads. How that translation is accomplished is specific to each target platform.

When a STRIDE command is sent to the RMS module, *palNotify* is called for the NID associated with the registered RMS STID or registered SUID's STID. The *palNotify* implementation and RMS implementation must be tied together. *palNotify* must cause the RMS module to run in a context where it can read its STRIDE mail, translate the SMID into a native ID and then send the native message into the native system. This usually requires the RMS module be a native task, or at least be driven by a native task, awakened through the call to *palNotify*

When a STRIDE command is received several key pieces of information may need to be maintained. This is the case only IF a native response is expected for the native command (translated SMID). If there is an expected response, the STRIDE Message Instance (MINT, which contains the STID and mailbox of the sender) must be kept for routing the response back into the STRIDE environment. Some native messaging systems will provide fields to route this information through the native system, others will not. If the native routing fields are not available, the RMS may be limited. The limitations are dependent on the native messaging rules and the native system's resource availability. The same methods, used to bind the command ID and response ID pairs, can be leveraged, as well, to maintain MINT information. If the MINT information cannot be maintained between native commands and responses, a possible solution is to allow only one STRIDE Host tool to access the device at any one time. In this case the MINT would be the same for every incoming STRIDE command. A single MINT variable could be saved and used for all command/response transactions.

In some rare cases the RMS may need to maintain STRIDE message auxiliary data (a 4 byte token), which should be treated as part of the MINT information, bound to each unique command/response transaction. An example scenario, requiring auxiliary data routing, would be Host development expanded to use native messaging tools on the Host communicating with the device, through the STRIDE environment.

2.2.3. How to Handle Responses

When the RMS thread reads a STRIDE command message it also receives a message instance. This message instance holds information that allows a response to be routed to the sender without knowing anything about the sender. This message instance needs to be included in the call to *srSendRsp* in order for a response message to be routed back to the sender. If the RMS thread is required to send responses to the senders of commands then the message instance needs to be saved and provided in the *srSendRsp* call. This requires the RMS thread to save the message instance and provide it with the correct response. Sending a response with a message instance is supported with the one-way response and the two-way response messages.

If the responses are registered on the target as one-way commands then the RMS thread can send one-way commands as responses.

If the native system sends responses to commands, the RMS module must have a way to hook into the native message system, in order to receive native responses. This may require the RMS module to be a native task, or at least be driven by a native task. See the comments above under "Receiving STRIDE Commands -Sending Native Commands"

Once a response is received from the native system, it must be translated into a corresponding SMID, as well, the originating command's MINT information must be used for routing back the STRIDE message. See the comments above under "Binding Native Commands With Native Responses"

2.2.4. Binding Native Commands with Native Responses

If the native messaging system associates native commands with native responses, the command/response must be bound in the RMS module. The RMS requires this binding in order to send the correct response message (SMID) into the STRIDE message environment. This may be a simple requirement or it may require tables, switch statements, etc. The implementation of the RMS should leverage any native patterns. For example the native command ID may be the same as the response. In this case the STRIDE messaging model provides a two way type message, associating a single SMID with the command and response. In another situation the native system may bind command ID and response ID pairs. If there is a standard algorithm for ID pairs, it can be leveraged for binding SMID command/response pairs, or two way type SMIDs. Native routing information may be leveraged as well. When sending a native command, any routing field(s) routed through the native system may be used to associate the response with the originating command. A worst case scenario would require tables/switches to map a native response ID back to the originating native command ID (and SMID)

2.2.5. Wait Event

Your wait event routine needs to differentiate between your native threads notification to the RMS and the STRIDE Runtime notification to the RMS. Since the RMS is the link between the two messaging systems. You need to decide how this will work. The STRIDE Runtime will notify the RMS when a remote message is received and your native threads will notify the RMS when they need to send a remote message. If your operating system (OS) uses some type of event notification you can define different events for your native threads and the STRIDE Runtime to use.

2.2.6. Subscriptions and Broadcasts

The RMS thread can use the STRIDE Runtime's resources for managing remote subscriptions or it can manage subscriptions itself.

2.2.6.1. How Subscribers are Stored

There are two different ways that SUIDs are managed by the Runtime: index-based SUID table and search-based SUID table. With an index-based SUID table, the entries for SUIDs are preallocated at startup time. The number of entries is fixed and SUIDs are stored in fixed locations in the SUID table. The indexing is done based on the SUID. SUIDs that are greater than the size of the SUID table do not have entries. SUID entries hold the SUID's subscriber list, filter settings, and pointer information.

When the search-based SUID table is used SUID entries are "allocated" only when needed.

For example, if a remote subscription is received and there is no SUID entry for the SUID, then the next available SUID entry is allocated and used for that SUID. If SUID entries were allocated to accommodate SUIDs with pointers but a remote subscribe is received, then a SUID entry will be used for that subscribe entry, leaving one less SUID entry to be used for the pointer SUIDs.

It is important to remember that the first remote subscriber for a particular SUID will cause the Runtime to allocate a SUID entry to save that SUID's subscriber info.

2.2.6.2. Broadcasting

When the RMS thread broadcasts, the Runtime will check for a subscriber list in the SUID entry. If no entry exists or if there are no local subscribers, then the broadcast will be forced across the link. To save resources you can have no SUID entries allocated on the target and then all broadcasts will be forced across the link, where the host platform will keep its subscriber list.

2.2.7. Pointers

STRIDE messages (registered or un-registered) that use pointers must have a SUID entry available for its use. This means that if you are using an index-based SUID table your SUID must not be greater than the size of your SUID table. If you are using a search-based SUID table then you must have an available entry for the SUID with the pointer.

2.2.8. Message Tracing

Automatic tracing of messaging is only available when using STRIDE-registered messages. However, trace points can be substituted in the sending and receiving application threads to allow for tracing. A trace point can be created with the message payload as the trace point payload. Pointers are not handled by trace points, so message payloads with pointers can either call multiple trace points for the message and for each pointer, or create a new payload with the pointers copied in. Any application thread using trace points needs to have an associated STID created.

2.2.9. Remote Message Stub Thread Example

In this example the RMS thread will receive both STRIDE and native messages. There are two types of STRIDE messages received: remote subscriptions and remote messages. When remote subscriptions are received the RMS thread calls its own routine to save the subscribers' information. When remote message are received the RMS thread converts the SMID into a native Message ID and calls the native send routine to route the message to the native threads. When native messages are received any message IDs larger than 0x8000 are broadcast. The RMS thread creates a SMID and does a broadcast. Any other message is sent back to the host as a one-way response with the latest message instance received.

```

/*****
 * RMSThread.c
 *****/

#include "Customer.h"
#include <sr.h>
#include <srutil.h>

#define MAX_MSG_SIZE 10000

void RMSThread(void)
{
    srWORD  wRMS_STID;
    srDWORD dwThreadId;
    srDWORD dwMsgInst;
    srWORD  wEventId;
    srBYTE  MsgBuffer[MAX_MSG_SIZE];
    srWORD  wSize;
    srDWORD dwMsgId;

    dwThreadId = CustomerGetThreadId();

    wResult = srCreateSTID( dwThreadId, "RMSThread", &wRMS_STID );

    /* identify this STID as the RMS */
    wResult = srRegisterAccessClass( srAC_REG_MESSAGES,
                                    wRMS_STID,
                                    srBOX_1,
                                    srTRUE );

```

```
/* subscribe for the remote subscriber's info */
wResult = srSubscribe( wRMS_STID,
                      srSUBSCRIBERS_REMOTE_B_SMID,
                      srTRUE );

/* Message Loop */
for(;;)
{
    /* This is the RTOS or Custom Event Wait routine */
    wEventType = CustomerWaitEvent( &wEventId );

    /* For this example, it is assumed that the srRsxMessageNotify()
       function was implemented to use a signal to indicate
       a SCL Message is pending for the remote message stub thread */

    if( wEventId & STRIDE_MSG_SIGNAL )
    {
        /* Read STRIDE Mail */
        wResult = srRead( wRMS_STID,
                        srBOX_1,
                        sizeof( MsgBuffer ),
                        &dwSMID,
                        &MsgBuffer,
                        &wSize,
                        &dwMsgInst );

        if( dwSMID == srSUBSCRIBERS_REMOTE_B_SMID )
        {
            /* Add subscriber to native subscriber list */
            CustomerAddSubscriber( (uint8*)&MsgBuffer.SubInfo );
        }
        else
        {
            /* translate into native environment */
            srDWORD dwTargetMsgId = dwSMID & srSMID_SUID_MASK;

            /* now send to native thread */
            CustomerSendMsg( dwTargetMsgId,
                            (uint8*)&MsgBuffer.WriteDataCmd,
                            wSize );

            break;
        }
    }
    else if( wEventType == CUSTOMER_SIGNAL_EVENT )
    {
        /* read mail from native threads */
        wResult = CustomerRead( &MsgBuffer ),
                    &dwMsgId,
                    &wSize );

        if( dwMsgId & CUSTOMER_BROADCAST_MSG )
        {
            /* broadcasts */
            dwMsgId = dwMsgId & srSMID_SUID_MASK;
            srBroadcast( wRMS_STID ,
                        srDWORD(dwMsgId + srMT_BRD + srST_RSP_VAL),
```


2.3.1. Remote Message Proxy Routing

The RMP module must have a way to hook into the native message system in order to receive native commands. This usually requires some sort of native registration for the specific messages the RMP will route out of the native system. Once the RMP is registered for the specific native message, the RMP must be hooked into the native system in order to receive the commands from other native clients. This may require the RMP module to be a native task, or at least be driven by a native task. Once the native message is received it must be translated into a SMID and sent into STRIDE messaging system.

2.3.2. Binding Native Commands with Native Responses

If the native messaging system associates native commands with native responses, the command/response may need to be bound in the RMP module. The RMP requires this binding in order to send the correct command message (SMID) into the STRIDE message environment. This may be a simple requirement or it may require tables, switch statements, etc. The implementation of the RMP should leverage any native patterns. For example the native command ID may be the same as the response. In this case the STRIDE messaging model provides a two-way-type message, associating a single SMID with the command and response. In another situation the native system may bind command ID and response ID pairs. If there is a standard algorithm for ID pairs, it can be leveraged for binding SMID command/response pairs, or two way type SMIDs.

The STRIDE messaging model provides a method to bind the command with the response through a routing mechanism, auxiliary data (a 4 byte token) routed through the STRIDE system. When a native command is received by the RMP, it can store any information, required for routing the native response, in the auxiliary data of the associated STRIDE command being sent out. When the RMP receives the corresponding STRIDE response, it can extract the auxiliary data and retrieve any information required for routing back the native response.

2.3.3. Receiving STRIDE Responses – Sending Native Responses

When a STRIDE response is sent to the RMP module, *palNotify* is called for the NID associated with the original STID used to send the originating command. The *palNotify* implementation and RMP implementation must be tied together. *palNotify* must cause the RMP module to run in a context where it can read its STRIDE mail, translate the SMID into a native ID and then send the native response into the native system. This usually requires the RMP module be a native task, or at least be driven by a native task, awakened through the call to *palNotify*.

3. Runtime API Services

API Services are divided into the following functionalities:

Setup and Shutdown – page 46

Messaging – page 52

Pointers – page 79

Tracing – page 96

Query – page 105

Access Class (Remote Messaging) Routines – page 113

I-block – page 114

Runtime Thread Entry and Exit Point – page 116

Host Override Routines – page 121

Connecting – page 124

Database Loading Routines – page 133

Trace Buffers – page 139

Trace Filtering – page 142



These services use the *sr.h* header file.

3.1. Setup and Shutdown

The following routines are utilities used to enable and support the sending and reading of messages.

- `srInit()`
- `srUninit()`
- `srCreateSTID()`
- `srDeleteSTID()`

srInit()

Initialize STRIDE Runtime

Prototype

```
srWORD srInit( void );
```

Description

The srInit() routine initializes STRIDE Runtime internal data structures, mutexes, sets up the connection parameters, and registers PAL routines.



This routine must be called before any STRIDE Runtime routines are called.

Return Value	Description
srOK	Success
srERR_INIT_FAILED	STRIDE Runtime was unable to initialize

Example

```
#include <sr.h> // contains prototypes and defines

srWORD wResult;

/*****
 * Initializing example:
 *****/

wResult = srInit();
```

srUninit()

Uninitialize STRIDE Runtime

Prototype

```
srWORD srUninit( void );
```

Description

The srUninit() routine uninitializes STRIDE Runtime internal data structures, mutexes, connection parameters, and unregisters PAL routines.



This routine must be called before the STRIDE Runtime is shutdown to cleanup resources.

Return Value	Description
srOK	Success
srERR_INIT_FAILED	STRIDE Runtime was unable to initialize

Example

```
#include <sr.h> // contains prototypes and defines

srWORD wResult;

/*****
 * Uninitializing example:
 *****/

wResult = srUninit();
```

srCreateSTID()

Create STRIDE Transact Identifier

Prototype

```
srWORD srCreateSTID( srDWORD dwNID,
                    srCHAR *szName,
                    srWORD *pwSTID,
                    srBOOL bNew );
```

Description

The srCreateSTID() routine is used to allocate the resources required to send and receive messages and to log the use of trace points or trace stings. This routine requires the Notification Identifier (NID), the STID name, and the bNew parameter.

If the NID is specified, it is used when the STRIDE Runtime notifies the system that there is a pending message for a STID's box. The NID is simply a value passed to the Runtime and saved, then passed back to the PAL Notify routine when a notify needs to take place. The Runtime knows nothing about the NID. The NID is held in order to be passed back when the Notify routine runs. The Pal Notify routine uses the NID to correctly notify the specific STID. The details of the NID are specific to the implementation of the PAL.

If a value of srNID_NONE is passed in as the NID, no NID will be used for this STID and no notification will take place.

An STID alphanumeric name (szName) can be between 0 and 15 characters. A null (zero-length) STID name can be used if it is desired that no tracing occur for this transactor. When a zero-length name is assigned it is not possible to trace on interface transactions originating from or terminating at the transactor (STID).

The bNew parameter tells the Runtime if a new STID is being requested or if the value in the pwSTID parameter is the STID requested. This can be used when STIDs need to be pre-determined regardless of the order the call to srCreateSTID is made. If the requested STID is already in use, the routine returns an error and the original STID is unchanged. If a new STID is requested the new value is returned in the pwSTID parameter.

Parameters	Type	Description
<i>dwNID</i>	Input	The Application Notification Identifier
<i>szName</i>	Input	Character pointer to a NULL terminated string
<i>pwSTID</i>	Input/Output	The STRIDE Transact Identifier
<i>bNew</i>	Input	Request new STID
		srTRUE = Create new STID dynamically

srFALSE = Use STID passed in parameter list

Return Value	Description
<i>srOK</i>	Success
<i>srERR_STID_ALLOC</i>	STID allocation failed
<i>srERR_STID_USED</i>	STID in use

Example

```
#include <sr.h> // contains prototypes and defines

srWORD  wResult;
srDWORD dwMyNID;
srWORD  wMySTID;

dwMyNID = 42; // Get ID for platform specific routine

wResult = srCreateSTID( dwMyNID, "MySTIDName", &wMySTID, srTRUE );
```

srDeleteSTID()

Release resources allocated to STID

Prototype

```
srWORD srDeleteSTID( srWORD wSTID );
```

Description

The `srDeleteSTID()` routine is used to free STRIDE Runtime resources previously allocated for an STID. The resources allocated by the following routines will be returned:

- `srRegister(..)` – registering messages
- `srSubscribe(..)` – subscribing to messages

Parameters	Type	Description
<code>wSTID</code>	Input	STRIDE Transact ID from the <code>srCreateSTID()</code> call

Return Value	Description
<code>srOK</code>	Success
<code>srERR_STID_INVALID</code>	STID is invalid

Example

```
#include <sr.h> // contains prototypes and defines

srWORD wResult;
srDWORD dwMyNID;
srWORD wMySTID;

dwMyNID = 42; // Get ID for platform specific routine

wResult = srCreateSTID( dwMyNID, "MySTIDName" ,&wMySTID, srTRUE );

...

wResult = srDeleteSTID( wMySTID );
```

3.2. Messaging

The following routines are used to send and read messages.

- `srRegister()`
- `srRegisterAccessClass()`
- `srRead()`
- `srReadComplete()`
- `srSendCmd()`
- `srSendRsp()`
- `srBroadcast()`
- `srSubscribe()`
- `srSetAuxData()`
- `srGetAuxData()`

srRegister()

Register a one-way or two-way message

Prototype

```
srWORD srRegister( srWORD    wSTID,
                  srBOX_e   eBox,
                  srDWORD   dwSMID,
                  srBOOL    bOn );
```

Description

The `srRegister()` routine is used to register a one-way or two-way message. An Owner must register these message types before they are available for public use. When registering a message, the Owner must pass in the associated STID and Message Box ID, the unique *STRIDE Message ID (SMID)*, and an indicator to turn on or off the registration.

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact Identifier
<i>eBox</i>	Input	The associated mailbox for the message
<i>dwSMID</i>	Input	The unique SMID
<i>bOn</i>	Input	Indicates if registration is On or Off srTRUE = On srFALSE = Off

Return Value	Description
<i>srOK</i>	Success
<i>srERR_STID_INVALID</i>	STID has not been created
<i>srERR_SMID_ATTR</i>	SMID attributes are incorrect
<i>srERR_SUID_RANGE</i>	SUID is not in the valid range
<i>srERR_REG_SET</i>	Registration already set
<i>srERR_REG_STORAGE_FULL</i>	Registration storage full

Example

```
#include <sr.h>    // contains API definitions

#define MYSMID_1 1 + ( srMT_ONE | srST_CMD_VAL )
#define MYSMID_2 2 + ( srMT_TWO | srST_CMD_VAL | srST_RSP_VAL )

srWORD wResult;

// Assumes STID already created

wResult = srRegister( wMySTID, srBOX_1, MYSMID_1, srTRUE );
wResult = srRegister( wMySTID, srBOX_1, MYSMID_2, srTRUE );
```


srRegisterAccessClass()

Registers the Access Class type for message routing

Prototype

```
srWORD srRegisterAccessClass ( srAccessClass_e  eAC,
                               srWORD           wSTID,
                               srBOX_e         eBox,
                               srBOOL          bOn );
```

Description

The *srRegisterAccessClass()* routine registers the Access Class (AC) type with the STRIDE Runtime, and associates the STID and mailbox with the Access Class. When set as an Access Class, function calls and messages with no registered owner will be routed to the registered STID depending on the Access Class type.

AC types:

- *srAC_REG_MESSAGES* – for the Remote Messaging
- *srAC_REG_FUNCTIONS* – for Intercept Modules

Parameters	Type	Description
<i>eAC</i>	Input	Access Class of type <i>srAccessClass_e</i> , defined as: <pre>typedef enum { srAC_REG_MESSAGES = 0, srAC_REG_FUNCTIONS = 1 } srAccessClass_e;</pre> Values are <i>srAC_REG_MESSAGES</i> or <i>srAC_REG_FUNCTIONS</i>
<i>wSTID</i>	Input	The associated STRIDE Transact Identifier for the Access Class
<i>eBox</i>	Input	The associated mailbox for the Access Class
<i>bOn</i>	Input	Set or unset registration srTRUE = On srFALSE = Off

Return Value	Description
<i>srOK</i>	Success
<i>srErr</i>	Failure

Example

Access Class IM Example

```
// srWORD wSTID -- assumes STID has already been created

srWORD    wResult;
wResult = srRegisterAccessClass (srAC_REG_FUNCTIONS,
                                wSTID,
                                srBOX_1,
                                srTRUE);
```

Remote Messaging Example (RMS & RMP)

```
// srWORD wSTID -- assumes STID has already been created

srWORD    wResult;
wResult = srRegisterAccessClass (srAC_REG_MESSAGES,
                                wSTID,
                                srBOX_1,
                                srTRUE);
```

srRead()

Read command or response

Prototype

```

srWORD srRead( srWORD    wSTID,
               srBOX_e   eBox,
               srWORD    wMaxRead,
               srDWORD   *pdwSMID,
               srBYTE    *pyBuffer,
               srWORD    *pwSize,
               srDWORD   *pdwMsgInst );

```

Description

The `srRead()` routine is used to read a command from a User or a response from an Owner. The STID, Box ID, and maximum number of bytes to read are input parameters. The SMID, the payload (or address of the payload), size of the payload, and message instance are output parameters. If no messages are pending for the specified box the routine will return `srERR_QUEUE_EMPTY` and zero (0) will be returned for the SMID.

The value written to the `pyBuffer` depends on the attributes defined within the SMID being read. If the SMID payload is defined by value then the entire contents of the payload is written to the address in the parameter list. If the payload is defined as a pointer then the pointer value is written to the payload parameter. The size parameter reflects either the size of the value payload (when the payload is by value) or the size of the data that the pointer points to (if the payload is by pointer).



The Message Instance Identifier represents a unique transaction between an Owner and User. This identifier contains important routing and resource information used by the STRIDE Runtime. The identifier is required input for both the `srSendRsp(..)` routine and the `srReadComplete(..)` routine.

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact ID from <code>srCreateSTID()</code> call
<i>eBox</i>	Input	Box from which to read
<i>wMaxRead</i>	Input	The max number of bytes allowable to read
<i>pdwSMID</i>	Output	The SMID read; 0 if nothing in box
<i>pyBuffer</i>	Output	Input data; address of or payload
<i>pwSize</i>	Output	Size of the payload data

pdwMsgInst Output The Message Instance ID

Return Value**Description**

srOK

Success

srERR_STID_INVALID

STID has not been created

srERR_QUEUE_EMPTY

Message queue empty

srERR_READ_SIZE

Read buffer is too small for the message payload

Example

```
Owner.h

#include <sr.h>            // Contains SMID Attributes

#define DOIT_NOW_SMID 10 + ( srMT_ONE | srST_CMD_VAL )

typedef struct {        // Command payload definition
    srDWORD dwData1;
} Doit_Now_Cmd;
```

```
Owner.c

#include <sr.h>        // Contains prototypes
#include <owner.h>

#define BIGENOUGH 100

srWORD        wResult;
srDWORD       dwMySMID;
srBYTE        InputBuf[BIGENOUGH];
srWORD        wMsgSize;
srDWORD       dwMsgInst;

// Assumes STID already setup

wResult = srRegister( wMySTID, srBOX_1, DOIT_NOW_SMID, srTRUE );
```

```
// Wait for new message

CustomerWait (...);
...

// Reading a message

wResult = srRead( wMySTID,
                  srBOX_1,
                  BIGENOUGH,
                  &dwMySMID,
                  &InputBuf,
                  &wMsgSize,
                  &dwMsgInst );

if( dwMySMID == DOIT_NOW_SMID )
{
    ...
}
```

srReadComplete()

Prototype

```
srWORD srReadComplete( srWORD    wSTID,
                       srDWORD    dwMsgInst );
```

Description

The `srReadComplete()` routine is used to release any temporary memory allocated to support marshaling of *private memory* across platform boundaries.

If no resources have been allocated, this routine returns immediately. The typical location for the routine is at the end of the message loop used for reading messages.



A payload or pointer that is allocated from a system pool and is required to be freed by the reader is considered *pool memory*. A payload or pointer that is not expected to be freed by the reader is always considered private memory regardless of where the memory has been allocated

The SMID attributes are used to define a message's pointer memory usage (private or pool). The Pointer Setup routine `srEPtrSetup` defines a pointer's memory usage. If either the SMID or the pointer uses private memory `srReadComplete` must be called.



When using a *private* pointer within a two-way command payload which uses an OUT or INOUT directional attribute, the Owner is required to respond to the User (calling `srSendRsp`) **before** calling this routine; otherwise `srReadComplete` can be called after the private memory is no longer needed.

Parameters	Type	Description
<code>wSTID</code>	Input	STRIDE Transact ID from the <code>srCreateSTID()</code> call
<code>dwMsgInst</code>	Input	The sender's Message Instance Identifier
Return Value		Description
<code>srOK</code>		Success
<code>srERR_STID_INVALID</code>		STID is invalid
<code>srERR_PTR_INVALID</code>		Pointer handle invalid

Example

```
#include <sr.h> // contains prototypes and defines

#define BIGENOUGH 100

srDWORD dwMySMID;
srBYTE InputBuf[BIGENOUGH];
srWORD wMsgSize;
srDWORD dwMsgInst;

for (;;)
{
    // Wait for new message

    CustomerWait(...);

    // Reading a message Payload

    // Assuming STID already created

    wResult = srRead( wMySTID,
                     srBOX_1,
                     BIGENOUGH,
                     &dwMySMID,
                     &InputBuf,
                     &wMsgSize,
                     &dwMsgInst );

    // Make sure to respond to any 2-ways using Eptr with OUT/INOUT first

    ...

    wResult = srReadComplete( wMySTID, dwMsgInst );
}
```

srSendCmd()

Send a command

Prototype

```

srWORD srSendCmd( srWORD    wSTID
                  srBOX_e   eRspBox,
                  srDWORD   dwSMID,
                  srBYTE    *pyPayload,
                  srWORD    wSize );

```

Description

The `srSendCmd()` routine is used to send a command to an Owner. The STID, Response Box ID, SMID, a pointer to the message payload, and the size of the payload are required input parameters. A User always sets the Box ID required for the response when issuing a two-way message. The Box ID (`eRspBox` parameter) is used to determine which box the response is sent back to. Only a user sending a two way command needs to assign a valid Box ID in the parameter. The Response Box ID is **not** applicable for one-way messages.

How the `pyPayload` parameter is used depends on how the SMID is defined. If the SMID is defined with a “command by value” attribute, the data is read directly, starting from the address in the `pyPayload` parameter. If the SMID is defined with a “command by pointer” (default) then the `pyPayload` parameter is an address of a pointer which will be used to reference the payload data. In the case of “command by pointer” if the SMID is defined with a “pool pointer command” (default) attribute, the SCM requires that the memory is allocated from the common pool being used by the system.



The `wSize` parameter always reflects the size of the payload data. If the SMID command is “by pointer” then the size reflects the amount of data the pointer actually points to, not the size of the pointer.

Parameters	Type	Description
<code>wSTID</code>	Input	STRIDE Transact ID from the <code>srCreateSTID()</code> call
<code>eRspBox</code>	Input	The Response Box Identifier
<code>dwSMID</code>	Input	The unique SMID
<code>pyPayload</code>	Input	Address (pointer) of the message payload
<code>wSize</code>	Input	The size of the payload

Return Value	Description
<i>srOK</i>	Success
<i>srERR_STID_INVALID</i>	STID is invalid
<i>srERR_STID_INACTIVE</i>	STID is inactive
<i>srERR_REG_NONE</i>	No local registration
<i>srERR_PAL_MEM_ALLOC</i>	PAL memory allocation failed
<i>srERR_PTR_ALLOC</i>	Pointer allocation failed
<i>srERR_SEND_PRIV</i>	Attempt to send private memory without pointers
<i>srERR_PTR_POOL</i>	Expected pool pointer memory
<i>srERR_PTR_OVERWRITE</i>	Payload pointer does not match pointer entry
<i>srERR_QUEUE_FULL</i>	Message queue full
<i>srERR_PAL_NOTIFY_SYS</i>	PAL notification to STRIDE Runtime failed
<i>srERR_PAL_NOTIFY_USER</i>	PAL notification to user failed

Example

```
Owner.h

#include <sr.h> // Contains SMID Attributes

#define DOIT_NOW_SMID 20 + ( srMT_ONE | srST_CMD_VAL )

typedef struct {
    srDWORD dwData1;
} Doit_Now_Cmd;
```

```
User.c

#include <sr.h> // contains prototypes and defines
#include <owner.h>

Doit_Now_Cmd MyDoitNow;
srWORD wResult;

MyDoitNow.dwData1=42;

// Assumes STID already created

wResult = srSendCmd( wMySTID,
                    srBOX_1, //Not used in One Way
                    DOIT_NOW_SMID,
                    (srBYTE *) &MyDoitNow,
                    sizeof(MyDoitNow) );
```

srSendRsp()

Send response

Prototype

```
srWORD srSendRsp( srWORD    wSTID,
                  srDWORD   dwMsgInst,
                  srDWORD   dwSMID,
                  srBYTE    *pyPayload,
                  srWORD    wSize );
```

Description

The `srSendRsp()` routine is used to send a response message to a User. The `STID`, Message Instance, `SMID`, a pointer to the payload, and the size of the payload are required input parameters. The Owner uses the message instance received from `srRead(..)` when responding to a two-way message. This message instance contains routing information that the STRIDE Runtime uses to correctly route the response.

How the `pyPayload` parameter is used depends on how the `SMID` is defined. If the `SMID` is defined with a “response by value” attribute then the data is read directly, starting from the address in the `pyPayload` parameter. If the `SMID` is defined with a “response by pointer” (default) then the `pyPayload` parameter is an address of a pointer which will be used to reference the payload data. In the case of “response by pointer” if the `SMID` is defined with a “pool pointer response” (default) attribute, the SCM requires that the memory is allocated from the common pool being used by the system.



The `wSize` parameter always reflects the size of the payload data. If the `SMID` response is “by pointer” then the size reflects the amount of data the pointer actually points to, not the size of the pointer.

Parameters	Type	Description
<code>wSTID</code>	Input	STRIDE Transact ID from the <code>srCreateSTID()</code> call
<code>dwMsgInst</code>	Input	The sender's Message Instance Identifier
<code>dwSMID</code>	Input	The unique <code>SMID</code>
<code>pyPayload</code>	Input	Address (pointer) of the message payload
<code>wSize</code>	Input	The size of the payload

Return Value	Description
<i>srOK</i>	Success
<i>srERR_STID_INVALID</i>	STID is not valid
<i>srERR_STID_INACTIVE</i>	Calling STID routing to inactive STID
<i>srERR_RMT_FAIL</i>	Attempt to route remote failed
<i>srERR_PAL_MEM_ALLOC</i>	PAL memory allocation failed
<i>srERR_PTR_ALLOC</i>	Pointer allocation failed
<i>srERR_SEND_PRIV</i>	Attempt to send private memory without pointers
<i>srERR_PTR_POOL</i>	Expected pool pointer memory
<i>srERR_PTR_OVERWRITE</i>	Payload pointer does not match pointer entry
<i>srERR_PTR_DUPLICATE</i>	Duplicate pointer set up
<i>srERR_PTR_LOCKED</i>	Pointer in lock state
<i>srERR_QUEUE_FULL</i>	Message queue full
<i>srERR_PAL_NOTIFY_SYS</i>	PAL notification to STRIDE Runtime failed
<i>srERR_PAL_NOTIFY_USER</i>	PAL notification to user failed

Example

```
Owner.h

#include <sr.h>          // Contains SMID Attributes

#define DOIT_LATER_SMID 30 + ( srMT_TWO | srST_CMD_VAL | srST_RSP_VAL )

typedef struct {        // Command payload definition
    srDWORD  dwData1;
} Doit_Later_Cmd;

typedef struct {        // Response payload definition
    srDWORD  dwData2;
} Doit_Later_Rsp;
```

```
Owner.c

#include <sr.h>      // Contains prototypes
#include <owner.h>

#define  BIGENOUGH 100

srWORD    wResult;
srDWORD   dwMySMID;
srBYTE    InputBuf[BIGENOUGH];
srWORD    wMsgSize;
srDWORD   dwMsgInst;

Doit_Later_Cmd * pMyDoitLaterCmd;
Doit_Later_Rsp  MyDoitLaterRsp;

// Assumes STID already created

wResult = srRegister( wMySTID, srBOX_1, DOIT_LATER_SMID, srTRUE );

...

// Wait for new message

CustomerWait(...);

...

// Reading a message

wResult = srRead( wMySTID,
                 srBOX_1,
                 BIGENOUGH,
                 &dwMySMID,
                 &InputBuf,
                 &wMsgSize,
                 &dwMsgInst );

// Sending back a response

if( dwMySMID == DOIT_LATER_SMID )
{
    pMyDoitLaterCmd = ( Doit_Later_Cmd ) InputBuf;

    MyDoitLaterRsp.dwData2 = pMyDoitLaterCmd->dwData1;

    wResult = srSendRsp( wMySTID,
                          dwMsgInst,
                          DOIT_LATER_SMID,
                          &MyDoitLaterRsp,
```

```
        sizeof(MyDoitLaterRsp) );  
    }
```

srBroadcast()

Broadcast response

Prototype

```
srWORD srBroadcast( srWORD   wSTID,
                   srDWORD  dwSMID,
                   srBYTE   *pyPayload,
                   srWORD   wSize );
```

Description

The `srBroadcast()` routine is used to broadcast a response to one or more *subscribers*. Only the Broadcast message type is applicable for this routine. The STID, SMID, pointer to the payload, and the size of the payload are required input parameters. Each subscriber receives its own copy of the message or pointer, depending on the attributes of the payload.

If there are multiple subscribers on a remote platform only one copy of the response is transmitted across the link. The Runtime on the remote platform is responsible for making the necessary copies of the payload for its subscribers. This is done to minimize traffic across the link. If there are no subscribers then the routine simply returns, cleaning up any pool memory that was allocated for the broadcast.

How the *pyPayload* parameter is used depends on how the SMID is defined. If the SMID is defined with a “response by value” attribute then the data is read directly, starting from the address in the *pyPayload* parameter. If the SMID is defined with a “response by pointer” (default) then the *pyPayload* parameter is an address of a pointer which will be used to reference the payload data. In the case of “response by pointer” if the SMID is defined with a “pool pointer response” (default) attribute, the SCM requires that the memory is allocated from the common pool being used by the system.

The *wSize* parameter always reflects the size of the payload data. If the SMID command is “by pointer” then the size reflects the amount of data the pointer actually points to, not the size of the pointer.



Each subscriber receives its own unique payload. For “by pointer” payloads that use *pool* memory, each subscriber is responsible for freeing its own payload once it has been read and processed. In addition, all pointers contained in the payload that have been allocated from the pool, are required to be freed. If the payload is using *private* memory, each subscriber is required to return the message back to the Runtime with a call to *srReadComplete*.

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact ID from the <i>srCreateSTID()</i> call
<i>dwSMID</i>	Input	The unique SMID
<i>pyPayload</i>	Input	Address (pointer) of the message payload
<i>wSize</i>	Input	The size of the payload

Return Value	Description
<i>srOK</i>	Success
<i>srERR_STID_INVALID</i>	STID is not valid
<i>srERR_STID_INACTIVE</i>	Calling STID routing to inactive STID
<i>srERR_RMT_FAIL</i>	Attempt to route remote failed
<i>srERR_PAL_MEM_ALLOC</i>	PAL memory allocation failed
<i>srERR_PTR_ALLOC</i>	Pointer allocation failed
<i>srERR_SEND_PRIV</i>	Attempt to send private memory without pointers
<i>srERR_PTR_POOL</i>	Expected pool pointer memory
<i>srERR_PTR_OVERWRITE</i>	Payload pointer does not match pointer entry
<i>srERR_QUEUE_FULL</i>	Message queue full
<i>srERR_PAL_NOTIFY_SYS</i>	PAL notification to STRIDE Runtime failed
<i>srERR_PAL_NOTIFY_USER</i>	PAL notification to user failed

Example

```
Owner.h

#include <sr.h>    // Contains SMID Attributes

#define DOIT_GOTIT_SMID 40 + ( srMT_BCAST | srST_RSP_VAL )

typedef struct {
    srDWORD dwData3;
} DoitGotit_Rsp;
```

```
OwnerCode.c

#include <sr.h>    // contains prototypes and defines
#include <owner.h>

srWORD          wResult;
DoitGotit_Rsp   MyDoitGotit_Rsp;

// Assumes STID already created

wResult = srRegister( wMySTID, srBOX_1, DOIT_GOTIT_SMID, srTRUE );

MyDoitGotit_Rsp.dwData3 = 42;

wResult = srBroadcast( wMySTID,
                       DOIT_GOTIT_SMID,
                       (srBYTE *) &MyDoitGotit_Rsp,
                       sizeof(MyDoitGotit_Rsp) );
```

srSubscribe()

Subscribe to broadcast message

Prototype

```

srWORD srSubscribe( srWORD    wSTID,
                   srBOX_e   eBox,
                   srDWORD   dwSMID,
                   srBOOL    bOn );

```

Description

The srSubscribe() routine is used to subscribe to a Broadcast message type. Only a Broadcast message type can be subscribed to. The STID, SMID, Box ID, and the current state of the subscription are required input parameters.



If a User requires the contents of a payload immediately, the *srSendCmd()* routine can be used with a different message ID and the same payload. This is often required for “state” based information that is only broadcasted when the data content changes. For this feature to be enabled, the Owner must register the auxiliary message.

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact Identifier
<i>eBox</i>	Input	The Box ID to send message to
<i>dwSMID</i>	Input	The unique SMID
<i>bOn</i>	Input	Indicates if Subscription is On or Off srTRUE = On srFALSE = Off

Return Value	Description
<i>srOK</i>	Success
<i>srERR_STID_INVALID</i>	STID is invalid
<i>srERR_SMID_ATTR</i>	SMID attributes are incorrect
<i>srERR_SUID_RANGE</i>	SUID is not in valid range

srERR_SUB_ALLOC

Subscription allocation failed

Example

UserCode.c

```
#include <sr.h>          // contains prototypes and defines
#include <owner.h>

#define  BIGENOUGH 100

srWORD   wResult;
srDWORD  dwMySMID;
srBYTE   InputBuf[BIGENOUGH];
srWORD   wMsgSize;
srDWORD  dwMsgInst;

// Assuming STID already created
wResult = srSubscribe( wMySTID,DOIT_GOTIT_SMID, srBOX_1, srTRUE );

// Example of requesting a immediate response payload
wResult = srSendCmd( wMySTID,
                    srBOX_1,
                    DOIT_GOTIT_SMID,
                    (srBYTE *) srNULL,
                    srNULL );

// Wait for new message
CustomerWait(...);

...

// Reading a message
wResult = srRead( wMySTID,
                  srBOX_1,
                  BIGENOUGH,
                  &dwMySMID,
                  &InputBuf,
                  &wMsgSize,
                  &dwMsgInst );
```

```
if( dwMySMID == DOIT_GOTIT_SMID ) {  
    ...  
}
```

srSetAuxData()

Set Auxiliary Data

Prototype

```
srWORD srSetAuxData( srWORD    wSTID,
                    srDWORD   dwAuxData );
```

Description

The `srSetAuxData()` routine is used to set the value of the auxiliary data for a specific STID. This data is persisted for as long as an STID is valid. Each time an STID is used to send a message, the current value of the auxiliary data will be sent along with the message.

The same auxiliary data will be sent with every message until it is changed by calling `srSetAuxData`. The auxiliary data value may be set once and left for the entire duration an STID is valid or it can be changed with every message sent.



Also, it is important to remember that the auxiliary data must be set before calling `srSendCmd` or `srSendRsp`. The value set will be sent along with your message to the reader's message queue.

Parameters	Type	Description
<code>wSTID</code>	Input	STRIDE Transact Identifier
<code>dwAuxData</code>	Input	The auxiliary data to set
Return Value	Description	
<code>srOK</code>	Success	
<code>srERR_STID_INVALID</code>	STID is invalid	

Example

```
UserCode.c

#include <sr.h>           // contains prototypes and defines
#include <owner.h>
```

```
srWORD    wResult;  
srWORD    wMySTID;  
srDWORD   dwMyAuxData;  
  
// Example of setting the Auxiliary data and sending message  
dwMyAuxData = 0x00A01000;  
  
wResult = srSetAuxData( wMySTID, dwMyAuxData );  
  
wResult = srSendCmd( wMySTID,  
                    srBOX_1,  
                    DOIT_GOTIT_SMID,  
                    (srBYTE *) srNULL,  
                    srNULL );
```

srGetAuxData()

Get Auxiliary Data

Prototype

```
srWORD srGetAuxData( srWORD    wSTID,
                    srDWORD   *pdwAuxData );
```

Description

The `srGetAuxData()` routine is used to get the auxiliary data received when reading a STRIDE message. This data is persisted only until the next message is read. Each time a message is read, the auxiliary data is saved and available for reading. If another message is read, then the previous auxiliary data will be overwritten.



The auxiliary data is stored in the message queue along with the message and is not available to be read until the message has been read with the `srRead` command. Once the message has been read the auxiliary data is available to be read. Also, once the message has been read the auxiliary data will not be overwritten until the next message is read.

Parameters	Type	Description
<code>wSTID</code>	Input	STRIDE Transact Identifier
<code>pdwAuxData</code>	Output	The auxilliary data read

Return Value	Description
<code>srOK</code>	Success
<code>srERR_STID_INVALID</code>	STID is invalid

Example

```
UserCode.c

#include <sr.h>          // contains prototypes and defines
#include <owner.h>

#define BIGENOUGH 100
```

```
srWORD    wResult;  
srDWORD   dwMySMID;  
srBYTE    InputBuf[BIGENOUGH];  
srWORD    wMsgSize;  
srDWORD   dwMsgInst;  
srDWORD   dwMyAuxData;
```

```
// Example of reading Auxiliary data  
  
// Wait for new message  
CustomerWait(...);  
  
...  
  
// Reading a message  
  
wResult = srRead( wMySTID,  
                  srBOX_1,  
                  BIGENOUGH,  
                  &dwMySMID,  
                  &InputBuf,  
                  &wMsgSize,  
                  &dwMsgInst );  
  
// now read the aux data for this message  
wResult = srGetAuxData( wMySTID, &dwMyAuxData );  
  
}
```


3.3. Pointers

The following routines are used by applications that use pointers.

- `srPtrSetup()`
- `srPtrSetupChild()`
- `srPtrTeardown()`
- `srPtrGetHandle()`
- `srPtrSize()`
- `srPtrCreateCmdInst()`
- `srPtrCreateRspInst()`
- `srPtrDeleteInst()`

srPtrSetup()

Prototype


```

srWORD  srPtrSetup( srWORD          wSTID,
                   srDWORD         dwSMID,
                   srWORD          wOffset,
                   srWORD          *pwOffsetTable,
                   srWORD          wSize,
                   srMsgDir e      eMsgDir,
                   srPtrDir e      ePtrDir,
                   srPtrUsage e    ePtrUsage,
                   srWORD          *pHandle );
    
```

Description

The *srPtrSetup* routine is used to set up a pointer field within a payload. STRIDE Runtime requires the setup of the pointer field information, enabling the Runtime to route the pointer locally and marshal the pointer across platforms. The setup routine is required to be called once for each unique pointer field contained in a payload. A pointer can be set up once and used for the lifetime of the STID used in the setup, or it can be set up and torn down for every message transaction.

For command return pointers such as *srPTRDIR_CMD_RET*, a onetime setup can be achieved by using the field *pwOffsetTable*. This table is an array of values used by the Runtime to route and marshal command return pointers. The first value is the offset depth. The following values are the corresponding offsets, starting from to topmost parent pointer offset in the command payload to the actual offset of the pointer in the parent pointer payload. When setting up a pointer in this mode, the parameter *wOffset* should equal the total number of offsets in the table, or the offset depth. This value must match the first value in the table.

	<p>Each user sending a command payload containing a pointer is required to call the <i>srPtrSetup</i> routine. The owner of a response payload containing a pointer is required to call the <i>srPtrSetup</i> routine.</p>
---	--

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact Identifier from the <i>srCreateSTID()</i> call
<i>dwSMID</i>	Input	The unique SMID
<i>wOffset</i>	Input	The pointer field offset within the payload
<i>pwOffsetTable</i>	Input	The pointer field offset table, containing offset

		depth and offsets into parent payloads
<i>eMsgDir</i>	Input	The message direction: wMSGDIR_COMMAND = 0 wMSGDIR_RESPONSE = 1
<i>ePtrDir</i>	Input	The direction attribute of the pointer: srPTRDIR_CMD_IN = 0 srPTRDIR_CMD_OUT = 1 srPTRDIR_CMD_INOUT = 2 srPTRDIR_CMD_RET = 3 srPTRDIR_RSP_RET = 3
<i>ePtrUsage</i>	Input	The pointer usage attribute: srPTR_USAGE_PRIVATE = 0 srPTR_USAGE_POOL = 1
<i>pwHandle</i>	Output	The unique handle of the pointer

Return Value	Description
<i>srOK</i>	Success
<i>srERR_STID_INVALID</i>	STID is invalid
<i>srERR_SUID_RANGE</i>	SUID is not in the valid range
<i>srERR_SMID_ATTR</i>	SMID attributes are incorrect
<i>srERR_PTR_MSG_DIR</i>	Pointer message direction invalid
<i>srERR_PTR_DIR</i>	Pointer direction invalid
<i>srERR_PTR_USAGE</i>	Pointer memory usage invalid
<i>srERR_PTR_DUPLICATE</i>	Duplicate pointer set up
<i>srERR_PTR_OFFSET</i>	Pointer offset invalid
<i>srERR_PTR_ALLOC</i>	Pointer allocation failed

Example

```
Interface.h
#include <sr.h>

#define INTERFACE SMID 50 + (srMT TWO | srST CMD VAL | srST RSP VAL )

typedef struct
{
    srDWORD *pdwField;
} InterfaceCmd;

IntefaceUser.c
#include <sr.h>
#include <Interface.h>
srWORD    wSTID;
srWORD    wHandle;
srWORD    wResult;

...

wResult = srPtrSetup( wSTID,
                      INTERFACE SMID,
                      GET_OFFSET(InterfaceCmd,pdwField),
                      srNULL,
                      sizeof(srDWORD),
                      srMSGDIR_COMMAND,
                      srPTRDIR_CMD IN,
                      srPTR USAGE POOL,
                      &wHandle );

...

wResult = srPtrTeardown( wSTID,
                          INTERFACE SMID,
                          & wHandle );

...
```

srPtrSetupChild()

Prototype

```
srWORD srPtrSetupChild( srWORD          wSTID,
                        srWORD          wParentHandle,
                        srWORD          wOffset,
                        srWORD          wSize,
                        srPtrDir_e      ePtrDir,
                        srPtrUsage_e    ePtrUsage,
                        srWORD          *pwHandle );
```

Description

The `srPtrSetupChild()` routine is used to set up a child pointer field within a parent pointer payload. STRIDE Runtime requires the setup of the pointer field information, enabling the Runtime to route the pointer locally and marshal the pointer across platforms. The setup routine is required for each child pointer field contained in the parent pointer payload.

When setting up the child pointer, it is necessary to first set up the parent pointer or get the handle to the parent pointer.



Each User sending a command payload containing child pointers is required to call the `srPtrSetupChild` routine. The Owner of a response payload which contains child pointers is required to call the `srPtrSetupChild` routine.

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact ID from the <code>srCreateSTID()</code> call.
<i>wParentHandle</i>	Input	Handle to the “parent” pointer that holds the nested (child) pointer
<i>wOffset</i>	Input	The pointer field offset within the parent pointer payload
<i>ePtrDir</i>	Input	The direction attribute of the pointer srPTRDIR_CMD_IN = 0 srPTRDIR_CMD_OUT = 1 srPTRDIR_CMD_INOUT = 2 srPTRDIR_CMD_RET = 3 srPTRDIR_RSP_RET = 3
<i>ePtrUsage</i>	Input	The pointer usage attribute

srPTR_USAGE_PRIVATE = 0

srPTR_USAGE_POOL = 1

<i>pwHandle</i>	Output (Optional)	The unique handle of the nested (child) pointer
-----------------	----------------------	---

Return Value

Description

srOK

Success

srERR_STID_INVALID

STID is invalid

srERR_PTR_DIR

Pointer direction invalid

srERR_PTR_USAGE

Pointer memory usage invalid

srERR_PTR_DUPLICATE

Duplicate pointer set up

srERR_PTR_OFFSET

Pointer offset invalid

srERR_PTR_ALLOC

Pointer allocation failed

Example

```
Interface.h
#include <sr.h>

#define INTERFACE SMID 50 + (srMT TWO | srST CMD VAL |
srST RSP VAL )

typedef struct
{
    srDWORD *pdwChild;
} InterfaceCmdFieldType;

typedef struct
{
    InterfaceCmdFieldType *ptField;
} InterfaceCmd;

InterfaceUser.c

#include <sr.h>
#include <Interface.h>

srWORD    wSTID;
srWORD    wParentHandle;
srWORD    wResult;

...

wResult = srPtrSetup( wSTID,
```

```
        INTERFACE SMID,  
        GET_OFFSET(InterfaceCmd,ptField) ,  
        srNULL,  
        sizeof(InterfaceCmdFieldType) ,  
        srMSGDIR_COMMAND,  
        srPTRDIR_CMD IN,  
        srPTR_USAGE_POOL,  
        &wParentHandle );  
  
wResult = srPtrSetupChild( wSTID,  
        wParentHandle,  
        GET_OFFSET(InterfaceCmdFieldType,pdwChild) ,  
        sizeof(srDWORD) ,  
        srPTRDIR_CMD IN,  
        srPTR_USAGE_POOL,  
        srNULL );  
  
...  
wResult = srPtrTeardown( wSTID,  
        INTERFACE SMID,  
        &wParentHandle );  
  
...
```

srPtrTeardown()

Free pointer handle

Prototype

```
srWORD srPtrTeardown (srWORD   wSTID,  
                      srDWORD  dwSMID,  
                      srWORD   wHandle )
```

Description

The srPtrTeardown routine is used to free the pointer handle previously allocated for the pointer setup.

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact ID from the <i>srCreateSTID()</i> call
<i>dwSMID</i>	Input	The unique SMID
<i>wHandle</i>	Input	Handle of the pointer from srPtrSetup or srPtrSetupChild

Return Value	Description
<i>srOK</i>	Success
<i>srERR_STID_INVALID</i>	STID invalid
<i>srERR_PTR_INVALID</i>	Pointer handle invalid
<i>srERR_PTR_TEARDOWN</i>	Pointer teardown failed

Example

```
Interface.h  
#include <sr.h>  
  
#define INTERFACE SMID 50 + (srMT TWO | srST CMD VAL |  
srST_RSP_VAL )  
  
typedef struct  
{  
    DWORD *pdwField;  
} InterfaceCmd;
```



```
IntefaceUser.c

#include <sr.h>
#include <Interface.h>

srWORD      wSTID;
srWORD      wHandle;
srWORD      wResult;

...

wResult = srPtrSetup( wSTID,
                      INTERFACE SMID,
                      GET OFFSET(InterfaceCmd,pdwField),
                      srNULL,
                      sizeof(srDWORD),
                      srMSGDIR COMMAND,
                      srPTRDIR CMD IN,
                      srPTR USAGE POOL,
                      &wHandle );

...

wResult = srPtrTeardown( wSTID,
                          INTERFACE SMID,
                          &wHandle );

...
```

srPtrGetHandle()

Retrieve handle

Prototype

```
srWORD srPtrGetHandle( srWORD    wSTID,  
                      srDWORD   dwMsgInst,  
                      srBYTE    *pyMemory,  
                      srWORD    *pwHandle );
```

Description

The srPtrGetHandle routine is used to retrieve a handle to a pointer using the message instance. This is used primarily by the Owner to access the handles of OUT and INOUT pointers in the message instance. The Owner then uses the handle to either change the size of the pointer, or to attach CMD_RET pointers.

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact Identifier from the srCreateSTID() call
<i>dwMsgInst</i>	Input	Message instance
<i>pyMemory</i>	Input	Pointer memory address
<i>pwHandle</i>	Output	Handle to the pointer in the dwMsgInst

Return Value	Description
<i>srOK</i>	Success
<i>srERR_STID_INVALID</i>	STID invalid
<i>srERR_PTR_INVALID</i>	Pointer handle invalid
<i>srERR_PTR_ADDRESS</i>	Pointer address not found
<i>srERR_PTR_LOCKED</i>	Pointer in lock state
<i>srERR_PTR_DIR</i>	Pointer direction invalid

Example

```
Owner.h

#include <sr.h> // Contains SMID Attributes

#define DOIT_NOW_SMID 60 + ( srMT_ONE | srST_CMD_VAL )

typedef srDWORD * MyDoitNow_t;

UserCode.c

#include <sr.h> // contains prototypes and defines
#include <owner.h>

MyDoitNow_t MyDoitNow;
srWORD      wMyEPtrHandle;
srWORD      wResult;

...

// Assuming STID already created
wResult = srEPtrSetup( wMySTID,
                      DOIT_NOW_SMID,
                      0, // Offset is zero
                      srMSGDIR_COMMAND,
                      srEPTRDIR_IN,
                      srEPTR_USAGE_POOL,
                      &wMyEPtrHandle );

...

wResult = srEPtrTearDown( wMyEPtrHandle );
```

srPtrSize()

Change pointer setup size field

Prototype

```
srWORD srPtrSize ( srWORD wSTID,  
                  srWORD wHandle,  
                  srWORD wSize );
```

Description

The `srPtrSize` routine is used to change the pointer setup size field for a pointer that has been set up previously. This allows the original pointer setup to be modified without tearing down the pointer and setting it up again with a new size. When an Owner responds with an OUT or INOUT pointer, the size of the OUT or INOUT pointer can be reduced if necessary, in order to minimize the number of bytes transferred between platforms.

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact Identifier from the <code>srCreateSTID()</code> call
<i>wHandle</i>	Input	Pointer handle
<i>wSize</i>	Input	New size for the pointer setup information

Return Value	Description
<i>srOK</i>	Success
<i>srERR_STID_INVALID</i>	STID is invalid
<i>srERR_PTR_INVALID</i>	Pointer handle invalid
<i>srERR_PTR_SIZE</i>	Pointer size is larger than the current size



When the Owner updates the size of an OUT or INOUT pointer, the new size can never be larger than the original size previously attached by the User.

Example

```
Interface.h

#include <sr.h>

#define INTERFACE_SMID 50 + (srMT_TWO | srST_CMD_VAL | srST_RSP_VAL )

typedef struct
{
    srDWORD *ptSizedOut;
} InterfaceCmd;

IntefaceUser.c

#include <sr.h>
#include <Interface.h>

srWORD    wSTID;
srWORD    wHandle;
srWORD    wResult;

...

wResult = srPtrSetup( wSTID,
                      INTERFACE_SMID,
                      GET_OFFSET(InterfaceCmd,ptSizedOut),
                      srNULL,
                      sizeof(DWORD) * 10,
                      srMSGDIR_COMMAND,
                      srPTRDIR_CMD_OUT,
                      srPTR_USAGE_PRIVATE,
                      &wHandle );

wResult = srSendCmd(...);

...

InterfaceOwner.c

#include <sr.h>
#include <Interface.h>

...

wResult = srRead( wSTID,
                 srBOX_1,
                 INTERFACE_SIZE,
```

```
        &dwSMID,  
        &tInterfaceCmd,  
        &sizeof(InterfaceCmd ),  
        &dwMsgInst );  
  
if( dwSMID == INTERFACE_SMID)  
{  
  
...  
  
    /* get the handle to the out pointer */  
    srPtrGetHandle(wSTID,  
                  dwMsgInst,  
                  tInterfaceCmd.ptSizedOut,  
                  &wHandle);  
  
    /* reduce the size of the out pointer */  
    srPtrSize(wSTID,  
              wHandle,  
              sizeof(srDWORD) * 5);  
  
...  
}  
  
...
```

srPtrCreateCmdInst()

Create command pointer instance

Prototype

```

srWORD  srPtrCreateCmdInst ( srWORD    wSTID,
                             srDWORD   dwSMID,
                             srBYTE    *pyPayload,
                             srWORD    wSize,
                             srDWORD   *pdwCmdInst );

```

Description

The srPtrCreateCmdInst routine is used to create a command pointer instance for srTraceInterface. This is used primarily for delegates for local tracing.

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact Identifier from the srCreateSTID() call
<i>dwSMID</i>	Input	The unique SMID
<i>pyPayload</i>	Input	Command payload
<i>wSize</i>	Input	Command payload size
<i>pdwCmdInst</i>	Output	Command pointer instance

Return Value	Description
<i>srOK</i>	Success
<i>srERR_PTR_OVERWRITE</i>	Payload pointer does not match pointer entry
<i>srERR_PTR_ALLOC</i>	Pointer allocation failed



dwCmdInst must be deleted by calling srPtrDeleteInst.

srPtrCreateRspInst()

Create response pointer instance

Prototype

```

srWORD srPtrCreatorRspInst ( srWORD    wSTID,
                             srDWORD   dwSMID,
                             srBYTE    *pyPayload,
                             srDWORD   dwCmdInst,
                             srDWORD   *pdwRspInst );

```

Description

The srPtrCreateRspInst routine is used to create a response pointer instance for srTraceInterface. This is used primarily for delegates for local tracing.

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact Identifier from the srCreateSTID() call
<i>dwSMID</i>	Input	The unique SMID
<i>pyPayload</i>	Input	Command payload
<i>wSize</i>	Input	Command payload size
<i>dwCmdInst</i>	Input	Command pointer instance
<i>pdwRspInst</i>	Output	Response pointer instance

Description

Return Value

<i>srOK</i>	Success
<i>srERR_PTR_OVERWRITE</i>	Payload pointer does not match pointer entry
<i>srERR_PTR_ALLOC</i>	Pointer allocation failed
<i>srERR_PTR_INVALID</i>	Pointer handle invalid
<i>srERR_PTR_DUPLICATE</i>	Pointer setup duplicate
<i>srERR_PTR_LOCKED</i>	Pointer in lock state



dwRspInst must be deleted by calling srPtrDeleteInst.

srPtrDeleteInst()

Delete pointer instance

Prototype

```
srWORD srPtrDeleteInst ( srWORD  wSTID,
                        srDWORD  dwPtrInst );
```

Description

The srPtrDeleteInst routine is used to delete a pointer instance created by srPtrCreateCmdInst or srPtrCreateRspInst. This is used primarily by delegates for local tracing.

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact Identifier from the srCreateSTID() call
<i>dwPtrInst</i>	Input	Pointer instance

Return Value	Description
<i>srOK</i>	Success
<i>srERR_PTR_INVALID</i>	Pointer handle invalid



dwPtrInst is created by srPtrCreateCmdInst or srPtrCreateRspInst.

3.4. Tracing

The following tracing routines are used by applications to instrument their software for greater visibility. The information collected by the STRIDE Runtime routines is available for collection by the host runtime environment. Each of the tracing routines requires a level as a parameter used for filtering. There are 8 levels supported by the STRIDE Runtime, with level zero (0) being the highest priority and level 7 being the lowest. If a tracing level of 3, for example, is selected then all traces of level 0, 1, 2 and 3 will be displayed.

The Tracing routines include:

- `srTracePoint()`
- `srTraceStr()`
- `srTraceInterface()`

srTracePoint()

Output data structure

Prototype

```
srWORD srTracePoint( srWORD    wSTID,
                    srWORD    wSTPID,
                    srBYTE    *pyPayload,
                    srWORD    wSize,
                    srLevel_e  eLevel );
```

Description

The `srTracePoint()` routine is used to output a data structure to the tracing window. The `eLevel` parameter is used to filter out different levels.

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact Identifier
<i>wSTPID</i>	Input	The unique STRIDE TracePoint ID
<i>pyPayload</i>	Input	Pointer to a payload
<i>wSize</i>	Input	The size of the payload
<i>eLevel</i>	Input	Indicates the tracing level to use srLEVEL_0 = 0 srLEVEL_1 = 1 srLEVEL_2 = 2 srLEVEL_3 = 3 srLEVEL_4 = 4 srLEVEL_5 = 5 srLEVEL_6 = 6 srLEVEL_7 = 7

Return Value	Description
<i>srOK</i>	Success
<i>srERR_STID_INVALID</i>	STID is invalid

Example

```
#include <sr.h> // contains prototypes and eLevel Type defines

#define DOIT_NOW_STPID    10

typedef struct {
    srBOOL    bValid;
    srDWORD   dwData;
} Doit_Now_TP;

Doit_Now_TP    MyDoit_Now;
srWORD         wResult;

MyDoit_Now.bValid = srTRUE;
MyDoit_Now.dwData = 43;

// Assuming STID already created
wResult = srTracePoint( wMySTID,
                        DOIT_NOW_STPID,
                        (srBYTE *)&MyDoit_Now,
                        sizeof(MyDoit_Now),
                        srLEVEL_0 );
```

srTraceStr()

Output string

Prototype

```

srWORD srTraceStr( srWORD    wSTID,
                  srCHAR    *szString,
                  srLevel_e  eLevel );

```

Description

The `srTraceStr()` routine is used to output a string that is automatically converted to a special trace point. The host runtime environment automatically displays this special trace point without requiring a unique STPID. The *eLevel* parameter is used to provide filtering based on levels.

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact Identifier.
<i>szString</i>	Input	A null terminated string.
<i>eLevel</i>	Input	Indicates the tracing level to use: srLEVEL_0 = 0 srLEVEL_1 = 1 srLEVEL_2 = 2 srLEVEL_3 = 3 srLEVEL_4 = 4 srLEVEL_5 = 5 srLEVEL_6 = 6 srLEVEL_7 = 7

Return Value	Description
<i>srOK</i>	Success
<i>srERR_STID_INVALID</i>	STID is invalid

Example

```
#include <sr.h> // contains prototypes and eLevel Type defines

srWORD wResult;

// Assuming STID already created

wResult = srTraceStr( wMySTID, "Should never get here!", srLEVEL_0 );
```

srTraceInterface()

Trace an interface

Prototype

```

srWORD srTraceInterface ( srWORD      wSTID,
                          srWORD      wSUID,
                          srBYTE      *pyPayload,
                          srWORD      wSize,
                          srDWORD     dwPtrInst,
                          srTraceType_e eTraceType );

```

Description

The `srTraceInterface()` routine is used to trace an interface. The `eTraceType` parameter specifies the interface trace type.

Command and response pointer instances for `srTraceInterface` are created by `srPtrCreateCmdInst` and `srPtrCreateRspInst`, respectively. These are used primarily by delegates for local tracing.

Parameters	Type	Description
<i>dwPtrInst</i>	Input	Handle returned by <code>srPtrCreateCmdInst</code> or <code>srPtrCreateRspInst</code> depending on <code>eTraceType</code>
<i>eTraceType</i>	Input	Interface trace type

Return Value	Description
<i>srOK</i>	Success
<i>srERR_STID_INVALID</i>	STID is invalid
<i>srERR_TRACE_TYPE</i>	Interface trace type invalid

3.5. Printing

The following printing routines are used by applications to display messages on Trace Views without having to specify a STID or a trace level and to set any trace filters on Trace Views. These routines support formatted strings with variable arguments. The `srPrintInfo()` will show as `srTraceStr()` with `STID=0` and `Level=3`. The `srPrintError()` will show as `srTraceStr()` with `STID=0` and the `Level=1`.

The Printing routines include:

- `srPrintInfo()`
- `srPrintError()`

srPrintInfo()

Output formatted string

Prototype

```
srWORD srPrintInfo( const srCHAR * szMsg, ... );
```

Description

The srPrintInfo() routine is used to output a formatted string with variable arguments that is automatically converted to a special trace string - srTraceStr with STID=0 and Level=3. The host runtime environment automatically displays this formatted string without having to set any trace filters.

Parameters	Type	Description
<i>szMsg</i>	Input	A string, which can be formatted. Cannot be null.
...	Input (Optional)	Variable argument list to format the string <i>szFmt</i> .
Return Value		Description
<i>srOK</i>		Success
<i>srERR</i>		STID is invalid

Example

```
#include <sr.h> // contains prototypes

srWORD wResult;
srWORD wNum1 = 5;
srWORD wNum2 = 10;

wResult = srPrintInfo("My info numbers: %d, %d", wNum1, wNum2);
```

srPrintError()

Output formatted string

Prototype

```
srWORD srPrintError( const srCHAR * szMsg, ... );
```

Description

The srPrintError() routine is used to output a formatted string with variable arguments that is automatically converted to a special trace string - srTraceStr with STID=0 and Level=1. The host runtime environment automatically displays this formatted string without having to set any trace filters.

Parameters	Type	Description
<i>szMsg</i>	Input	A string, which can be formatted. Cannot be null.
...	Input (Optional)	Variable argument list to format the string <i>szFmt</i> .

Return Value	Description
<i>srOK</i>	Success
<i>srERR</i>	STID is invalid

Example

```
#include <sr.h> // contains prototypes

srWORD wResult;
srWORD wNum1 = 5;
srWORD wNum2 = 10;

wResult = srPrintError("My error numbers: %d, %d", wNum1, wNum2);
```

3.6. Query

The following routines are used to retrieve status and internal information stored in the STRIDE Runtime.

- srQueryAccessClass()
- srQueryNID()
- srQueryName()
- srQuerySMID()
- srQueryBox()

srQueryAccessClass()

Query Access Class for registration

Prototype

```
srWORD  srQueryAccessClass ( srAccessClass_e  eAC,  
                             srWORD          *pwSTID,  
                             srBOX_e         *peBox );
```

Description

The *srQueryAccessClass()* routine is used to query if the given Access Class is registered with the STRIDE Runtime. If registered, the associated STID and mailbox are returned.

Parameters	Type	Description
<i>eAC</i>	Input	Access Class of type <i>srAccessClass_e</i> , defined as: <pre>typedef enum { srAC_REG_MESSAGES = 0, srAC_REG_FUNCTIONS = 1 } srAccessClass_e;</pre> Values are <i>srAC_REG_MESSAGES</i> or <i>srAC_REG_FUNCTIONS</i>
<i>pwSTID</i>	Output	The associated STRIDE Transact Identifier for the Access Class
<i>peBox</i>	Output	The associated mailbox for the Access Class

Return Value	Description
<i>srOK</i>	Success
<i>srErr</i>	Failure

Example

```
srWORD  wResult;  
srWORD  wSTID;  
srBOX_e eBox;  
wResult = srQueryAccessClass(srAC_IM, &wSTID, &eBox);
```

srQueryNID()

Retrieve STID From NID

Prototype

```
srWORD srQueryNID( srDWORD   dwNID,
                  srWORD   *pwSTID );
```

Description

The `srQueryNID()` routine is used to retrieve the STID associated with a NID. This is the STID that was created using `srCreateSTID()`.

Parameters	Type	Description
<i>DwNID</i>	Input	Notification Identifier
<i>pwSTID</i>	Output	STRIDE Transact ID associated with NID

Return Value	Description
<i>srOK</i>	Success
<i>srERR_NID_NOT_FOUND</i>	No STID created for NID

Example

```
#include <sr.h> // contains prototypes and defines

srWORD  wResult;
srWORD  wSTID;
srDWORD dwMyThreadId;

dwMyThreadId = CustomerGetCurrentThreadId();

wResult = srQueryNID( dwMyThreadId, &wSTID );
```

srQueryName()

Look up STID based on STID name

Prototype

```
srWORD srQueryName ( srCHAR    *szName,  
                    srWORD    *pwSTID );
```

Description

The `srQueryName()` routine is used to look up the STID based on the STID's name. It takes a null-terminated string as a parameter and returns the STID. The name was set when the STID was created by the call `srCreateSTID`.

Parameters	Type	Description
<i>szName</i>	Input	Name used when creating the STID
<i>pwSTID</i>	Output	STRIDE Transact Identifier (STID)

Return Value	Description
<i>srOK</i>	Success
<i>srERR_NAME_NOT_FOUND</i>	No matching STID for name

Example

```
#include <sr.h> // contains prototypes and defines  
  
srWORD wResult;  
  
srWORD wSTID;  
  
// Assuming NID already setup  
  
wResult = srCreateSTID( dwMyNID, "MySTIDName", wSTID, srTRUE );  
  
wResult = srQueryName( "MySTIDName", &wSTID );
```

srQuerySMID()

Query a specific SMID

Prototype

```
srWORD srQuerySMID ( srDWORD      dwSMID,
                    srSMIDInfo_t *ptSMIDInfo );
```

Description

The srQUERYSMID routine is used to query information about a specific STRIDE Message ID (SMID). It will return the MID, SMID attributes, registration status, registered STID, and number of current subscribers (if applicable).

Parameters	Type	Description
<i>dwSMID</i>	Input	The unique SMID
<i>ptSMIDInfo</i>	Output	ptSMIDInfo is of type srSMIDInfo_t, defined as: <pre>typedef struct { srBOOL bReg; srBYTE yRegSTID; srBOX_e eRegBox; srBOOL bRegOverride; srWORD wSubIdsCnt; } srSMIDInfo_t;</pre> bReg: Registration status srTrue = Registered srFalse = Not registered yRegSTID: Registered STID eRegBox: Specified message box bRegOverride: Register override wSubsIdsCnt: Count of current subscribers

Return Value	Description
<i>srOK</i>	Success
<i>srERR_SMID_OUT_OF_RANGE</i>	SMID is not in configured range



This routine can be used to avoid the overhead of generating broadcast information if there are no subscribers. Also, the STRIDE Runtime only broadcasts response payloads when one or more active subscribers exists.

This routine only checks the SMID on the local platform. Refer to the STRIDE Runtime Message Services (srmsg.h) to check a SMID on a remote platform.

Example

```
Owner.h

#include <sr.h>    // Contains SMID Attributes

#define DOIT_GOTIT_SMID 40 + ( srMT_BCAST | srST_RSP_VAL )

typedef struct {
    srDWORD dwData3;
} DoitGotit_Rsp;

OwnerCode.c

#include <sr.h>    // contains prototypes and defines
#include <owner.h>

srWORD          wResult;
DoitGotit_Rsp   MyDoitGotit_Rsp;
srSMIDInfo_t    tSMIDInfo;

// Assumes STID already created
wResult = srRegister( wMySTID, srBOX_1, DOIT_GOTIT_SMID, srTRUE );

MyDoitGotit_Rsp.dwData3 = 42;

wResult = swQuerySMID( DOIT_GOTIT_SMID, &tSMIDInfo );

if ( ( tSMIDInfo.wSubsIdsCnt >= 0 ) {

    wResult = srBroadcast( wMySTID, DOIT_GOTIT_SMID,
                          (srBYTE *) &MyDoitGotit_Rsp,
                          sizeof(MyDoitGotit_Rsp) );

}
```


srQueryBox()

Check mailbox

Prototype

```
srWORD srQueryBox ( srDWORD      wSTID,
                   srBox_e      eBox,
                   srBoxInfo_t  *ptBoxInfo );
```

Description

The `srQueryBox()` routine is used to check a specific mailbox, reporting the number of pending messages, the next message's SMID, and the size of the next message in the box.

Parameters	Type	Description
<i>wSTID</i>	Input	STRIDE Transact Identifier
<i>eBox</i>	Input	The mailbox to check
<i>ptBoxInfo</i>	Output	The number of pending messages, the next message's SMID, and the size of the next message in the box.

Return Value	Description
<i>srOK</i>	Success
<i>srERR_STID_INVALID</i>	STID is invalid

Example

```
#include <sr.h> // contains prototypes and defines

srWORD      wResult;
srBoxInfo  tBoxInfo;

...

wResult = srQueryBox( wMySTID,
                    srBOX_1,
                    &tBoxInfo );
```

```
if ( tBoxInfo.wNumPending > 0 )
{
    //messages waiting, do something.
}
```

3.7. Access Class (Remote Messaging) Routines

These routines are used for Access Class registration, which includes Remote Messaging and Access Class Intercept Module.

- `srRegisterAccessClass()` – see Section 3.2 on Messaging
- `srQueryAccessClass()` – see Section 3.5 on Query

3.8. I-block

This routine is used to indicate when the next I-block can be sent out to the system transport mechanism.

- `srIBlockOutReady()`

`srIBlockOutReady()`

IBlockOut Is Ready

Prototype

```
srWORD srIblockOutReady( void )
```

Description

The `srIBlockOutReady()` routine is called to indicate to the STRIDE Runtime that the system transport mechanism is ready to dispatch an I-block. Any pending I-block will be queued in the STRIDE Runtime until this routine is called.

After this routine is called `palOut()` is called to send out the next pending I-block. If the transport mechanism uses static buffers to receive an I-block, this function should be used once the buffer can be safely overwritten.

Parameters	Type	Description
------------	------	-------------

None

Return Value	Description
--------------	-------------

srWORD srOK

Success

Example

```
#include <sr.h> // contains prototypes and defines

srWORD      wResult;
srBYTE      yTPBuffer [MAX_TP_SIZE];
...

// Send current buffer
CustomerTransportSendBuffer( &yTPBuffer );
```

```
// done with buffer, indicate ready for next I-block  
wResult = srIblockOutReady();
```

3.9. Runtime Thread Entry and Exit Points

These routines are related to the operation of the STRIDE Runtime Thread.

- `srThread()`
- `srThreadInit()`
- `srThreadUninit()`
- `srThreadProc()`

srThread()

STRIDE Runtime Thread

Prototype

```
void srThread( void );
```

Description

The STRIDE Runtime entry points consist of srThreadInit and srThreadProc. The srThread routine combines the two entry points into a single control loop.

Parameters	Type	Description
------------	------	-------------

None		
------	--	--

Return Value	Description
--------------	-------------

None	
------	--

Example

```
#include <sr.h> // contains prototypes and defines

void StartApplication( void )
{
    . . .
    rtosBeginThread( srThread, PRIORITY, &dwThreadId );
}
```

srThreadInit()

STRIDE Runtime Initialization

Prototype

```
void srThreadInit( void );
```

Description

The srThreadInit routine is used to initialize the STRIDE Runtime. If srThread is spawned on the Target system, this routine does not need to be directly called; otherwise, the user must call srThreadInit before using the STRIDE Runtime.

Parameters	Type	Description
None		

Return Value	Description
None	

Example

```
#include <sr.h> // contains prototypes and defines

void StartApplication( void )
{
    . . .
    /*
    NOT CALLED    rtosBeginThread(srMiddleThread, PRIORITY, &dwThreadId) ;

    / . . .
    . . .
    srThreadInit(); /* must be called before using the STRIDE Runtime */
}
}
```

srThreadUninit()

STRIDE Runtime Uninitialization**Prototype**

```
void srThreadUninit( void );
```

Description

The srThreadUninit routine is used to uninitialized the STRIDE Runtime. If srThread is spawned on the Target system, this routine does not need to be directly called; otherwise, the user must call srThreadUninit to shutdown the STRIDE Runtime.

Parameters	Type	Description
------------	------	-------------

None		
------	--	--

Return Value	Description
--------------	-------------

None	
------	--

Example

```
#include <sr.h> // contains prototypes and defines

void StopApplication( void )
{
    . . .
    srThreadUninit(); /* must be called when stopping STRIDE Runtime */
}
```

srThreadProc()

STRIDE Runtime Thread

Prototype

```
void srThreadProc ( void );
```

Description

The srThreadProc routine is used to process the STRIDE Runtime system events. This does not need to be called if spawning srThread; otherwise, the user must call srThreadProc to drive the Runtime engine.

Parameters	Type	Description
------------	------	-------------

None		
------	--	--

Return Value	Description
--------------	-------------

None	
------	--

Example

```
#include <sr.h> // contains prototypes and defines

void StartApplication( void )
{
    . . .
    /*
    NOT CALLED rtosBeginThread(srMiddleThread, PRIORITY, &dwThreadId);
    */
    srThreadInit();
    ...
    /* main processing loop */
    for(;;)
        ...
        srThreadProc();
        ...
}
```

3.10. Host Override Routines

These routines are related to the use of Intercept Modules.

- `srHostIMPoolMemOverride()`

3.10.1. srHostShutdownIM()

STRIDE Runtime Thread

Prototype

```
typedef void ( *srHostIMShutdownFunc_t ) ( void );
srDWORD srHostShutdownIM ( srHostIMShutdownFunc_t pShutdownFunc );
```

Description

srHostShutdownIM() is used to shut down a STRIDE Intercept Module thread started with srHostStartIM(). This routine will call the Intercept Module shutdown routine passed in as a parameter. The routine cleans up associated STRIDE resources after the thread has finished.

Parameters	Type	Description
pShutdownFunc	Input	Pointer to an IM Shutdown routine.

Return Value	Description
None	

Example

```
#include <sr.h> // contains prototypes and defines
#include <myFuncsIM.h> // STRIDE generated Intercept Module

void myAppInit( void )
{
    . . .
    srHostStartIM(myFuncsIMInit, // IM Init routine
                 myFuncsIMRead) // IM Read routine
    . . .
}

void myApp( void )
{
    int x;
    . . .
    x = myFunc(123); //calls proxy to myFunc().
                   //synchronization between Proxy/Stub handled
                   //by STRIDE Host.
}
```

```
void myAppExit( void )
{
    ...
    srHostShutdownIM(myFuncsIMShutdown);
    ...
}
```

3.11. Connecting

The following messages are used to connect to a remote platform or disconnect from an already established remote connection.

- `srCONNECT_OPEN_T_SMID`
- `srCONNECT_CLOSE_T_SMID`
- `srCONNECT_STATUS_B_SMID`
- `srCONNECT_STATUS_T_SMID`

srCONNECT_OPEN_T_SMID

Request remote connection

The srCONNECT_OPEN_T_SMID message is a two-way message used to request a remote connection. If the connect request timeouts, the STRIDE Runtime will enter the Listen state.

Attributes	Description
<i>srMT_TWO</i>	Two-way message
<i>srST_RSP_VAL</i>	Response payload sent by value
<i>srAC_SYS</i>	STRIDE Runtime system message
Command	Description
<i>No payload</i>	Used to initiate a connection
Response	Description
<i>srConnectOpenRsp_t</i>	Contains the results of the connection request.
<i>eConnection</i>	Indicates the connection result: srCONNECTION_CLOSED = 0 srCONNECTION_OPEN = 1 srCONNECTION_LISTEN = 2

Example

```
#include <sr.h> // contains prototypes and defines
#include <srmsg.h> // contains message definitions

srWORD          wResult;
srDWORD         dwMySMID;
srWORD          wMsgSize;
srDWORD         dwMsgInst;
srConnectOpenRsp_t tConnectOpenRsp;

//assumes STID already created
wResult = srSendCmd( wMySTID,
                    srBOX_1,
                    srCONNECT_OPEN_T_SMID,
                    srNULL,
                    0 );

//Wait for Response
CustomerEventWait( CUST_EVENT_MBOX1 );

wResult = srRead( wMySTID,
                 srBOX_1,
                 sizeof(tConnectOpenRsp),
                 &dwMySMID,
                 (srBYTE*)&tConnectOpenRsp,
                 &wMsgSize,
                 &dwMsgInst );

// check to see if connected successfully
if( tConnectOpenRsp.eConnection != srCONNECTION_OPEN )
{
    //do something
}
```

See Also

srIntialize, srSendCmd, srRead, srCONNECT_CLOSE_T_SMID

srCONNECT_CLOSE_T_SMID***Terminate remote connection***

The srCONNECT_CLOSE_T_SMID message is a two-way message used to terminate a remote connection. This command has no payload. When the connection is closed the response is returned. To check the connection status the *srCONNECT_STATUS_B_SMID* subscription can be used.

Attributes	Description
<i>srMT_TWO</i>	Two-way message
<i>srAC_SYS</i>	STRIDE Runtime system message
Command	Description
No payload	Used to close a connection
Response	Description
No payload	Indicates the connection has been released

Example

```
#include <sr.h>           // contains prototypes and defines
#include <srmsg.h>        // contains message definitions

srWORD                    wResult;
srDWORD                   dwMySMID;
srWORD                    wMsgSize;
srDWORD                   dwMsgInst;

//assumes STID already created
wResult = srSendCmd( wMySTID,
                    srBOX_1,
                    srCONNECT_CLOSE_T_SMID,
                    (srBYTE*)NULL,
                    0 );

//Wait for Response
CustomerEventWait( CUST_EVENT_MBOX1 );

wResult = srRead( wMySTID
```

```
srBOX_1,  
0,  
&dwMySMID,  
0,  
&wMsgSize,  
&dwMsgInst );
```

srCONNECT_STATUS_B_SMID**Broadcast connection status change**

The srCONNECT_STATUS_B_SMID message is broadcast when the connection status changes.

Attributes	Description
<i>srMT_BRD</i>	Broadcast message
<i>srST_RSP_VAL</i>	Response payload sent by value
<i>srAC_SYS</i>	STRIDE Runtime system message
Response	Description
<i>srConnectOpenRsp_t</i>	Contains the results of the connection request
<i>eConnection</i>	Indicates the Connection results: srCONNECTION_CLOSED = 0 srCONNECTION_OPEN = 1 srCONNECTION_LISTEN = 2

Example

```
#include <sr.h>          // contains prototypes and defines
#include <srmsg.h>       // contains message definitions

srWORD          wResult;
srDWORD         dwMySMID;
srWORD          wMsgSize;
srDWORD         dwMsgInst;
srConnectStatusRsp_t tConnectStatusRsp;

//assumes STID already created
wResult = srSubscribe( MySTID, srBOX_1, srCONNECT_STATUS_B_SMID, srTRUE );

CustomerEventWait( CUST_EVENT_MBOX1 );

wResult = srMsgRead( wMySTID,
                    srBOX_1,
                    sizeof(tConnectStatusRsp),
                    &dwMySMID,
                    (srBYTE*)&tConnectStatusRsp,
```

```
        &wMsgSize,  
        &dwMsgInst );  
  
if(tConnectStatusRsp.eConnection != srCONNECTION_OPEN)  
{  
    //Lost connection  
}
```

srCONNECT_STATUS_T_SMID***Request Connection Status***

The srCONNECT_STATUS_T_SMID message

Attributes	Description
<i>srMT_TWO</i>	Two-way message
<i>srST_RSP_VAL</i>	Response payload sent by value
<i>srAC_SYS</i>	STRIDE Runtime system message

Response	Description
<i>srConnectOpenRsp_t</i>	Contains the results of the connection request.
<i>eConnection</i>	Indicates the Connection results: srCONNECTION_CLOSED = 0 srCONNECTION_OPEN = 1 srCONNECTION_LISTEN = 2

Example

```
#include <sr.h> // contains prototypes and defines
#include <srmsg.h> // contains message definitions

srWORD          wResult;
srDWORD         dwMySMID;
srWORD          wMsgSize;
srDWORD         dwMsgInst;
srConnectStatusRsp_t tConnectStatusRsp;

//assumes STID already created
wResult = srSendCmd( wMySTID,
                    srBOX_1,
                    srCONNECT_STATUS_T_SMID,
                    (srBYTE*)NULL,
                    0 );

//Wait for Response
CustomerEventWait( CUST_EVENT_MBOX1 );

wResult = srMsgRead( wMySTID,
                    srBOX_1,
```

```
        sizeof(tConnectStatusRsp),
        &dwMySMID,
        (srBYTE*)&tConnectStatusRsp,
        &wMsgSize,
        &dwMsgInst );

if(tConnectStatusRsp.eConnection != srCONNECTION_OPEN)
{
    //Lost connection
}
```

3.12. Database Loading Routines

The following messages are used to load the STRIDE databases.

- srHOST_LOAD_DB_O_SMID
- srHOST_LOAD_DB_STATUS_B_SMID
- srHOST_LOAD_DB_STATUS_T_SMID

srHOST_LOAD_DB_O_SMID

Load database file

The srHOST_LOAD_DB_O_SMID message is used to tell the Host Runtime to load a STRIDE database file.

Attributes	Description
<i>srMT_ONE</i>	One-way message
<i>srST_CMD_VAL</i>	Command payload sent by value
<i>srAC_SYS_STRIDE</i>	Runtime system message

Response	Description
<i>srHostLoadDatabaseCmd_t</i>	Contains the database filename and path
<i>szDatabase</i>	String containing name and path to database

Example

```
srHostLoadDatabaseCmd_t tLoadDatabase;  
  
/* Set name of database to load */  
strcpy(tLoadDatabase.szDatabase, "c:\\Stride\\myproj\\mydatabase.xml");  
  
/* Send command to load database */  
srSendCmd( mySTID,  
           srBOX_1,  
           srHOST_LOAD_DB_O_SMID,  
           (srBYTE*)&tLoadDatabase,  
           sizeof(srHostLoadDatabaseCmd_t) );
```


srHOST_LOAD_DB_STATUS_B_SMID**Receive host database status**

The srHOST_LOAD_DB_STATUS_B_SMID message is used to receive the current host database status. The status will indicate the name of a database and its load state. This message is broadcast when the status changes.

Attributes	Description
<i>srMT_BCST</i>	Broadcast message
<i>srST_RSP_VAL</i>	Response sent by value
<i>srAC_SYS_STRIDE</i>	Runtime system message

Response	Description
<i>srHostLoadDBStatusRsp_t</i>	Database status information
<i>szDatabase</i>	String containing name and path to database
<i>eStatus</i>	srHOST_DB_STATUS_NOT_LOADED srHOST_DB_STATUS_LOADED srHOST_DB_STATUS_LOADING srHOST_DB_STATUS_ERROR

Example

```
srHostLoadDBStatusRsp_t tLoadStatus;

srDWORD dwSMID;
srDWORD dwSize;

srSubscribe( mySTID,
             srBOX_2, /* Receive broadcast in Box 2 */
             srHOST_LOAD_DB_STATUS_B_SMID,
             srTRUE );

while(1)
{
    /* Wait for broadcast message */
    os_wait_event(MBOX2_EVENT);

    /* Read message (assume only status broadcast on Box 2) */
    srRead( mySTID,
            srBOX_2,
```

```
        sizeof(srHostLoadDBStatusRsp_t),
        &dwSMID,
        (srBYTE*)&tLoadStatus,
        &dwSize,
        &dwMsgInst );

/* Make sure SMID is correct */

if(dwSMID == srHOST_LOAD_DB_STATUS_T_SMID)
{
    switch(tLoadStatus.eStatus)
    {
        case srHOST_DB_STATUS_NOT_LOADED:
            /* No Database Loaded */
            break;

        case srHOST_DB_STATUS_LOADED:
            /* Database is loaded */
            strcpy(szDatabaseLoadedString,
                tLoadStatus.szDatabase);
            break;

        case srHOST_DB_STATUS_LOADING:
            /* Database is still loading */
            break;

        case srHOST_DB_STATUS_ERROR:
            break;

        default:
            /* Handle Error */
            break;
    }
}
}
```

srHOST_LOAD_DB_STATUS_T_SMID**Query runtime for database status**

The srHOST_LOAD_DB_O_SMID message is used to tell the Host Runtime to load a STRIDE database file.

Attributes	Description
<i>srMT_TWO</i>	Two-way message
<i>srST_RSP_VAL</i>	Response sent by value
<i>srAC_SYS_STRIDE</i>	Runtime system message

Response	Description
<i>srHostLoadDBStatusRsp_t</i>	Database status information
<i>szDatabase</i>	String containing name and path to database
<i>eStatus</i>	srHOST_DB_STATUS_NOT_LOADED srHOST_DB_STATUS_LOADED srHOST_DB_STATUS_LOADING srHOST_DB_STATUS_ERROR

Example

```
srHostLoadDatabaseCmd_t tLoadDatabase;
srHostLoadDBStatusRsp_t tLoadStatus;
srDWORD dwSMID;
srDWORD dwSize;

/* Set name of database to load */
strcpy(tLoadDatabase.szDatabase, "c:\Stride\myproj\mydatabase.xml");

/* Send command to load database */
srSendCmd( mySTID,
           srBOX_1,
           srHOST_LOAD_DB_O_SMID,
           (srBYTE*)&tLoadDatabase,
           sizeof(srHostLoadDatabaseCmd_t) );

/* Query the Current Status */
srSendCmd( mySTID,
           srBOX_2,
```

```
        srHOST_LOAD_DB_STATUS_T_SMID,  
        srNULL, /* No Command Payload */  
        0 );  
  
/* Wait for response to Two-Way message */  
os_wait_event(MBOX2_EVENT);  
  
/* Read response (assume only response to this command on Box 2) */  
srRead( mySTID,  
        srBOX_2,  
        sizeof(srHostLoadDBStatusRsp_t),  
        &dwSMID,  
        (srBYTE*)&tLoadStatus,  
        &dwSize,  
        &dwMsgInst );  
  
/* Make sure SMID is correct */  
if(dwSMID == srHOST_LOAD_DB_STATUS_T_SMID)  
{  
    switch(tLoadStatus.eStatus)  
    {  
        case srHOST_DB_STATUS_NOT_LOADED:  
            /* No Database Loaded */  
            break;  
  
        case srHOST_DB_STATUS_LOADED:  
            /* Database is loaded */  
            strcpy(szDatabaseLoadedString,  
                tLoadStatus.szDatabase);  
            break;  
  
        case srHOST_DB_STATUS_LOADING:  
            /* Database is still loading */  
            break;  
  
        case srHOST_DB_STATUS_ERROR:  
            break;  
  
        default:  
            /* Handle Error */  
            break;  
    }  
}
```

3.13. Trace Buffers

The following message is used to broadcast the current STRIDE trace buffer.

- `srTRACE_BUFFER_B_SMID`

srTRACE_BUFFER_B_SMID

Broadcast current trace buffer

The `srTRACE_BUFFER_B_SMID` message is used to broadcast the current trace buffer. When the STRIDE Runtime tracing sub-system has a trace buffer available to send, users subscribed to this message will receive that buffer. This can be used for tracing when a host connection is not available and trace logs need to be saved for future viewing. The trace buffers can then be sent back to the STRIDE Runtime using the `srTRACE_BUFFER_SEND_O_SMID` command.

Attributes	Description
<code>srMT_BRD</code>	Broadcast message
<code>srST_RSP_PTR</code>	Response payload sent by pointer
<code>srPU_RSP_POL</code>	Response payload pointer uses pool memory
<code>srAC_SYS</code>	STRIDE Runtime system message

Response	Description
<code>srTraceBufferRsp_t</code>	Contains the trace buffer
<code>wTraceBufSize</code>	Indicates the size of the conformant array
<code>TraceBuffer[1]</code>	Conformant array containing the trace buffer

Example

```
#include <sr.h>          // contains prototypes and defines
#include <srmsg.h>       // contains message definitions

srWORD                  wResult;
srDWORD                dwMySMID;
srWORD                 wMsgSize;
srDWORD                dwMsgInst;
srTraceBufferRsp_t    ptTraceBuffer;

//assumes STID already created
wResult = srSubscribe( wMySTID,          //Assumes valid STID
                      srBOX_1,         //Box to receive broadcast
                      srTRACE_BUFFER_B_SMID, //ID of Trace Buffer Message
                      srTRUE );        //Enable subscription

CustomerWaitEvent( CUST_EVENT_MBOX1 ); // Wait for subscription

...

//Read Broadcast Message
wResult = srRead( wMySTID,
                  srBOX_1,
                  sizeof(ptTraceBuffer), // reading a pointer
                  &dwMySMID,
                  &ptTraceBuffer,
                  &wMsgSize,
                  &dwMsgInst );

// Do something with the buffer ( ptTraceBuffer->TraceBuffer[n] )

CustomerFreeMemory(ptTraceBuffer);
```

3.13.1. Trace Filtering

The following message is used for configuring local filtering options related to the trace buffer.

- `srTRACE_FILTER_STID_O_SMID`

srTRACE_FILTER_STID_O_SMID

Set up local STID tracing filters

The `srTRACE_FILTER_STID_O_SMID` command message is used to set up local filters for STID tracing. A local application can set filter options for a particular STID. Options include whether or not to include payload data, enabling/disabling MID (Message ID) tracing, enabling/disabling STPID tracing, setting STPID logging level and globally disabling message tracing for the specific STID.

Attributes	Description
<i>srMT_ONE</i>	One-way message
<i>srST_CMD_VAL</i>	Command payload sent by value
<i>srAC_SYS</i>	STRIDE Runtime system message
Command	Description
<i>srTraceFilterSTIDCmd_t</i>	Contains the STID filter
<i>wSTID</i>	Unique STID identifier
<i>bGlobalDisable</i>	Global disable any tracing for STID
<i>bEnableMID</i>	Enable/disable MID tracing
<i>bPayloadMID</i>	Enable/disable MID payloads
<i>bEnableSTPID</i>	Enable/disable trace point tracing
<i>bPayloadSTPID</i>	Enable/disable trace point payloads
<i>eLevel</i>	Active level for trace points

Example

```
#include <sr.h>          // contains prototypes and defines
#include <srmsg.h>       // contains message definitions

srTraceFilterSTIDCmd_t  tTraceFilterSTIDCmd;
srWORD                  wGUIThreadSTID;
srWORD                  srResult;

// assuming "GUIThreadName" exists
srResult = srQueryName( "GUIThreadName", &wGUIThreadSTID );

// Enable Message Tracing in UI Thread
tTraceFilterSTIDCmd.wSTID          = wGUIThreadSTID;
tTraceFilterSTIDCmd.bGlobalDisable = srFALSE;
tTraceFilterSTIDCmd.bEnableMID     = srTRUE;
tTraceFilterSTIDCmd.bPayloadMID    = srTRUE;
tTraceFilterSTIDCmd.bEnableSTPID   = srFALSE;
tTraceFilterSTIDCmd.bPayloadSTPID  = srFALSE;
tTraceFilterThreadCmd.eLevel       = srLEVEL_0

//assumes STID already created
wResult = srSendCmd( wMySTID,
                    srBOX_1,
                    srTRACE_FILTER_STID_O_SMID,
                    (srBYTE*)&tTraceFilterSTIDCmd,
                    sizeof(srTraceFilterSTIDCmd_t) );
```

3.14. Subscriber Information

The following messages are used to receive subscriber information. One routine is for the local subscribers and one for remote subscribers.

- srSUBSCRIBERS_LOCAL_B_SMID
- srSUBSCRIBERS_REMOTE_B_SMID

srSUBSCRIBERS_LOCAL_B_SMID

Receive local subscriber information

The srSUBSCRIBERS_LOCAL_B_SMID message is used to receive local subscriber information. Subscribers of this message will receive the subscribe information that the Runtime receives on the local platform.

Attributes	Description
<i>srMT_BRD</i>	Broadcast message
<i>srST_RSP_VAL</i>	Response payload sent by value
<i>srAC_SYS</i>	STRIDE Runtime system message

Response	Description
<i>srRemoteSubRsp_t</i>	Contains the remote subscriber information
<i>WSRID</i>	STRIDE Response Identifier
<i>WSUID</i>	STRIDE Unique Identifier
<i>BOn</i>	Indicates whether subscribe is <i>on</i> or <i>off</i>

Example

```
#include <sr.h>           // contains prototypes and defines
#include <srmsg.h>        // contains message definitions

srWORD                    wResult;
srDWORD                   dwMySMID;
srWORD                    wMsgSize;
srDWORD                   dwMsgInst;
srRemoteSubRsp_t          tRemoteSub;

//assumes STID already created
wResult = srSubscribe( wMySTID,           //Assumes valid STID
                      srBOX_1,          //Box to receive broadcast
                      srSUBSCRIBERS_LOCAL_B_SMID, //ID of LocalSub msg
                      srTRUE );         //Enable subscription

CustomerWaitEvent( CUST_EVENT_MBOX1 ); // Wait for subsubscription

...
```

```
//Read Broadcast Message
wResult = srRead( wMySTID,
                 srBOX_1,
                 sizeof(tRemoteSub), // reading a pointer
                 &dwMySMID,
                 &tRemoteSub,
                 &wMsgSize,
                 &dwMsgInst );

// Do something with the sub info
AddSubToMySubList(tRemoteSub);
```

srSUBSCRIBERS_REMOTE_B_SMID

Receive remote subscriber information

The srSUBSCRIBERS_REMOTE_B_SMID message is used to receive remote subscriber information. Subscribers of this message will receive the subscribe information that the Runtime receives from the remote platform.

Attributes	Description
<i>srMT_BRD</i>	Broadcast message
<i>srST_RSP_VAL</i>	Response payload sent by value
<i>srAC_SYS</i>	STRIDE Runtime system message

Response	Description
<i>srRemoteSubRsp_t</i>	Contains the remote subscriber information
<i>wSRID</i>	STRIDE Response Identifier
<i>wSUID</i>	STRIDE Unique identifier
<i>bOn</i>	Indicates the subscribe is on or off

Example

```
#include <sr.h>           // contains prototypes and defines
#include <srmsg.h>        // contains message definitions

srWORD           wResult;
srDWORD          dwMySMID;
srWORD           wMsgSize;
srDWORD          dwMsgInst;
srRemoteSubRsp_t tRemoteSub;

//assumes STID already created
wResult = srSubscribe( wMySTID,           //Assumes valid STID
                      srBOX_1,          //Box to receive broadcast
                      srSUBSCRIBERS_REMOTE_B_SMID, //ID of RemoteSub msg
                      srTRUE );         //Enable subscription
```

```
CustomerWaitEvent( CUST_EVENT_MBOX1 ); // Wait for subsubscription

...

//Read Broadcast Message
wResult = srRead( wMySTID,
                 srBOX_1,
                 sizeof(tRemoteSub), // reading a pointer
                 &dwMySMID,
                 &tRemoteSub,
                 &wMsgSize,
                 &dwMsgInst );

// Do something with the sub info
AddSubToMySubList(tRemoteSub);
```

3.15.

3.16. Marshaling Errors

The following message is used to receive remote subscriber information.

- `srERROR_MARSHAL_B_SMID`

srERROR_MARSHAL_B_SMID**Receive remote subscriber information**

The srERROR_MARSHAL_B_SMID message is used to receive remote subscriber information. Subscribers of this message will receive the subscribe information that the Runtime receives from the remote platform.

Attributes	Description
<i>srMT_BRD</i>	Broadcast message
<i>srST_RSP_VAL</i>	Response payload sent by value
<i>srAC_SYS</i>	STRIDE Runtime system message

Response	Description
<i>srErrMarshalRsp_t</i>	Contains the marshaling error information
<i>wCode</i>	Marshaling error code (for a list of marshaling error codes, refer to sr.h on page 205).
<i>bCmd</i>	Boolean flag to indicate direction: srTRUE = Command srFALSE = Response
<i>dwSMID</i>	Message identifier
<i>wRmtSRID</i>	STRIDE Response Identifier
<i>bMarshalOut</i>	Direction of the marshaling error: TRUE = outgoing marshaling error (target to host) FALSE = incoming marshaling error (host to target)

Example

```
// Assume STID has been created
// Subscribe to marshaling error message
srSubscribe(wSTID, srBOX_1, srERROR_MARSHAL_B_SMID, srTRUE);

.....

// Wait for message to be received
```

```
// Read message from input queue

wRet = _srCgUtil_Read(wSTID, srBOX_1, wMsgBuffSize, &dwSMIDRead,
                    pyMsgBuff, &wMsgSize, pdwMsgInst);

.....

if(dwSMIDRead==srERROR_MARSHAL_B_SMID)
{
    srErrMarshalRsp_t* marshalErr;
    marshalErr = (srErrMarshalRsp_t*)pyMsgBuff;
    // Error handling
}
}
```

4. Runtime Test Services (RTS)

The Runtime Test Services (RTS) are a set of C-based APIs in the STRIDE Runtime that facilitate the writing of target based test code. These APIs make up an optional portion of the STRIDE Runtime and can be used to communicate additional information about tests to the host based reporting mechanism. These APIs also allow target test code to create additional test suites and test cases dynamically at runtime.

These C-based APIs work equally well from **C Test Functions** and **C++ Test Classes**. If, however, you choose to derive your C++ test classes from the **STRIDE Runtime Base Class (srTest)**, then you will have access to member objects in srTest class and their methods that provide the same functionality as the C APIs.

C Test Functions

C test functions enable testing of C code similarly to xUnit-style testing. Test functions can be written by users, captured using the `scl_test_flist` pragma, and executed from the host. C-based Runtime Test Services APIs are available for use in test functions.

C++ Test Classes

C++ test classes enable testing of C++ code similarly to xUnit-style testing. Test classes can be written by users, captured using `scl_test_class`, `scl_test_setup` and `scl_test_teardown` pragmas, and executed from the host. C-based Runtime Test Services APIs as well as C++ Runtime Base Class are available for use in test classes.

C-based Runtime Test Services APIs provided:

- **srTestSuiteAddSuite**: creates an additional sub-suite at runtime.
- **srTestSuiteSetName**: sets the name of the specified suite.
- **srTestSuiteSetDescription**: sets the description of the specified suite.
- **srTestSuiteAddTest**: creates an additional test case at runtime.
- **srTestCaseSetName**: sets the name of the specified test case.
- **srTestCaseSetDescription**: sets the description of the specified test case.
- **srTestCaseAddComment**: adds a comment to the specified test case.
- **srTestCaseSetStatus**: explicitly sets the status for the specified test case.
- **srTestCaseSetStatusEx**: explicitly sets the status for the specified test case and allows specification of an extended failure code.
- **srTestSuiteAddAnnotation**: creates an annotation at runtime.
- **srTestAnnotationAddComment**: adds a comment to the specified annotation.

Member objects of srTest base class, via which you can access the same functionality:

- **testSuite** member object provides following methods:
 - **AddSuite**
 - **SetName**
 - **SetDescription**
 - **AddTest**
 - **AddAnnotation**
 - **AddAnnotation.AddComment**
- **testCase** member object provides following methods:
 - **SetName**
 - **SetDescription**
 - **AddComment**
 - **SetStatus**



These services use the *srttest.h* header file.

4.1. Runtime Test Services APIs

srTestSuiteAddSuite

Prototype

```
srTestSuiteHandle_t srTestSuiteAddSuite(srTestSuiteHandle_t tParent,
const srCHAR * szName)
```

Description

The *srTestSuiteAddSuite()* routine is used to add a new test suite to the specified test suite.

Parameters	Type	Description
<i>tParent</i>	Input	Handle to the parent test suite to which new test suite is to be added. srTEST_SUITE_DEFAULT can be used for the default test suite.
<i>szName</i>	Input	Pointer to a null-terminated string that represents the name of test suite. If null, the default host naming scheme will be used.

Return Value	Description
<i>srTestSuiteHandle_t</i>	Handle of the newly created test suite. srTEST_SUITE_INVALID indicates failure to create test suite.

Example

```
#include "srtest.h"

void tfsuite_addSuite(void)
{
    srTestSuiteHandle_t subSuite =
        srTestSuiteAddSuite(srTEST_SUITE_DEFAULT, "tf Sub Suite");
}

#ifdef _SCL
#pragma scl_test_flist("testfunc", tfsuite_addSuite)
```

```
#endif
```

srTestSuiteSetName

Prototype

```
srWORD srTestSuiteSetName(srTestSuiteHandle t tTestSuite, const srCHAR
* szName)
```

Description

The *srTestSuiteSetName()* routine is used to set the name of the specified test suite.

Parameters	Type	Description
<i>tTestSuite</i>	Input	Handle to a test suite. srTEST_SUITE_DEFAULT can be used for the default test suite.
<i>szName</i>	Input	Pointer to a null-terminated string representing the name of test suite. Cannot be null.

Return Value	Description
<i>srOK</i>	Success
<i>srERR</i>	Null string passed in.
<i>srTEST_ERR_HANDLE_INVALID</i>	Invalid handle passed in.
<i>srTEST_WARN_STR_TRUNCATED</i>	String passed in was too large and was truncated.

Example

```
#include "srtest.h"

void tfsuite_setName(void)
{
    srTestSuiteSetName(srTEST_SUITE_DEFAULT, "Setting name for default suite");
}

#ifdef _SCL
#pragma scl_test_flist("testfunc", tfsuite_setName)
#endif
```

srTestSuiteSetDescription

Prototype

```
srWORD srTestSuiteSetDescription(srTestSuiteHandle t tTestSuite, const
srCHAR * szDescr)
```

Description

The *srTestSuiteSetDescription()* routine is used to set the description of the specified test suite.

Parameters	Type	Description
<i>tTestSuite</i>	Input	Handle to a test suite. srTEST_SUITE_DEFAULT can be used for the default test suite.
<i>szDescr</i>	Input	Pointer to a null-terminated string representing the description of test suite. Cannot be null.

Return Value	Description
<i>srOK</i>	Success
<i>srERR</i>	Null string passed in.
<i>srTEST_ERR_HANDLE_INVALID</i>	Invalid handle passed in.
<i>srTEST_WARN_STR_TRUNCATED</i>	String passed in was too large and was truncated.

Example

```
#include "srtest.h"

void tfsuite_setDescription(void)
{
    srTestSuiteSetDescription(srTEST_SUITE_DEFAULT,
                             "Setting description for default suite");
}
```



```
#ifdef _SCL  
#pragma scl_test_flist("testfunc", tfsuite_setDescription)  
#endif
```

srTestSuiteAddTest

Prototype

```
srTestCaseHandle_t srTestSuiteAddTest(srTestSuiteHandle_t tParent,  
const srCHAR * szName)
```

Description

The *srTestSuiteAddTest()* routine is used to add a new test case to the specified test suite.

Parameters	Type	Description
<i>tParent</i>	Input	Handle to the parent test suite to which new test case is to be added. srTEST_SUITE_DEFAULT can be used for the default test suite.
<i>szName</i>	Input	Pointer to a null-terminated string that represents the name of test case. If null, the default host naming scheme will be used.

Return Value

srTestCaseHandle_t

Description

Handle of the newly created test case.
srTEST_CASE_INVALID indicates failure to create test case.

Example

```
#include "srtest.h"  
#include <sstream>  
  
void tfsuite_addTest(void)  
{  
    const std::string prefix("dynamic test case ");  
    for (int count = 0; count < 5; ++count)  
    {  
        std::stringstream strm;  
        strm << prefix << count;  
  
        srTestCaseHandle_t test = srTestSuiteAddTest(srTEST_SUITE_DEFAULT,  
                                                    strm.str().c_str());  
    }  
}
```

```
        srTEST_ADD_COMMENT_WITH_LOCATION(test, "this is a dynamic test case");
    }
}

#ifdef _SCL
#pragma scl_test_flist("testfunc", tfsuite_addTest)
#endif
```

srTestCaseSetName

Prototype

```
srWORD srTestCaseSetName(srTestCaseHandle t tTestCase, const srCHAR *
szName)
```

Description

The *srTestCaseSetName()* routine is used to set the name of the specified test case.

Parameters	Type	Description
<i>tTestCase</i>	Input	Handle to a test case. srTEST_CASE_DEFAULT can be used for the default test case.
<i>szName</i>	Input	Pointer to a null-terminated string representing the name of test case. Cannot be null.

Return Value	Description
<i>srOK</i>	Success
<i>srERR</i>	Null string passed in.
<i>srTEST_ERR_HANDLE_INVALID</i>	Invalid handle passed in.
<i>srTEST_WARN_STR_TRUNCATED</i>	String passed in was too large and was truncated.

Example

```
#include "srtest.h"
#include <sstream>

void tfcase_setName(void)
{
    srTestCaseSetName(srTEST_CASE_DEFAULT, "Setting name for default case");
}

#ifdef _SCL
```

```
#pragma scl_test_flist("testfunc", tftest_setName)
#endif
```

srTestCaseSetDescription

Prototype

```
srWORD srTestCaseSetDescription(srTestCaseHandle t tTestCase, const
srCHAR * szDescr)
```

Description

The *srTestCaseSetDescription()* routine is used to set the description of the specified test case.

Parameters	Type	Description
<i>tTestCase</i>	Input	Handle to a test case. srTEST_CASE_DEFAULT can be used for the default test case.
<i>szDescr</i>	Input	Pointer to a null-terminated string representing the description of test case. Cannot be null.

Return Value	Description
<i>srOK</i>	Success
<i>srERR</i>	Null string passed in.
<i>srTEST_ERR_HANDLE_INVALID</i>	Invalid handle passed in.
<i>srTEST_WARN_STR_TRUNCATED</i>	String passed in was too large and was truncated.

Example

```
#include "srtest.h"

void tfcase_setDescription(void)
{
    srTestCaseSetDescription(srTEST_CASE_DEFAULT,
                            "Setting description for default case");
}
```

```
#ifdef _SCL  
#pragma scl_test_flist("testfunc", tfcase_setDescription)  
#endif
```

srTestCaseAddComment

Prototype

```
srWORD srTestCaseAddComment(srTestCaseHandle t tTestCase, const srCHAR  
* szFmtComment, ...)
```

Description

The *srTestCaseAddComment()* routine is used to add a comment (aka a log) to be reported with the specified test case.

Parameters	Type	Description
<i>tTestCase</i>	Input	Handle to a test case. srTEST_CASE_DEFAULT can be used for the default test case.
<i>szFmtComment</i>	Input	Pointer to a null-terminated string, which can be formatted, representing the comment. Cannot be null.
...	Input (Optional)	Variable argument list to format the comment <i>szFmtComment</i> .

Return Value	Description
<i>srOK</i>	Success
<i>srERR</i>	Null string passed in.
<i>srTEST_ERR_HANDLE_INVALID</i>	Invalid handle passed in.
<i>srTEST_WARN_STR_TRUNCATED</i>	String passed in was too large and was truncated.

Example

```
#include "srtest.h"  
  
void tfcase_addComment(void)  
{  
    srTestCaseAddComment(srTEST_CASE_DEFAULT,
```



```
        "this comment should print %s", "A STRING");  
    }  
  
#ifdef _SCL  
#pragma scl_test_flist("testfunc", tfcase_addComment)  
#endif
```

srTestCaseSetStatus

Prototype

```
srWORD srTestCaseSetStatus(srTestCaseHandle t tTestCase, srTestStatus eStatus, srDWORD dwDuration)
```

Description

The *srTestCaseSetStatus()* routine is used to set the result of test case.

Parameters	Type	Description
<i>tTestCase</i>	Input	Handle to a test case. srTEST_CASE_DEFAULT can be used for the default test case.
<i>eStatus</i>	Input	Result of the test.
<i>dwDuration</i>	Input	The duration of the test in clock ticks. Using “0” allows the STRIDE Runtime (Intercept Module) to set the time automatically.
Return Value		Description
<i>srOK</i>		Success
<i>srTEST_ERR_HANDLE_INVALID</i>		Invalid handle passed in.

4.2.

Example

```
#include "srtest.h"

void tfcase_setStatus(void)
{
    srTestCaseSetStatus(srTEST_CASE_DEFAULT, srTEST_PASS, 0);
}

#ifdef _SCL
#pragma scl_test_flist("testfunc", tfcase_setStatus)
#endif
```

srTestCaseSetStatusEx

Prototype

```
srWORD srTestCaseSetStatus(srTestCaseHandle t tTestCase, srTestStatus e
eStatus, srDWORD dwDuration, srLONG lExtendedFailureCode)
```

Description

The *srTestCaseSetStatusEx()* routine is used to set the result of test case and allow specification of an *extendedFailureCode*.

Parameters	Type	Description
<i>tTestCase</i>	Input	Handle to a test case. srTEST_CASE_DEFAULT can be used for the default test case.
<i>eStatus</i>	Input	Result of the test.
<i>dwDuration</i>	Input	The duration of the test in clock ticks. Using "0" allows the STRIDE Runtime (Intercept Module) to set the time automatically.
<i>lExtendedFailureCode</i>	Input	The Stride framework uses the <i>extendedFailureCode</i> to capture the numeric results of test method when the test method fails via a numeric (non-void, non-bool) return type.

Return Value	Description
<i>srOK</i>	Success
<i>srTEST_ERR_HANDLE_INVALID</i>	Invalid handle passed in.

4.3.

Example

```
#include "srtest.h"

void tfcase_setStatusEx(void)
{
    srTestCaseSetStatusEx(srTEST_CASE_DEFAULT, srTEST_FAIL, 0, -5);
}
```

```
}  
  
#ifdef _SCL  
#pragma scl_test_flist("testfunc", tfcase_setStatusEx)  
#endif
```

srTestSuiteAddAnnotation

Prototype

```
srTestAnnotationHandle_t srTestSuiteAddAnnotation(srTestSuiteHandle_t
tParent, srTestAnnotationLevel_e eLevel, const srCHAR * szName, const
srCHAR * szDescr)
```

Description

The *srTestSuiteAddAnnotation()* routine is used to add a new annotation to the specified test suite.

Parameters	Type	Description
<i>tParent</i>	Input	Handle to the parent test suite to which new annotation is to be added. srTEST_SUITE_DEFAULT can be used for the default test suite.
<i>eLevel</i>	Input	Annotation level. Cannot be empty.
<i>szName</i>	Input	Pointer to a null-terminated string that represents the name of annotation. If null, the default host naming scheme will be used.
<i>szDescr</i>	Input	Pointer to a null-terminated string representing the description of annotation. If null, description will be empty.

Return Value	Description
<i>srTestAnnotationHandle_t</i>	Handle of the newly created annotation. srTEST_ANNOTATION_INVALID indicates failure to create annotation.

Example

```
#include "srtest.h"
#include <sstream>

void tfsuite_addAnnotation(void)
{
    const std::string prefixName("annotation ");
    const std::string prefixDescr("description of annotation ");

    for (int count = 0; count < 5; ++count)
    {
        std::stringstream strmName;
        std::stringstream strmDescr;
        strmName << prefixName << count;
        strmDescr << prefixDescr << count;

        srTestAnnotationHandle_t annot =
            srTestSuiteAddAnnotation(srTEST_SUITE_DEFAULT,
                                    srTEST_ANNOTATION_LEVEL_ERROR,
                                    strmName.str().c_str(),
                                    strmDescr.str().c_str());
    }
}

#ifdef _SCL
#pragma scl_test_flist("testfunc", tfsuite_addAnnotation)
#endif
```

srTestAnnotationAddComment

Prototype

```
srWORD srTestAnnotationAddComment(srTestAnnotationHandle t
tTestAnnotation, const srCHAR * szLabel, const srCHAR * szFmtComment,
...)
```

Description

The *srTestAnnotationAddComment()* routine is used to add a comment (aka a log) to be reported with the specified annotation.

Parameters	Type	Description
<i>tTestAnnotation</i>	Input	Handle to an annotation created using srTestSuiteAddAnnotation .
<i>szLabel</i>	Input	Pointer to a null-terminated string for the label. If null, the default label will be applied.
<i>szFmtComment</i>	Input	Pointer to a null-terminated string, which can be formatted, representing the comment. Cannot be null.
...	Input (Optional)	Variable argument list to format the comment <i>szFmtComment</i> .

Return Value	Description
<i>srOK</i>	Success
<i>srERR</i>	Null string passed in.
<i>srTEST_ERR_HANDLE_INVALID</i>	Invalid handle passed in.
<i>srTEST_WARN_STR_TRUNCATED</i>	String passed in was too large and was truncated.

Example

```
#include "srtest.h"

void tfsuiteAnnotation _ addComment(void)
{
    srTestAnnotationHandle_t annot =
        srTestSuiteAddAnnotation(srTEST_SUITE_DEFAULT,
                                srTEST_ANNOTATION_LEVEL_ERROR,
                                "annot",
                                "annot description");

    srTestAnnotationAddComment(annot,
                                srNULL,
                                "this comment should print %s", "A STRING");
}

#ifdef _SCL
#pragma scl_test_flist("testfunc", tfsuiteAnnotation_addComment)
#endif
```


4.3.1. Messages to Support Runtime Test Services

srTEST_SUITE_ADD_SUITE_MSG

The srTEST_SUITE_ADD_SUITE_MSG message is a broadcast message used to send the host information required for adding a new test suite.

Attributes	Description
<i>srMT_BRD</i>	Broadcast message
<i>srST_RSP_VAL</i>	Response payload sent by value - srTestSuiteAddSuite_t

Response Fields	Description
parentTestSuiteID (srDWORD)	Test suite ID to which to add the new test.
subTestSuiteID (srDWORD)	Sequentially generated test suite ID for the new suite.
name (srCHAR *)	String to use for test suite name, up to srTEST_MAX_STRING_LENGTH bytes. If NULL or empty, a default suite name will be assigned by the host.

srTEST_SUITE_SET_NAME_MSG

The srTEST_SUITE_SET_NAME_MSG message is a broadcast message used to change the name of a test suite on the host.

Attributes

srMT_BRD

srST_RSP_VAL

Description

Broadcast message

Response payload sent by value -
srTestSuiteSetName_t

Response Fields

testSuiteID (srDWORD)

name (srCHAR *)

Description

Test suite ID whose name is to be changed.

String to use for test suite name, up to srTEST_MAX_STRING_LENGTH bytes. If NULL or empty, the host will ignore this message (name will not be changed).

srTEST_SUITE_SET_DESCRIPTION_MSG

The srTEST_SUITE_SET_DESCRIPTION_MSG message is a broadcast message used to change the description of a test suite on the host.

Attributes

srMT_BRD

Description

Broadcast message

srST_RSP_VAL

Response payload sent by value -
srTestSuiteSetDescription_t

Response Fields

testSuiteID (srDWORD)

Description

Test suite ID whose description is to be changed.

description (srCHAR *)

String to use for suite description, up to srTEST_MAX_STRING_LENGTH bytes. If NULL or empty, the host will ignore this message (description will not be changed).

srTEST_SUITE_ADD_TEST_MSG

The srTEST_SUITE_ADD_TEST_MSG message is a broadcast message used to create a new test case.

Attributes

srMT_BRD

srST_RSP_VAL

Description

Broadcast message

Response payload sent by value -
srTestSuiteAddTest_t

Response Fields

parentTestSuiteID (srDWORD)

testCaseID (srDWORD)

name (srCHAR *)

Description

Test suite ID to which this test case should be added.

Sequentially generated test case ID for the new test.

String to use for test case name, up to srTEST_MAX_STRING_LENGTH bytes. If NULL or empty, a default test case name will be assigned by the host.

srTEST_CASE_SET_NAME_MSG

The srTEST_CASE_SET_NAME_MSG message is a broadcast message used to change the name of a test case on the host.

Attributes

srMT_BRD

Description

Broadcast message

srST_RSP_VAL

Response payload sent by value -
srTestCaseSetName_t

Response Fields

testCaseID (srDWORD)

Description

Test case ID whose name is to be changed.

name (srCHAR *)

String to use for test case name, up to srTEST_MAX_STRING_LENGTH bytes. If NULL or empty, the host will ignore this message (name will not be changed).

srTEST_CASE_SET_DESCRIPTION_MSG

The srTEST_CASE_SET_DESCRIPTION_MSG message is a broadcast message used to change the description of a test case on the host.

Attributes

srMT_BRD

srST_RSP_VAL

Description

Broadcast message

Response payload sent by value -
srTestCaseSetDescription_t

Response Fields

testCaseID (srDWORD)

description (srCHAR *)

Description

Test case ID whose description is to be changed.

String to use for test case description, up to srTEST_MAX_STRING_LENGTH bytes. If NULL or empty, the host will ignore this message (description will not be changed).

srTEST_CASE_ADD_COMMENT_MSG

The srTEST_CASE_ADD_COMMENT_MSG message is a broadcast message used to add a comment to the specified test case on the host.

Attributes

srMT_BRD

Description

Broadcast message

srST_RSP_VAL

Response payload sent by value -
`srTestCaseAddComment_t`

Response Fields

`testCaseID (srDWORD)`

Description

Test case ID to which the comment is to be added.

`comment (srCHAR *)`

String to use for the comment, up to `srTEST_MAX_STRING_LENGTH` bytes. If NULL or empty, the host will ignore this message (no comment will be added).

srTEST_CASE_SET_STATUS_MSG

The srTEST_CASE_SET_STATUS_MSG message is a broadcast message used to set the status for a test case on the host.

Attributes

srMT_BRD

Description

Broadcast message

srST_RSP_VAL

Response payload sent by value -
srTestCaseSetStatus_t

Response Fields

testCaseID (srDWORD)

Description

Test case ID for which to set the status.

status (srTestStatus_t)

The status value to assign to the test case.

duration (srDWORD)

Duration value for the test case in clock ticks.

fromIM (srBOOL)

Boolean flag to indicate whether the status message was generated by the IM. If generated by the IM and the host has already previously gotten a status message for the given test case, the host must ignore this message (that is, not update the status). In other words, status updates from the IM cannot preempt status values set by test code.

srTEST_METHOD_START_MSG

The srTEST_METHOD_START_MSG message is a broadcast message used to inform the host that the IM is about to run a test method.

Attributes*srMT_BRD***Description**

Broadcast message

*srST_RSP_VAL*Response payload sent by value -
srTestMethodStart_t**Response Fields**

testUnitID (srDWORD)

Description

The SMID that identifies the current Test Unit.

methodName (srCHAR *)

The name of the method.

srTEST_SUITE_ADD_ANNOTATION_MSG

The srTEST_SUITE_ADD_ANNOTATION_MSG message is a broadcast message used to create a new annotation.

Attributes	Description
<i>srMT_BRD</i>	Broadcast message
<i>srST_RSP_VAL</i>	Response payload sent by value - <code>srTestSuiteAddAnnotation_t</code>

Response Fields	Description
parentTestSuiteID (srDWORD)	Test suite ID to which this annotation should be added.
testAnnotation ID (srDWORD)	Sequentially generated annotation ID for the new annotation.
Level (srCHAR *)	String to use for annotation level., up to srTEST_MAX_STRING_LENGTH bytes. Cannot be NULL.
name (srCHAR *)	String to use for annotation name, up to srTEST_MAX_STRING_LENGTH bytes. If NULL or empty, a default name will be assigned by the host.
description (srCHAR *)	String to use for description of annotation, up to srTEST_MAX_STRING_LENGTH bytes. If NULL or empty, description will be empty.

srTEST_ANNOTATION_ADD_COMMENT_MSG

The srTEST_ANNOTATION_ADD_COMMENT_MSG message is a broadcast message used to add a comment to the specified annotation on the host.

Attributes

srMT_BRD

Description

Broadcast message

srST_RSP_VAL

Response payload sent by value -
`srTestAnnotationAddComment_t`

Response Fields

testAnnotationID (srDWORD)

Description

Annotation ID to which the comment is to be added.

label (srCHAR *)

String to use for the label, up to
srTEST_MAX_STRING_LENGTH bytes.
If NULL or empty, the label will be empty.

comment (srCHAR *)

String to use for the comment, up to
srTEST_MAX_STRING_LENGTH bytes.
If NULL or empty, the host will ignore this
message (no comment will be added).

4.4. Runtime Base Class (srTest)

The `srTest` base class provides two member objects to access the functionality of Runtime Test Services.

testSuite

Type

```
srTestSuite testSuite
```

Description

The *testSuite* is a protected member object of `srTest` class representing the default test suite.

testCase

Type

```
srTestCase testCase
```

Description

The *testCase* is a protected member object of `srTest` class representing the default test case.

4.4.1. Methods of Member Object testSuite

AddSuite

Prototype

```
srTestSuite AddSuite(const srCHAR * szName = srNULL)
```

Description

The *AddSuite* method is used to add a new test suite.

Parameters	Type	Description
<i>szName</i>	Input (Optional)	Pointer to null-terminated string that represents the name of test suite. If empty, the default host naming scheme will be used.

Return Value	Description
<i>srTestSuite</i>	Newly created test suite class.

Example

```
#include "srtest.h"

class srtest_class : public stride::srTest
{
public:
    void suiteAddSuite(void);
};

void srtest_class::suiteAddSuite(void)
{
    stride::srTestSuite suite = testSuite.AddSuite("tc Sub Suite");
}

#ifdef _SCL
#pragma scl_test_class(srtest_class)
#endif
```

SetName

Prototype

```
srWORD SetName(const srCHAR * szName)
```

Description

The *SetName* method is used to set the name of test suite.

Parameters	Type	Description
<i>szName</i>	Input	Pointer to null-terminated string representing the name of test suite. Cannot be empty.

Return Value	Description
<i>srOK</i>	Success
<i>srERR</i>	Null string passed in.
<i>srTEST_ERR_HANDLE_INVALID</i>	Invalid handle passed in.
<i>srTEST_WARN_STR_TRUNCATED</i>	String passed in was too large and was truncated.

Example

```
#include "srtest.h"

class srtest_class : public stride::srTest
{
public:
    void suiteSetName(void);
};

void srtest_class::suiteSetName(void)
{
    testSuite.SetName("Setting name for suite");
}

#ifdef _SCL
#pragma scl_test_class(srtest_class)
#endif
```

SetDescription

Prototype

```
srWORD SetDescription(const srCHAR * szDescr)
```

Description

The *SetDescription* method is used to set the description of test suite.

Parameters	Type	Description
<i>szDescr</i>	Input	Pointer to null-terminated string representing the description of test suite. Cannot be empty.

Return Value	Description
<i>srOK</i>	Success
<i>srERR</i>	Null string passed in.
<i>srTEST_ERR_HANDLE_INVALID</i>	Invalid handle passed in.
<i>srTEST_WARN_STR_TRUNCATED</i>	String passed in was too large and was truncated.

Example

```
#include "srtest.h"

class srtest_class : public stride::srTest
{
public:
    void suiteSetDescription(void);
};

void srtest_class::suiteSetDescription(void)
{
    testSuite.SetDescription("Setting description for suite");
}

#ifdef _SCL
#pragma scl_test_class(srtest_class)
#endif
```

AddTest

Prototype

```
srTestCase AddTest(const srCHAR * szName = srNULL)
```

Description

The *AddTest* method is used to add a new test case to test suite.

Parameters	Type	Description
<i>szName</i>	Input (Optional)	Pointer to null-terminated string that represents the name of test case. If empty, the default host naming scheme will be used.
Return Value	Description	
<i>srTestCase</i>	Newly created test case class.	

Example

```
#include "srtest.h"
#include <sstream>

class srtest_class : public stride::srTest
{
public:
    void suiteAddTest(void);
};

void srtest_class::suiteAddTest(void)
{
    const std::string prefix("dynamic test case ");
    for (int count = 0; count < 5; ++count)
    {
        std::stringstream strm;
        strm << prefix << count;
        stride::srTestCase tc = testSuite.AddTest(strm.str().c_str());
        tc.AddComment("this is a dynamic test case");
        tc.SetStatus(srTEST_PASS);
    }
}

#ifdef _SCL
#pragma scl_test_class(srtest_class)
```



```
#endif
```

AddAnnotation

Prototype

```
srTestAnnotation AddAnnotation(srTestAnnotationLevel_e eLevel,  
                               const srCHAR * szName = srNULL,  
                               const srCHAR * szDescr = srNULL)
```

Description

The *AddAnnotation* method is used to add a new annotation to test suite.

Parameters	Type	Description
<i>eLevel</i>	Input	Annotation level. Cannot be empty.
<i>szName</i>	Input (Optional)	Pointer to null-terminated string that represents the name of annotation. If empty, the default host naming scheme will be used.
<i>szDescr</i>	Input (Optional)	Pointer to null-terminated string that represents the description of annotation. If empty, the description will be blank.
Return Value		Description
<i>srTestAnnotation</i>		Newly created annotation class.

Example

```
#include "srtest.h"  
#include <sstream>  
  
class srtest_class : public stride::srTest  
{  
public:  
    void suiteAddAnnotation(void);  
};  
  
void srtest_class::suiteAddAnnotation(void)  
{  
    const std::string prefixName("annotation ");  
    const std::string prefixDescr("description of annotation ");
```

```
for (int count = 0; count < 5; ++count)
{
    std::stringstream strmName;
    std::stringstream strmDescr;
    strmName << prefixName << count;
    strmDescr << prefixDescr << count;

    stride::srTestAnnotation ta =
        testSuite.AddAnnotation(srTEST_ANNOTATION_LEVEL_INFO,
                                strmName.str().c_str(),
                                strmDescr.str().c_str());
}

#ifdef _SCL
#pragma scl_test_class(srtest_class)
#endif
```

AddAnnotation.AddComment

Prototype

```
srWORD AddComment(const srCHAR * szLabel, const srCHAR * szFmtComment,  
...)
```

Description

The *AddComment* method is used to add a comment to an annotation created under a test suite.

Parameters	Type	Description
<i>szLabel</i>	Input	Pointer to null-terminated string for the label. If null, the default label will be applied.
<i>szFmtComment</i>	Input	Pointer to null-terminated string, which can be formatted, representing the comment. Cannot be empty.
...	Input (Optional)	Variable argument list to format the comment <i>szFmtComment</i> .

Return Value	Description
<i>srOK</i>	Success
<i>srERR</i>	Null string passed in.
<i>srTEST_ERR_HANDLE_INVALID</i>	Invalid handle passed in.
<i>srTEST_WARN_STR_TRUNCATED</i>	String passed in was too large and was truncated.

Example

```
#include "srtest.h"  
  
class srtest_class : public stride::srTest  
{  
public:  
    void suiteAnnotationAddComment(void);  
};  
  
void srtest_class:: suiteAnnotationAddComment(void)  
{
```

```
stride::srTestAnnotation ta =
    testSuite.AddAnnotation(srTEST_ANNOTATION_LEVEL_INFO,
        "annot name",
        "annot description");

    ta.AddComment(srNULL,
        "this comment on annotation should print %s", "A STRING");
}

#ifdef _SCL
#pragma scl_test_class(srtest_class)
#endif
```

4.4.2. Methods of Member Object `testCase`

SetName

Prototype

```
srWORD SetName(const srCHAR * szName)
```

Description

The *SetName* method is used to set the name of test case.

Parameters	Type	Description
<i>szName</i>	Input	Pointer to null-terminated string representing the name of test case. Cannot be empty.

Return Value	Description
<i>srOK</i>	Success
<i>srERR</i>	Null string passed in.
<i>srTEST_ERR_HANDLE_INVALID</i>	Invalid handle passed in.
<i>srTEST_WARN_STR_TRUNCATED</i>	String passed in was too large and was truncated.

Example

```
#include "srtest.h"

class srtest_class : public stride::srTest
{
public:
    void caseSetName(void);
};

void srtest_class::caseSetName(void)
{
    testCase.SetName("Setting name for case");
}
#ifdef _SCL
#pragma scl_test_class(srtest_class)
#endif
```

SetDescription

Prototype

```
srWORD SetDescription(const srCHAR * szDescr)
```

Description

The *SetDescription* method is used to set the description of test case.

Parameters	Type	Description
<i>szDescr</i>	Input	Pointer to null-terminated string representing the description of test case. Cannot be empty.

Return Value	Description
<i>srOK</i>	Success
<i>srERR</i>	Null string passed in.
<i>srTEST_ERR_HANDLE_INVALID</i>	Invalid handle passed in.
<i>srTEST_WARN_STR_TRUNCATED</i>	String passed in was too large and was truncated.

Example

```
#include "srtest.h"

class srtest_class : public stride::srTest
{
public:
    void caseSetDescription(void);
};

void srtest_class::caseSetDescription(void)
{
    testCase.SetDescription("Setting description for case");
}

#ifdef _SCL
#pragma scl_test_class(srtest_class)
#endif
```

AddComment

Prototype

```
srWORD AddComment(const srCHAR * szFmtComment, ...)
```

Description

The *AddComment* method is used to add a comment to test case.

Parameters	Type	Description
<i>szFmtComment</i>	Input	Pointer to null-terminated string, which can be formatted, representing the comment. Cannot be empty.
...	Input (Optional)	Variable argument list to format the comment <i>szFmtComment</i> .

Return Value	Description
<i>srOK</i>	Success
<i>srERR</i>	Null string passed in.
<i>srTEST_ERR_HANDLE_INVALID</i>	Invalid handle passed in.
<i>srTEST_WARN_STR_TRUNCATED</i>	String passed in was too large and was truncated.

Example

```
#include "srtest.h"

class srtest_class : public stride::srTest
{
public:
    void caseAddComment(void);
};

void srtest_class::caseAddComment(void)
{
    testCase.AddComment("this comment should print %s", "A STRING");
}

#ifdef _SCL
```



```
#pragma scl_test_class(srtest_class)
#endif
```

SetStatus

Prototype

```
srWORD SetStatus(srTestStatus_e eStatus, srDWORD dwDuration = 0)
```

Description

The *SetStatus* method is used to set the result of the default test case.

Parameters	Type	Description
<i>eStatus</i>	Input	Result of the test.
<i>dwDuration</i>	Input (Optional)	The duration of the test in clock ticks. The default is 0.

Return Value	Description
<i>srOK</i>	Success
<i>srTEST_ERR_HANDLE_INVALID</i>	Invalid handle.

Example

```
#include "srtest.h"

class srtest_class : public stride::srTest
{
public:
    void caseSetStatus(void);
};

void srtest_class::caseSetStatus(void)
{
    testCase.SetStatus(srTEST_INPROGRESS, 0);
}

#ifdef _SCL
#pragma scl_test_class(srtest_class)
#endif
```

5. STRIDE Runtime Internals

The STRIDE Runtime is organized with one main procedure that can be called from the Runtime Thread (or any other thread) and a set of modules that are called out of the contents of the caller.

5.1. STRIDE Runtime Thread and Procedure

The STRIDE Runtime Thread consists of an endless loop which waits on the `palWait` routine and calls the STRIDE Runtime procedure when notified. The Runtime procedure contains logic for maintaining the flow of trace buffers and reading received messages. These messages control the sending and receiving of I-blocks, setting of tracing filters, and other Runtime functions.

There are three main message loops. One is used for messages that will generate outgoing I-blocks. These messages are not read until the I-block Ready signal has been set. This is useful for allowing control of the transmission of I-blocks. The second message loop handles the rest of the messages that the Runtime procedure receives. The third message loop is used for receiving fragmented I-blocks. Once the first fragment of an I-block is received the remaining fragments are collected in the same message loop.

5.2. STRIDE Runtime Modules

The STRIDE Runtime is organized into nine functional modules. Each module is organized as a set of files with a common file name prefix. Modules do not share memory directly. Functions that are used outside of a file are exported by the use of a module header file. These module header files only contain the information that is shared outside the file.

Module Description	File Prefix
Connection Routines	srconn
Error Routines	srerr
I-block Routines	srib
Memory Management Routines	srmem
Message Routines	srmsg
STID Routines	srstid
SUID Routines	srsuid
Test Services	srtest
Thread Routines	srthread
Timer Routines	srttime

Any interface exported through the module header file (either a function prototype, define or structure) will contain an underscore and the filename as the first characters of the name. The rest of the name will be descriptive of the function or define.

5.3. STRIDE Runtime Files

Below is a list of the STRIDE Runtime files provided in the STRIDE ITE.

File	Description
sr.h	Public STRIDE prototypes and defines
srapi.c	Public API source code
srapiogl.c	Regional API source code
srapiogl.h	Regional API prototypes and defines
srcfg.h	STRIDE configuration file
srcgutil.c	Code generation utility routines
srcgutil.h	Code generation utility prototypes and defines
srconn.c	Connection source code
srconn.h	Connection prototypes and defines
srerr.c	Error reporting source code
srerr.h	Error reporting prototypes and defines
srib.c	Main I-block source code
srib.h	Main I-block prototypes and defines
sribctrl.c	Control I-block source code
sribctrl.h	Control I-block prototypes and defines
sribmsg.c	Message I-block source code
sribmsg.h	Message I-block prototypes and defines
sribrpt.c	Report I-block source code
sribrpt.h	Report I-block prototypes and defines
sribtr.c	Trace I-block routines
sribtr.h	Trace I-block prototypes and defines
srmem.c	Memory management source code
srmem.h	Memory management prototypes and defines
srmsg.h	STRIDE Runtime message interface defines
srmsgbox.c	Message Box source code
srmsgbox.h	Message Box prototypes and defines
srmsgmar.c	Message Marshaling source code
srmsgmar.h	Message Marshaling prototypes and defines
srmsgptr.c	Pointer module source code
srmsgptr.h	Pointer module prototypes and defines

srmsgque.c	Message Queueing source code
srmsgque.h	Message Queueing prototypes and defines
srmsgprt.c	Message Routing source code
srmsgprt.h	Message Routing prototypes and defines
srmsgsub.c	Message Subscribing source code
srmsgsub.h	Message Subscribing prototypes and defines
srstid.c	STID source code
srstid.h	STID prototypes and defines
rsuid.c	SUID source code
rsuid.h	SUID prototypes and defines
srthread.c	Runtime Thread source code
srthread.h	Runtime Thread source code prototypes and defines
srttime.c	Timer source code
srttime.h	Timer prototypes and defines
srtp.h	Trace Point defines
srtest.h	Runtime Test Services (RTS) C APIs and C++ base class (Optional)
srtest.c	Runtime Test Services (RTS) C APIs source code (Optional)
srtest.cpp	Runtime Test Services (RTS) C++ base class source code (Optional)
srtestutil.c	Runtime Test Services (RTS) utility routines (Optional)
srtestutil.h	Runtime Test Services (RTS) utility prototypes and defines (Optional)

Appendix A: STRIDE Runtime API (sr.h)

```
/*
 *
 * FILE NAME: sr.h
 *
 * DESCRIPTION:
 *   STRIDE Runtime (SR) public API prototypes, structures and defines.
 *
 * -----
 * Copyright 2001 - 2008 by S2 Technologies, Inc.
 * -----
 */

#ifndef SR_H
#define SR_H

#ifdef __cplusplus
extern "C" {
#endif

#include "pal.h"

#define srEXTERN extern

#define srEXPORT
#define srSTDCALL

/*
 *
 * Runtime Primitive Types
 *
 */

typedef palCHAR          srCHAR;
typedef palBYTE          srBYTE;
typedef palSHORT         srSHORT;
typedef palWORD          srWORD;
typedef palLONG          srLONG;
typedef palDWORD         srDWORD;
typedef palBOOL          srBOOL;

#define srTRUE           palTRUE
#define srFALSE          palFALSE
#define srNULL           palNULL

/*
 *
 * SMID Attributes
 *
 */

/* Message Types (MT) */
#define srMT_ONE_CMD     0x00000000
#define srMT_ONE_RSP     0x01000000
#define srMT_TWO        0x02000000
#define srMT_BRD         0x03000000

/* Abbreviations MT */
#define srMT_ONE         srMT_ONE_CMD
#define srMT_ONEc        srMT_ONE_CMD
#define srMT_ONEr        srMT_ONE_RSP

/* Send Type for Command (ST_CMD) */
#define srST_CMD_PTR     0x00000000
#define srST_CMD_VAL     0x04000000
```

```

#define srST_CMD_NOP    0x00000000 /* Deprecated */

/* Send Type for Response (ST_RSP) */
#define srST_RSP_PTR    0x00000000
#define srST_RSP_VAL    0x08000000
#define srST_RSP_NOP    0x00000000 /* Deprecated */

/* Pointer Usage for Command (PU_CMD) */
#define srPU_CMD_POL    0x00000000
#define srPU_CMD_PRI    0x10000000

/* Pointer Usage for Response (PU_RSP) */
#define srPU_RSP_POL    0x00000000
#define srPU_RSP_PRI    0x20000000

/* Access Class (AC) */
#define srAC_MSG        0x00000000
#define srAC_FUNCTION    0x40000000
#define srAC_SYS        0x80000000

/*****
 *
 * Runtime API Return codes
 *
 *****/

#define srOK                0
#define srERR                1
#define srERR_PAL_MEM_ALLOC  2
#define srERR_PAL_NOTIFY_SYS 3
#define srERR_PAL_NOTIFY_USER 4
#define srERR_PAL_OUT        5
#define srERR_SMID_ATTR      6
#define srERR_SUID_RANGE     7
#define srERR_STID_INVALID   8
#define srERR_STID_INACTIVE  9
#define srERR_STID_ALLOC     10
#define srERR_STID_MAX       11
#define srERR_STID_USED      12
#define srERR_QUEUE_FULL    13
#define srERR_QUEUE_EMPTY   14
#define srERR_PTR_INVALID    15
#define srERR_PTR_ALLOC      16
#define srERR_PTR_TEARDOWN   17
#define srERR_PTR_DUPLICATE  18
#define srERR_PTR_LOCKED     19
#define srERR_PTR_POOL       20
#define srERR_PTR_OVERWRITE  21
#define srERR_PTR_ADDRESS    22
#define srERR_PTR_OFFSET     23
#define srERR_PTR_MSG_DIR    24
#define srERR_PTR_DIR        25
#define srERR_PTR_USAGE      26
#define srERR_PTR_SIZE       27
#define srERR_REG_SET        28
#define srERR_REG_OVERRIDE_SET 29
#define srERR_REG_STORAGE_FULL 30
#define srERR_REG_NONE       31
#define srERR_SUB_ALLOC      32
#define srERR_SUB_NONE       33
#define srERR_RMT_FAIL       34
#define srERR_SEND_PRIV      35
#define srERR_READ_SIZE     36
#define srERR_TRACE_TYPE     37
#define srERR_MAP_STATUS     38
#define srERR_MAP_FAIL       39
#define srERR_RT_INITIALIZE_NONE 40
#define srERR_LAST_CODE     41

```



```
/* *****  
 *  
 * STRIDE Runtime API Enums and defines  
 *  
 * ***** */  
  
/* Mail Box Ids */  
typedef enum  
{  
    srBOX_1 = 0,  
    srBOX_2 = 1,  
    srBOX_3 = 2,  
    srBOX_4 = 3,  
    srBOX_5 = 4,  
    srBOX_6 = 5,  
    srBOX_7 = 6,  
    srBOX_8 = 7  
} srBOX_e;  
  
#define srBOX_MAX 8  
  
/* TracePoint Levels */  
typedef enum  
{  
    srLEVEL_0 = 0,  
    srLEVEL_1 = 1,  
    srLEVEL_2 = 2,  
    srLEVEL_3 = 3,  
    srLEVEL_4 = 4,  
    srLEVEL_5 = 5,  
    srLEVEL_6 = 6,  
    srLEVEL_7 = 7  
} srLevel_e;  
  
/* Embedded Pointer Msg Direction */  
typedef enum  
{  
    srMSGDIR_COMMAND = 0,  
    srMSGDIR_RESPONSE = 1  
} srMsgDir_e;  
  
/* Embedded Pointer Direction */  
typedef enum  
{  
    srPTRDIR_IN = 0,  
    srPTRDIR_OUT = 1,  
    srPTRDIR_INOUT = 2,  
    srPTRDIR_RET = 3  
} srPtrDir_e;  
  
#define srPTRDIR_CMD_IN    srPTRDIR_IN  
#define srPTRDIR_CMD_OUT  srPTRDIR_OUT  
#define srPTRDIR_CMD_INOUT srPTRDIR_INOUT  
#define srPTRDIR_CMD_RET  srPTRDIR_RET  
#define srPTRDIR_RSP_RET  srPTRDIR_RET  
  
/* Embedded Pointer Usage */  
typedef enum  
{  
    srPTR_USAGE_PRIVATE = 0,  
    srPTR_USAGE_POOL = 1  
} srPtrUsage_e;  
  
/* Trace Type */  
typedef enum  
{
```

```
    srTRACE_SEND_CMD      = 0,
    srTRACE_READ_CMD     = 1,
    srTRACE_SEND_RSP     = 2,
    srTRACE_READ_RSP     = 3,
    srTRACE_SEND_BCAST   = 4,
    srTRACE_READ_BCAST   = 5,
    srTRACE_CALL         = 6,
    srTRACE_RETURN       = 7
} srTraceType_e;

/* Access Class Registration */
typedef enum
{
    srAC_REG_MESSAGES = 0,
    srAC_REG_FUNCTIONS = 1
} srAccessClass_e;

/* pointer setup offset macros */
#define srPTR_OFFSET( Base, Offset )      ( (srWORD)((srDWORD)&Base.Offset -
(srDWORD)&Base) )
#define srPTR_OFFSET_BYREF( Base, Offset ) ( (srWORD)((srDWORD)&Base->Offset -
(srDWORD)Base) )

/* pointer setup handle is empty */
#define srPTR_EMPTY 0x0FFF

/* used by trace interface to indicate empty pointer instance */
#define srPTR_INST_EMPTY 0x0FFF0000

/* srCreateSTID NID - indicates that NID is not used */
#define srNID_NONE palNID_RESERVED_0

/* NOTE: STID==0 is for private runtime use only */
#define srSTID_RESERVED 0

/* max size of any trace string */
#define srTRACE_STR_MAX 255

/* pool memory management prototypes */
typedef void* (*srPoolMemAlloc_t)(srWORD wSize);
typedef void (*srPoolMemFree_t)(void* pvMemory);

/*****
 *
 * STRIDE Runtime API Structures
 *
 *****/

/* Query SMID Info */
typedef struct
{
    srBOOL    bReg;
    srBYTE    yRegSTID;
    srBOX_e   eRegBox;
    srBOOL    bRegOverride;
    srWORD    wSubIdsCnt;
} srSMIDInfo_t;

/* Query Box Info */
typedef struct
{
```

```

    srDWORD   dwNextSMID;
    srWORD    wNumPending;
    srWORD    wSizeOfNextSMID;
} srBoxInfo_t;

/*****
 *
 * Setup & Shutdown API
 *
 *****/

/**
 * Initialize the STRIDE Runtime. This should be called once per process, at startup.
 * @return srOK on success, stride error code otherwise
 */
srEXPORT srWORD srSTDCALL srInit( void );

/**
 * Uninitialize the STRIDE Runtime. This should be called only once per process, at
 shutdown.
 * @return srOK on success, stride error code otherwise
 */
srEXPORT srWORD srSTDCALL srUninit( void );

srEXPORT srWORD srSTDCALL srCreateSTID( srDWORD   dwNID,
                                       srCHAR     *szName,
                                       srWORD     *pwSTID,
                                       srBOOL     bNew );

srEXPORT srWORD srSTDCALL srDeleteSTID( srWORD   wSTID );

/*****
 *
 * Messaging API
 *
 *****/

srEXPORT srWORD srSTDCALL srRegister( srWORD     wSTID,
                                       srBOX_e    eBox,
                                       srDWORD    dwSMID,
                                       srBOOL     bOn );

srEXPORT srWORD srSTDCALL srRead( srWORD     wSTID,
                                   srBOX_e    eBox,
                                   srWORD     wMaxRead,
                                   srDWORD    *pdwSMID,
                                   srBYTE     *pyBuffer,
                                   srWORD     *pwSize,
                                   srDWORD    *pdwMsgInst );

srEXPORT srWORD srSTDCALL srReadComplete( srWORD     wSTID,
                                           srDWORD    dwMsgInst );

srEXPORT srWORD srSTDCALL srSendCmd( srWORD     wSTID,
                                       srBOX_e    eRspBox,
                                       srDWORD    dwSMID,
                                       srBYTE     *pyPayload,
                                       srWORD     wSize );

srEXPORT srWORD srSTDCALL srSendRsp( srWORD     wSTID,
                                       srDWORD    dwMsgInst,
                                       srDWORD    dwSMID,
                                       srBYTE     *pyPayload,
                                       srWORD     wSize );

srEXPORT srWORD srSTDCALL srBroadcast( srWORD     wSTID,
                                       srDWORD    dwSMID,
                                       srBYTE     *pyPayload,
                                       srWORD     wSize );

```

```

srEXPORT srWORD srSTDCALL srSubscribe( srWORD    wSTID,
                                       srWORD_e   eBox,
                                       srDWORD    dwSMID,
                                       srBOOL     bOn );

srEXPORT srWORD srSTDCALL srSetAuxData( srWORD    wSTID,
                                       srDWORD    dwAuxData );

srEXPORT srWORD srSTDCALL srGetAuxData( srWORD    wSTID,
                                       srDWORD    *pdwAuxData );

/*****
 *
 * Pointer API
 *
 *****/

srEXPORT srWORD srSTDCALL srPtrSetup( srWORD      wSTID,
                                       srDWORD     dwSMID,
                                       srWORD      wOffset,
                                       srWORD      *pwOffsetTable,
                                       srWORD      wSize,
                                       srMsgDir_e  eMsgDir,
                                       srPtrDir_e  ePtrDir,
                                       srPtrUsage_e ePtrUsage,
                                       srWORD      *pwHandle );

srEXPORT srWORD srSTDCALL srPtrSetupChild( srWORD      wSTID,
                                           srWORD      wParentHandle,
                                           srWORD      wOffset,
                                           srWORD      wSize,
                                           srPtrDir_e  ePtrDir,
                                           srPtrUsage_e ePtrUsage,
                                           srWORD      *pwHandle );

srEXPORT srWORD srSTDCALL srPtrTearDown( srWORD    wSTID,
                                          srDWORD    dwSMID,
                                          srWORD    wHandle );

srEXPORT srWORD srSTDCALL srPtrGetHandle( srWORD    wSTID,
                                          srDWORD    dwMsgInst,
                                          srBYTE     *pyMemory,
                                          srWORD     *pwHandle );

srEXPORT srWORD srSTDCALL srPtrSize( srWORD    wSTID,
                                      srWORD    wHandle,
                                      srWORD    wSize );

srEXPORT srWORD srSTDCALL srPtrCreateCmdInst( srWORD    wSTID,
                                               srDWORD    dwSMID,
                                               srBYTE     *pyPayload,
                                               srWORD     wSize,
                                               srDWORD    *pdwCmdInst );

srEXPORT srWORD srSTDCALL srPtrCreateRspInst( srWORD    wSTID,
                                              srDWORD    dwSMID,
                                              srBYTE     *pyPayload,
                                              srWORD     wSize,
                                              srDWORD    dwCmdInst,
                                              srDWORD    *pdwRspInst );

srEXPORT srWORD srSTDCALL srPtrDeleteInst( srWORD    wSTID,
                                           srDWORD    dwPtrInst );

/*****
 *
 * Tracing API
 *
 *****/

```

```

*****/
srEXPORT srWORD srSTDCALL srTracePoint( srWORD      wSTID,
                                         srDWORD     dwTPID,
                                         srBYTE      *pyPayload,
                                         srWORD      wSize,
                                         srLevel_e    eLevel );

srEXPORT srWORD srSTDCALL srTraceStr( srWORD      wSTID,
                                       srCHAR      *szString,
                                       srLevel_e    eLevel );

srEXPORT srWORD srSTDCALL srTraceInterface( srWORD      wSTID,
                                             srDWORD     dwSUID,
                                             srBYTE      *pyPayload,
                                             srWORD      wSize,
                                             srDWORD     dwPtrInst,
                                             srTraceType_e eTraceType );

/*****
 *
 *  Printing API
 *
 *****/
srEXPORT srWORD srSTDCALL srPrintInfo(const srCHAR * szMsg, ...);
srEXPORT srWORD srSTDCALL srPrintError(const srCHAR * szMsg, ...);

/*****
 *
 *  Query API
 *
 *****/
srEXPORT srWORD srSTDCALL srQueryNID( srDWORD     dwNID,
                                       srWORD      *pwSTID );

srEXPORT srWORD srSTDCALL srQueryName( srCHAR      *szName,
                                       srWORD      *pwSTID );

srEXPORT srWORD srSTDCALL srQuerySMID( srDWORD     dwSMID,
                                       srSMIDInfo_t *ptSMIDInfo );

srEXPORT srWORD srSTDCALL srQueryBox( srWORD      wSTID,
                                       srBOX_e     eBox,
                                       srBoxInfo_t  *ptBoxInfo );

srEXPORT srWORD srSTDCALL srQuerySTID( srDWORD     *pdwNID,
                                       srWORD      wSTID );

srEXPORT srWORD srSTDCALL srQueryAccessClass( srAccessClass_e eAC,
                                              srWORD      *pwSTID,
                                              srBOX_e     *peBox );

/*****
 *
 *  Remote Message Stub/Proxy API
 *
 *****/

/* Deprecated in favor of srRegisterAccessClass() */
srEXPORT srWORD srSTDCALL srSetRMS( srWORD      wSTID,
                                    srBOX_e     eBox );

srEXPORT srWORD srSTDCALL srRegisterAccessClass( srAccessClass_e eAC,
                                                  srWORD      wSTID,
                                                  srBOX_e     eBox,
                                                  srBOOL      bOn );

```

```

/*****
 *
 * Runtime Thread Entry / Exit Points
 *
 *****/

void srThread( void );
void srThreadInit( void );
void srThreadUninit( void );
srBOOL srThreadProc( void );

#ifdef __cplusplus
}
#endif

#endif /* SR_H */
```

Appendix B: STRIDE Runtime Messages (srmsg.h)

```
/*
 *
 * FILE NAME: srmsg.h
 *
 * DESCRIPTION:
 *   STRIDE Runtime public message-based API
 *
 * -----
 * Copyright 2001 - 2008 by S2 Technologies, Inc.
 * -----
 */

#ifndef SRMSG_H
#define SRMSG_H

#ifdef __cplusplus
extern "C" {
#endif

#include "sr.h"

/*
 *
 * Special Message Type (MT) for STRIDE Runtime System Messaging
 *
 */

/* connection messages */
#define srCONNECT_OPEN_T_SMID      (1 | srMT_TWO | srST_CMD_VAL | srST_RSP_VAL |
srAC_SYS)
#define srCONNECT_CLOSE_T_SMID    (2 | srMT_TWO | srST_RSP_VAL |
srAC_SYS)
#define srCONNECT_STATUS_B_SMID   (3 | srMT_BRD | srST_RSP_VAL |
srAC_SYS)
#define srCONNECT_STATUS_T_SMID   (4 | srMT_TWO | srST_RSP_VAL |
srAC_SYS)

/* marshaling error message */
#define srERROR_MARSHAL_B_SMID    (5 | srMT_BRD | srST_RSP_VAL |
srAC_SYS)

/* remote subscriber message */
#define srSUBSCRIBERS_REMOTE_B_SMID (6 | srMT_BRD | srST_RSP_VAL |
srAC_SYS)

#define srMSG_SUID_LAST 6

/*
 *
 * Connecting
 *
 */

typedef enum
{
    srCONNECTION_CLOSED    = 0,
    srCONNECTION_OPEN      = 1,
    srCONNECTION_LISTEN    = 2
} srCONNECTION_e;

typedef enum
{
    srCONNECT_TIME_MICRO = 0,

```

```

    srCONNECT_TIME_MILLI = 1,
    srCONNECT_TIME_SEC   = 2
} srCONNECT_TIME_e;

typedef struct
{
    srCONNECTION_e eConnection;
} srConnectOpenRsp_t;

typedef struct
{
    srCONNECTION_e eConnection;
    srCONNECT_TIME_e eConnectTime;
    srWORD wConnectDuration;
    srWORD wConnectMaxTransportUnit;
    srWORD wConnectDatabaseVer;
} srConnectStatusRsp_t;

/*****
 *
 * TraceBuffer
 *
 *****/

typedef struct
{
    srWORD wSize;
    srBYTE yData[1];
} srTraceBufferRsp_t;

/*****
 *
 * TraceFiltering STID
 *
 *****/

typedef struct
{
    srWORD wSTID;
    srBOOL bSUIDEnable;
    srBOOL bSUIDPayload;
    srBOOL bSUIDGlobalDisable;
    srBOOL bTPIDEnable;
    srBOOL bTPIDPayload;
    srLevel_e eTPIDLevel;
} srTraceFilterSTIDCmd_t;

/*****
 *
 * TraceFiltering SUID
 *
 *****/

typedef struct
{
    srDWORD dwSUID;
    srBOOL bEnable;
    srBOOL bPayload;
    srBOOL bGlobalDisable;
} srTraceFilterSUIDCmd_t;

/*****
 *
 * Error Info
 *
 *****/

```



```
typedef struct
{
    srWORD   wCode;
    srBOOL   bCmd;
    srDWORD  dwSMID;
    srWORD   wRmtSRID;
    srBOOL   bMarshalOut;
} srErrMarshalRsp_t;

/*****
 *
 * Remote Subscriber broadcast
 *
 *****/

typedef struct
{
    srWORD   wSRID;
    srDWORD  dwSUID;
    srBOOL   bOn;
} srRemoteSubRsp_t;

#ifdef __cplusplus
}
#endif

#endif /* SRMSG_H */
```

Appendix C: STRIDE Runtime Configuration (srcfg.h)

```

/*****
 *
 * FILE NAME: srcfg.h
 *
 * DESCRIPTION:
 *   STRIDE Runtime configuration defines.
 *
 * -----
 * Copyright 2001 - 2008 by S2 Technologies, Inc.
 * -----
 *****/

#ifndef SRCFG_H
#define SRCFG_H

/*****
 *
 * Messaging
 *
 *****/

#define SRCFG_TOTAL_STIDS          16 /* number of STRIDE Transact IDs in system */
#define SRCFG_TOTAL_SUBCS         20 /* total number of subscribers at one time */

#define SRCFG_TOTAL_PTRS          30 /* total number of pointer entries */
#define SRCFG_SUID_TABLE_TYPE      1 /* 1 = Search based, 0 = Index based */
#define SRCFG_SUID_TABLE_SIZE     225 /* number of SUID Table entries */
#define SRCFG_TOTAL_SUIDS_QUEUED  50 /* total number of SUIDS queued at one time*/
#define SRCFG_STID_NAME_SIZE      15 /* max size of a STRIDE Transact ID name */

#define SRCFG_SUID_OVERRIDE        1 /* override (1 = enabled, 0 = disabled) */
#define SRCFG_SUID_OVERRIDE_STORAGE 0 /* override registration storage allocation*/

/* Set the hash function if SRCFG_SUID_TABLE_TYPE is search based- should return values*/
/* between 0 and (SRCFG_SUID_TABLE_SIZE-1) */
#define SRCFG_SUID_HASH_FUNC(x) (srWORD)((x)%(SRCFG_SUID_TABLE_SIZE))

/*****
 *
 * Tracing
 *
 *****/

#define SRCFG_TRACING_ENABLED      1 /* tracing enabled (1=enabled, 0=disabled)*/
#define SRCFG_TOTAL_TRACING_MEMORY 4096 /* number of bytes allocated for tracing */
#define SRCFG_TRACEBUFFER_MAX_SIZE 1000 /* max size of a single trace buffer */
#define SRCFG_TRACEBUFFER_WAKEUP_TIME 100 /* number of milliseconds between sending */

/*****
 *
 * Time Stamp
 *
 *****/

#define SRCFG_TIMESTAMP_UNITS      1 /* 0 = microseconds, 1 = Millisecs, 2 = secs */
#define SRCFG_TIMESTAMP_DURATION  1 /* number of TimeStamp Units per Tick */

/*****
 *
 * Auxiliary Data
 *
 *****/
#define SRCFG_AUXDATA              0 /* Use Auxiliary Data (1=Yes, 0=No) */

```

```

/*****
 *
 * Transport Settings
 *
 *****/
#define srCFG_MAX_TRANSPORT_UNIT      2048 /* 0=No Fragmentation, Number=Fragment Size*/
#define srCFG_DEFAULT_TRANSPORT_STATE 1    /* 1=Transport Ready, 0=Transport Not Ready*/

/*****
 *
 * Connection Settings
 *
 *****/

#define srCFG_CONNECTION_TIMEOUT      5000 /*connection timeout (1=enabled,0=disabled)*/

/*****
 *
 * RFC (Remote Function Call) Settings
 *
 *****/
#define srCFG_RFC_ENABLED              1    /* RFC enabled (1=Yes, 0=No) */
#define srCFG_RFC_PMM                  1    /* RFC pool memory management */
/* - (0=None, 1=Recovery, 2=Reallocation)*/
#define srCFG_RFC_PAL_PMM              0    /* PAL pool memory management (1=Yes, 0=No)*/

/*****
 *
 * Memory Management Settings
 *
 *****/
#define srCFG_MEMORY_MANAGEMENT        0    /*memory management (1=enabled, 0=disabled)*/

/* if memory management is enabled, set block size and max limits for dynamic and */
/* configurable memory */
#if srCFG_MEMORY_MANAGEMENT
#define srCFG_MEMORY_BLOCK_SIZE_SMALL  30    /* size of a small memory block */
#define srCFG_MEMORY_BLOCK_SIZE_MEDIUM 100   /* size of a medium memory block */
#define srCFG_MEMORY_BLOCK_SIZE_LARGE  500   /* size of a large memory block */
#define srCFG_MEMORY_BLOCK_SIZE_LARGE2 1000  /* size of a large2 memory block */
#define srCFG_MEMORY_BLOCK_SIZE_LARGE3 10000 /* size of a large3 memory block */
#define srCFG_MEMORY_BLOCK_SIZE_HUGE   0xFFFF /* size of a huge memory block */

#define srCFG_MEMORY_BLOCK_MAX_SMALL   5000  /* max number of small memory blocks */
#define srCFG_MEMORY_BLOCK_MAX_MEDIUM 250    /* max number of medium memory blocks */
#define srCFG_MEMORY_BLOCK_MAX_LARGE   250   /* max number of large memory blocks */
#define srCFG_MEMORY_BLOCK_MAX_LARGE2  100   /* max number of large2 memory blocks */
#define srCFG_MEMORY_BLOCK_MAX_LARGE3  50    /* max number of large3 memory blocks */
#define srCFG_MEMORY_BLOCK_MAX_HUGE    50    /* max number of huge memory blocks */
#endif /* srCFG_MEMORY_MANAGEMENT */

/*****
 *
 * Multi-Process Settings
 *
 *****/
#define srCFG_MULTI_PROC_TARGET        0    /*multi-process target (1=enabled,0=disabled)*/

#if srCFG_MULTI_PROC_TARGET && !srCFG_MEMORY_MANAGEMENT
#error "Multi-process target requires memory management."
#endif

/*****
 *

```

```
* Variable Arguments
*
*****/
#define srCFG_VAR_ARGS_ENABLED          1    /*variable arguments (1=enabled,0=disabled)*/

/*****
*
* Debug Settings
*
*****/
#define srCFG_ERROR_CHECK_LEVEL        2    /* levels(0,1,2), none(0) */

#endif /* SRCFG_H */
```

Appendix D: STRIDE Runtime Test Services (srtest.h)

```
/*
 *
 * FILE NAME: srtest.h
 *
 * DESCRIPTION:
 * STRIDE C based Runtime Test Services (RTS) APIs and Messages &
 * C++ Base Class Wrappers for Runtime Test Services (RTS).
 *
 * API Documentation follows doxygen style. Tool available at
 * www.doxygen.org can be used to generate an html output for documentation.
 *
 * -----
 * Copyright 2007 - 2008 by S2 Technologies, Inc.
 * -----
 */

#ifndef SRTEST_H
#define SRTEST_H

#ifdef __cplusplus
extern "C" {
#endif

#include "sr.h"

/*
 *
 * Configurables:
 *
 */

/**
 * Constant for maximum string length. The value is configurable.
 */
#define srTEST_MAX_STRING_LENGTH 1024

/**
 * Constant for starting SUID (Message ID) for internal messages. The value is
configurable.
 */
#define srTEST_BASE_SUID 0xFFFFF0

/*
 *
 * Type Definitions and Constants:
 *
 */

/**
 * Return error code to indicate an error on an invalid handle.
 */
#define srTEST_ERR_HANDLE_INVALID 101

/**
 * Return error code to indicate a warning on a truncated string.
 */
#define srTEST_WARN_STR_TRUNCATED 102

/**
 * Return code to indicate failure to create a new test suite.
 */
#define srTEST_SUITE_INVALID 0xFFFFFFFF

/**
 * Return code to indicate failure to create a new test case.
 */
#define srTEST_CASE_INVALID 0xFFFFFFFF

```

```

/**
 * Return code to indicate failure to create a new test case.
 */
#define srTEST_ANNOTATION_INVALID  0xFFFFFFFF

/**
 * Constant for default test suite. When used this constant in calling test APIs, data
 applies to the default test suite.
 */
#define srTEST_SUITE_DEFAULT  0

/**
 * Constant for default test case. When used this constant in calling test APIs, data
 applies to the current default test suite.
 */
#define srTEST_CASE_DEFAULT  0

/**
 * Test suite handle.
 */
typedef srDWORD srTestSuiteHandle_t;

/**
 * Test case handle.
 */
typedef srDWORD srTestCaseHandle_t;

/**
 * Test annotation handle.
 */
typedef srDWORD srTestAnnotationHandle_t;

/**
 * Enum for test results.
 */
typedef enum
{
    srTEST_PASS,
    srTEST_FAIL,
    srTEST_NOTINUSE,
    srTEST_INPROGRESS,
    srTEST_TIMINGONLY
} srTestStatus_e;

/**
 * Struct for test case totals.
 */
typedef struct
{
    srDWORD pass;
    srDWORD fail;
    srDWORD not_in_use;
    srDWORD in_progress;
} srTestCaseTotals_t;

/**
 * Enum for annotation levels.
 */
typedef enum
{
    srTEST_ANNOTATION_LEVEL_TRACE,
    srTEST_ANNOTATION_LEVEL_DEBUG,
    srTEST_ANNOTATION_LEVEL_INFO,
    srTEST_ANNOTATION_LEVEL_WARN,
    srTEST_ANNOTATION_LEVEL_ERROR,
    srTEST_ANNOTATION_LEVEL_FATAL
} srTestAnnotationLevel_e;

/*****

```

```

*
* Test Service APIs:
*
*****/

/**
 * Public API to add a new test suite to the specified test suite.
 * @param tParent Handle to the parent test suite to which new test suite is to be
added. srTEST_SUITE_DEFAULT can be used for the default test suite.
 * @param szName Pointer to a null-terminated string that represents the name of test
suite. If null, the default host naming scheme will be used.
 * @return Handle of the newly created test suite. srTEST_SUITE_INVALID indicates
failure to create test suite.
 */
srEXPORT srTestSuiteHandle_t srSTDCALL srTestSuiteAddSuite(srTestSuiteHandle_t tParent,
const srCHAR * szName);

/**
 * Public API to set the name of the specified test suite.
 * @param tTestSuite Handle to a test suite. srTEST_SUITE_DEFAULT can be used for the
default test suite.
 * @param szName Pointer to a null-terminated string representing the name of test
suite. Cannot be null.
 * @return srOK on success, srERR on null string, srTEST_ERR_HANDLE_INVALID on invalid
handle, srTEST_WARN_STR_TRUNCATED on truncated string.
 */
srEXPORT srWORD srSTDCALL srTestSuiteSetName(srTestSuiteHandle_t tTestSuite, const srCHAR
* szName);

/**
 * Public API to set the description of the specified test suite.
 * @param tTestSuite Handle to a test suite. srTEST_SUITE_DEFAULT can be used for the
default test suite.
 * @param szDescr Pointer to a null-terminated string representing the description of
test suite. Cannot be null.
 * @return srOK on success, srERR on null string, srTEST_ERR_HANDLE_INVALID on invalid
handle, srTEST_WARN_STR_TRUNCATED on truncated string.
 */
srEXPORT srWORD srSTDCALL srTestSuiteSetDescription(srTestSuiteHandle_t tTestSuite, const
srCHAR * szDescr);

/**
 * Public API to add a new test case to the specified test suite.
 * @param tParent Handle to the parent test suite to which new test case is to be added.
srTEST_SUITE_DEFAULT can be used for the default test suite.
 * @param szName Pointer to a null-terminated string that represents the name of test
case. If null, the default host naming scheme will be used.
 * @return Handle of the newly created test case. srTEST_CASE_INVALID indicates failure
to create test case.
 */
srEXPORT srTestSuiteHandle_t srSTDCALL srTestSuiteAddTest(srTestSuiteHandle_t tParent,
const srCHAR * szName);

/**
 * Public API to set the name of the specified test case.
 * @param tTestCase Handle to a test case. srTEST_CASE_DEFAULT can be used for the
default test case.
 * @param szName Pointer to a null-terminated string representing the name of test case.
Cannot be null.
 * @return srOK on success, srERR on null string, srTEST_ERR_HANDLE_INVALID on invalid
handle, srTEST_WARN_STR_TRUNCATED on truncated string.
 */
srEXPORT srWORD srSTDCALL srTestCaseSetName(srTestCaseHandle_t tTestCase, const srCHAR *
szName);

/**
 * Public API to set the description of the specified test case.
 * @param tTestCase Handle to a test case. srTEST_CASE_DEFAULT can be used for the
default test case.
 * @param szDescr Pointer to a null-terminated string representing the description of
test case. Cannot be null.

```

```

    * @return srOK on success, srERR on null string, srTEST_ERR_HANDLE_INVALID on invalid
    handle, srTEST_WARN_STR_TRUNCATED on truncated string.
    */
srEXPORT srWORD srSTDCALL srTestCaseSetDescription(srTestCaseHandle_t tTestCase, const
srCHAR * szDescr);

/**
 * Public API to add a comment (aka a log) to be reported with the specified test case.
 * @param tTestCase Handle to a test case. srTEST_CASE_DEFAULT can be used for the
    default test case.
 * @param szLabel Pointer to a null-terminated string for the label. If null, the
    default label will be applied.
 * @param szFmtComment Pointer to a null-terminated string representing the comment.
    Cannot be null.
 * @param ... (optional) Variable argument list to format szFmtComment.
 * @return srOK on success, srERR on null string or invalid formatted string,
    srTEST_ERR_HANDLE_INVALID on invalid handle, srTEST_WARN_STR_TRUNCATED on truncated
    string.
    */
srEXPORT srWORD srSTDCALL srTestCaseAddComment(srTestCaseHandle_t tTestCase, const srCHAR
* szLabel, const srCHAR * szFmtComment, ...);

/**
 * Public API to set the result of test case.
 * @param tTestCase Handle to a test case. srTEST_CASE_DEFAULT can be used for the
    default test case.
 * @param eStatus Result of the test.
 * @param dwDuration The duration of the test in clock ticks. Using "0" allows the
    STRIDE Runtime (Intercept Module) to set the time automatically.
 * @return srOK on success, srTEST_ERR_HANDLE_INVALID on invalid handle.
    */
srEXPORT srWORD srSTDCALL srTestCaseSetStatus(srTestCaseHandle_t tTestCase,
srTestStatus_e eStatus, srDWORD dwDuration);

/**
 * Public API to set the result of test case with extended failure information.
 * @param tTestCase Handle to a test case. srTEST_CASE_DEFAULT can be used for the
    default test case.
 * @param eStatus Result of the test.
 * @param dwDuration The duration of the test in clock ticks. Using "0" allows the
    STRIDE Runtime (Intercept Module) to set the time automatically.
 * @param lExtendedFailureCode In the case of failure the actual return value of a
    numeric test method can be provided.
 * @return srOK on success, srTEST_ERR_HANDLE_INVALID on invalid handle.
    */
srEXPORT srWORD srSTDCALL srTestCaseSetStatusEx(srTestCaseHandle_t tTestCase,
srTestStatus_e eStatus, srDWORD dwDuration,
srLONG lExtendedFailureCode);

/**
 * Public API to add a new annotation to the specified test suite.
 * @param tParent Handle to the parent test suite to which new test annotation is to be
    added. srTEST_SUITE_DEFAULT can be used for the default test suite.
 * @param eLevel Annotation level. Cannot be null.
 * @param szName Pointer to a null-terminated string representing the name of
    annotation. If null, the default host naming scheme will be used.
 * @param szDescr Pointer to a null-terminated string representing the description of
    annotation. If null, description will be empty.
 * @return Handle of the newly created test annotation. srTEST_ANNOTATION_INVALID
    indicates failure to create test annotation.
    */
srEXPORT srTestAnnotationHandle_t srSTDCALL srTestSuiteAddAnnotation(srTestSuiteHandle_t
tParent,

srTestAnnotationLevel_e eLevel,
szName,
szDescr);

```



```

* Public API to add a comment (aka a log) to the specified test annotation.
* @param tTestAnnotation Handle to a test annotation.
* @param szLabel Pointer to a null-terminated string for the label. If null, the
default label will be applied.
* @param szFmtComment Pointer to a null-terminated string representing the comment.
Cannot be null.
* @param ... (optional) Variable argument list to format szFmtComment.
* @return srOK on success, srERR on null string or invalid formatted string,
srTEST_ERR_HANDLE_INVALID on invalid handle, srTEST_WARN_STR_TRUNCATED on truncated
string.
*/
srEXPORT srWORD srSTDCALL srTestAnnotationAddComment(srTestAnnotationHandle_t
tTestAnnotation, const srCHAR * szLabel, const srCHAR * szFmtComment, ...);

/**
* Public macro to add a comment with location to the test report.
*/
#define srTEST_ADD_COMMENT_WITH_LOCATION(id,label,str) \
    srTestCaseAddComment(id, label, "%s : %d in %s", str, __LINE__, __FILE__)

/*****
*
* Test Service Messages:
*
*****/

/* SUIDs */

#define srTEST_SUITE_ADD_SUITE_SUID (srTEST_BASE_SUID + 0)
#define srTEST_SUITE_SET_NAME_SUID (srTEST_BASE_SUID + 1)
#define srTEST_SUITE_SET_DESCRIPTION_SUID (srTEST_BASE_SUID + 2)
#define srTEST_SUITE_ADD_TEST_SUID (srTEST_BASE_SUID + 3)
#define srTEST_CASE_SET_NAME_SUID (srTEST_BASE_SUID + 4)
#define srTEST_CASE_SET_DESCRIPTION_SUID (srTEST_BASE_SUID + 5)
#define srTEST_CASE_ADD_COMMENT_SUID (srTEST_BASE_SUID + 6)
#define srTEST_CASE_SET_STATUS_SUID (srTEST_BASE_SUID + 7)
#define srTEST_METHOD_START_SUID (srTEST_BASE_SUID + 8)
#define srTEST_SUITE_ADD_ANNOTATION_SUID (srTEST_BASE_SUID + 9)
#define srTEST_ANNOTATION_ADD_COMMENT_SUID (srTEST_BASE_SUID + 10)

/* Message IDs (SMIDs) */

#define srTEST_SUITE_ADD_SUITE_MSG (srTEST_SUITE_ADD_SUITE_SUID | srMT_BRD
| srST_RSP_VAL)
#define srTEST_SUITE_SET_NAME_MSG (srTEST_SUITE_SET_NAME_SUID | srMT_BRD
| srST_RSP_VAL)
#define srTEST_SUITE_SET_DESCRIPTION_MSG (srTEST_SUITE_SET_DESCRIPTION_SUID | srMT_BRD
| srST_RSP_VAL)
#define srTEST_SUITE_ADD_TEST_MSG (srTEST_SUITE_ADD_TEST_SUID | srMT_BRD
| srST_RSP_VAL)
#define srTEST_CASE_SET_NAME_MSG (srTEST_CASE_SET_NAME_SUID | srMT_BRD
| srST_RSP_VAL)
#define srTEST_CASE_SET_DESCRIPTION_MSG (srTEST_CASE_SET_DESCRIPTION_SUID | srMT_BRD
| srST_RSP_VAL)
#define srTEST_CASE_ADD_COMMENT_MSG (srTEST_CASE_ADD_COMMENT_SUID | srMT_BRD
| srST_RSP_VAL)
#define srTEST_CASE_SET_STATUS_MSG (srTEST_CASE_SET_STATUS_SUID | srMT_BRD
| srST_RSP_VAL)
#define srTEST_METHOD_START_MSG (srTEST_METHOD_START_SUID | srMT_BRD
| srST_RSP_VAL)
#define srTEST_SUITE_ADD_ANNOTATION_MSG (srTEST_SUITE_ADD_ANNOTATION_SUID | srMT_BRD
| srST_RSP_VAL)
#define srTEST_ANNOTATION_ADD_COMMENT_MSG (srTEST_ANNOTATION_ADD_COMMENT_SUID | srMT_BRD
| srST_RSP_VAL)

/* Message Payload Structures */

typedef struct {
    srDWORD parentTestSuiteID;
    srDWORD subTestSuiteID;
    srCHAR * name;

```

```

} srTestSuiteAddSuite_t;

typedef struct {
    srDWORD testSuiteID;
    srCHAR * name;
} srTestSuiteSetName_t;

typedef struct {
    srDWORD testSuiteID;
    srCHAR * description;
} srTestSuiteSetDescription_t;

typedef struct {
    srDWORD parentTestSuiteID;
    srDWORD testCaseID;
    srCHAR * name;
} srTestSuiteAddTest_t;

typedef struct {
    srDWORD testCaseID;
    srCHAR * name;
} srTestCaseSetName_t;

typedef struct {
    srDWORD testCaseID;
    srCHAR * description;
} srTestCaseSetDescription_t;

typedef struct {
    srDWORD testCaseID;
    srCHAR * label;
    srCHAR * comment;
} srTestCaseAddComment_t;

typedef struct {
    srDWORD testCaseID;
    srTestStatus_e status;
    srDWORD duration;
    srBOOL fromIM;
    /* the extendedFailureCode is used to optionally propogate additional,
    optional, failure information. By convention it has meaning only
    when status == srTEST_FAIL. When status != srTEST_FAIL, it should
    be 0. When status == srTEST_PASS then extendedFailureCode may be
    0 or non-zero. If 0 then there is no extended failure information.
    if non-zero, then we assume the test method in question returns
    an integral type and extendedFailureCode records the value returned.
    */
    srLONG extendedFailureCode;
} srTestCaseSetStatus_t;

typedef struct {
    srDWORD testUnitID;
    srCHAR * methodName;
} srTestMethodStart_t;

typedef struct {
    srDWORD parentTestSuiteID;
    srDWORD testAnnotationID;
    srCHAR * level;
    srCHAR * name;
    srCHAR * description;
} srTestSuiteAddAnnotation_t;

typedef struct {
    srDWORD testAnnotationID;
    srCHAR * label;
    srCHAR * comment;
} srTestAnnotationAddComment_t;

/*

```

```

*****
*
* SCL Pragmas
*
*****
*/

#ifdef _SCL

#pragma scl_msg(srTEST_SUITE_ADD_SUITE_MSG, srTestSuiteAddSuite_t)
#pragma scl_ptr(srTestSuiteAddSuite_t.name, RETURN, PRIVATE)
#pragma scl_string(srTestSuiteAddSuite_t.name, srTEST_MAX_STRING_LENGTH)

#pragma scl_msg(srTEST_SUITE_SET_NAME_MSG, srTestSuiteSetName_t)
#pragma scl_ptr(srTestSuiteSetName_t.name, RETURN, PRIVATE)
#pragma scl_string(srTestSuiteSetName_t.name, srTEST_MAX_STRING_LENGTH)

#pragma scl_msg(srTEST_SUITE_SET_DESCRIPTION_MSG, srTestSuiteSetDescription_t)
#pragma scl_ptr(srTestSuiteSetDescription_t.description, RETURN, PRIVATE)
#pragma scl_string(srTestSuiteSetDescription_t.description, srTEST_MAX_STRING_LENGTH)

#pragma scl_msg(srTEST_SUITE_ADD_TEST_MSG, srTestSuiteAddTest_t)
#pragma scl_ptr(srTestSuiteAddTest_t.name, RETURN, PRIVATE)
#pragma scl_string(srTestSuiteAddTest_t.name, srTEST_MAX_STRING_LENGTH)

#pragma scl_msg(srTEST_CASE_SET_NAME_MSG, srTestCaseSetName_t)
#pragma scl_ptr(srTestCaseSetName_t.name, RETURN, PRIVATE)
#pragma scl_string(srTestCaseSetName_t.name, srTEST_MAX_STRING_LENGTH)

#pragma scl_msg(srTEST_CASE_SET_DESCRIPTION_MSG, srTestCaseSetDescription_t)
#pragma scl_ptr(srTestCaseSetDescription_t.description, RETURN, PRIVATE)
#pragma scl_string(srTestCaseSetDescription_t.description, srTEST_MAX_STRING_LENGTH)

#pragma scl_msg(srTEST_CASE_ADD_COMMENT_MSG, srTestCaseAddComment_t)
#pragma scl_ptr(srTestCaseAddComment_t.label, RETURN, PRIVATE)
#pragma scl_string(srTestCaseAddComment_t.label, srTEST_MAX_STRING_LENGTH)
#pragma scl_ptr(srTestCaseAddComment_t.comment, RETURN, PRIVATE)
#pragma scl_string(srTestCaseAddComment_t.comment, srTEST_MAX_STRING_LENGTH)

#pragma scl_msg(srTEST_CASE_SET_STATUS_MSG, srTestCaseSetStatus_t)

#pragma scl_msg(srTEST_METHOD_START_MSG, srTestMethodStart_t)
#pragma scl_ptr(srTestMethodStart_t.methodName, RETURN, PRIVATE)
#pragma scl_string(srTestMethodStart_t.methodName, srTEST_MAX_STRING_LENGTH)

#pragma scl_msg(srTEST_SUITE_ADD_ANNOTATION_MSG, srTestSuiteAddAnnotation_t)
#pragma scl_ptr(srTestSuiteAddAnnotation_t.level, RETURN, PRIVATE)
#pragma scl_string(srTestSuiteAddAnnotation_t.level, srTEST_MAX_STRING_LENGTH)
#pragma scl_ptr(srTestSuiteAddAnnotation_t.name, RETURN, PRIVATE)
#pragma scl_string(srTestSuiteAddAnnotation_t.name, srTEST_MAX_STRING_LENGTH)
#pragma scl_ptr(srTestSuiteAddAnnotation_t.description, RETURN, PRIVATE)
#pragma scl_string(srTestSuiteAddAnnotation_t.description, srTEST_MAX_STRING_LENGTH)

#pragma scl_msg(srTEST_ANNOTATION_ADD_COMMENT_MSG, srTestAnnotationAddComment_t)
#pragma scl_ptr(srTestAnnotationAddComment_t.label, RETURN, PRIVATE)
#pragma scl_string(srTestAnnotationAddComment_t.label, srTEST_MAX_STRING_LENGTH)
#pragma scl_ptr(srTestAnnotationAddComment_t.comment, RETURN, PRIVATE)
#pragma scl_string(srTestAnnotationAddComment_t.comment, srTEST_MAX_STRING_LENGTH)

#endif _SCL_TESTING

#pragma scl_function(srTestSuiteAddSuite)
#pragma scl_string(srTestSuiteAddSuite.szName, srTEST_MAX_STRING_LENGTH)

#pragma scl_function(srTestSuiteSetName)
#pragma scl_string(srTestSuiteSetName.szName, srTEST_MAX_STRING_LENGTH)
#pragma scl_values(srTestSuiteSetName(), srOK, srERR, srTEST_ERR_HANDLE_INVALID,
srTEST_WARN_STR_TRUNCATED)

#pragma scl_function(srTestSuiteSetDescription)

```

```

#pragma scl_string(srTestSuiteSetDescription.szDescr, srTEST_MAX_STRING_LENGTH)
#pragma scl_values(srTestSuiteSetDescription(), srOK, srERR, srTEST_ERR_HANDLE_INVALID,
srTEST_WARN_STR_TRUNCATED)

#pragma scl_function(srTestSuiteAddTest)
#pragma scl_string(srTestSuiteAddTest.szName, srTEST_MAX_STRING_LENGTH)

#pragma scl_function(srTestCaseSetName)
#pragma scl_string(srTestCaseSetName.szName, srTEST_MAX_STRING_LENGTH)
#pragma scl_values(srTestCaseSetName(), srOK, srERR, srTEST_ERR_HANDLE_INVALID,
srTEST_WARN_STR_TRUNCATED)

#pragma scl_function(srTestCaseSetDescription)
#pragma scl_string(srTestCaseSetDescription.szDescr, srTEST_MAX_STRING_LENGTH)
#pragma scl_values(srTestCaseSetDescription(), srOK, srERR, srTEST_ERR_HANDLE_INVALID,
srTEST_WARN_STR_TRUNCATED)

#pragma scl_function(srTestCaseAddComment)
#pragma scl_string(srTestCaseAddComment.szLabel, srTEST_MAX_STRING_LENGTH)
#pragma scl_string(srTestCaseAddComment.szFmtComment, srTEST_MAX_STRING_LENGTH)
#pragma scl_values(srTestCaseAddComment(), srOK, srERR, srTEST_ERR_HANDLE_INVALID,
srTEST_WARN_STR_TRUNCATED)

#pragma scl_function(srTestCaseSetStatus)
#pragma scl_values(srTestCaseSetStatus(), srOK, srTEST_ERR_HANDLE_INVALID)

#pragma scl_function(srTestCaseSetStatusEx)
#pragma scl_values(srTestCaseSetStatusEx(), srOK, srTEST_ERR_HANDLE_INVALID)

#pragma scl_function(srTestSuiteAddAnnotation)
#pragma scl_string(srTestSuiteAddAnnotation.szName, srTEST_MAX_STRING_LENGTH)
#pragma scl_string(srTestSuiteAddAnnotation.szDescr, srTEST_MAX_STRING_LENGTH)

#pragma scl_function(srTestAnnotationAddComment)
#pragma scl_string(srTestAnnotationAddComment.szLabel, srTEST_MAX_STRING_LENGTH)
#pragma scl_string(srTestAnnotationAddComment.szFmtComment, srTEST_MAX_STRING_LENGTH)
#pragma scl_values(srTestAnnotationAddComment(), srOK, srERR, srTEST_ERR_HANDLE_INVALID,
srTEST_WARN_STR_TRUNCATED)

#endif /* _SCL_TESTING */

#endif /* _SCL */

#ifdef __cplusplus
}
#endif

#ifdef __cplusplus

/*****
 *
 * C++ Base Class Wrappers for Runtime Test Services (RTS)
 *
 *****/

namespace stride
{
/**
 * Base class wrapper for Test Case
 */
class srEXPORT srTestCase
{
    friend class srTestSuite;
    friend class srTest;

private:
/**
 * Private Constructor.

```

```

    * @param szName (optional) Pointer to null-terminated string that represents the
    name of test case. If empty, the default host naming scheme will be used.
    * @param tHandle (optional) Handle to test case. If empty, default test case will
    be used.
    */
    srTestCase(const srCHAR * szName = srNULL,
               srTestCaseHandle_t tHandle = srTEST_CASE_DEFAULT);

public:
    /**
     * Destructor.
     */
    ~srTestCase();

public:
    /**
     * Public method to set the name of test case.
     * @param szName Pointer to null-terminated string representing the name of test
     case. Cannot be empty.
     */
    srWORD SetName(const srCHAR * szName);

    /**
     * Public method to set the description of test case.
     * @param szDescr Pointer to null-terminated string representing the description of
     test case. Cannot be empty.
     */
    srWORD SetDescription(const srCHAR * szDescr);

    /**
     * Public method to add a comment to test case.
     * @param szLabel Pointer to null-terminated string for the label. If null, the
     default label will be applied.
     * @param szFmtComment Pointer to null-terminated string representing the comment.
     Cannot be empty.
     * @param ... (optional) Variable argument list to format szFmtComment.
     */
    srWORD AddComment(const srCHAR * szLabel, const srCHAR * szFmtComment, ...);

    /**
     * Public method to set the result of test case.
     * @param eStatus Result of the test.
     * @param dwDuration (optional) The duration of the test in clock ticks. The default
     is 0.
     */
    srWORD SetStatus(srTestStatus_e eStatus, srDWORD dwDuration = 0);

private:
    /**
     * Member variable for test case handle.
     */
    srTestCaseHandle_t m_tHandle;
};

/**
 * Base class wrapper for Test Annotation
 */
class srEXPORT srTestAnnotation
{
    friend class srTestSuite;
    friend class srTest;

private:
    /**
     * Private Constructor.
     * @param tHandle Handle to test annotation.
     */
    srTestAnnotation(srTestAnnotationHandle_t tHandle);

public:
    /**

```

```

    * Destructor.
    */
    ~srTestAnnotation();

public:
    /**
     * Public method to add a comment to test annotation.
     * @param szLabel Pointer to null-terminated string for the label. If null, the
     default label will be applied.
     * @param szFmtComment Pointer to null-terminated string representing the comment.
     Cannot be empty.
     * @param ... (optional) Variable argument list to format szFmtComment.
     */
    srWORD AddComment(const srCHAR * szLabel, const srCHAR * szFmtComment, ...);

private:
    /**
     * Member variable for test annotation handle.
     */
    srTestAnnotationHandle_t m_tHandle;
};

/**
 * Base class wrapper for Test Suite
 */
class srEXPORT srTestSuite
{
    friend class srTest;

private:
    /**
     * Private Constructor.
     * @param szName (optional) Pointer to null-terminated string that represents the
     name of test suite. If empty, the default host naming scheme will be used.
     * @param tHandle (optional) Handle to test suite. If empty, default test suite will
     be used.
     */
    srTestSuite(const srCHAR * szName = srNULL,
                srTestSuiteHandle_t tHandle = srTEST_SUITE_DEFAULT);

public:
    /**
     * Destructor.
     */
    ~srTestSuite();

    /* Default copy constructor is sufficient */

public:
    /**
     * Public method to add a new test suite.
     * @param szName (optional) Pointer to null-terminated string that represents the
     name of test suite. If empty, the default host naming scheme will be used.
     */
    srTestSuite AddSuite(const srCHAR * szName = srNULL);

    /**
     * Public method to set the name of test suite.
     * @param szName Pointer to null-terminated string representing the name of test
     suite. Cannot be empty.
     */
    srWORD SetName(const srCHAR * szName);

    /**
     * Public method to set the description of test suite.
     * @param szDescr Pointer to null-terminated string representing the description of
     test suite. Cannot be empty.
     */
    srWORD SetDescription(const srCHAR * szDescr);

    /**

```

```

    * Public method to add a new test case to test suite.
    * @param szName (optional) Pointer to null-terminated string that represents the
name of test case. If empty, the default host naming scheme will be used.
    */
    srTestCase AddTest(const srCHAR * szName = srNULL);

    /**
    * Public method to add a new test annotation to test suite.
    * @param eLevel Annotation level.
    * @param szName (optional) Pointer to null-terminated string that represents the
name of test annotation. If empty, the default host naming scheme will be used.
    * @param szDescr (optional) Pointer to null-terminated string representing the
description of annotation. If empty, description will be blank.
    */
    srTestAnnotation AddAnnotation(srTestAnnotationLevel_e eLevel,
                                   const srCHAR * szName = srNULL,
                                   const srCHAR * szDescr = srNULL);

private:
    /**
    * Member variable for test suite handle.
    */
    srTestSuiteHandle_t m_tHandle;
};

/**
 * Base class wrapper for all Test objects
 */
class srEXPORT srTest
{
protected:
    /**
    * Protected Constructor.
    * @param szName (optional) Pointer to null-terminated string that represents names
of test suite and case. If empty, the default host naming scheme will be used.
    */
    srTest(const srCHAR * szName = srNULL);

    /**
    * Virtual Destructor.
    */
    virtual ~srTest();

protected:
    /**
    * Protected member variable for the default test suite.
    */
    srTestSuite testSuite;

    /**
    * Protected member variable for the default test case.
    */
    srTestCase testCase;
};

} /* namespace stride */

#endif /* __cplusplus */

#endif /* SRTEST_H */

```