Runtime Developer's Guide

Version 4.01

Published by

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

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

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) 17

1.1.3. STRIDE Transactor ID (STID) 17

1.1.4. STRIDE Trace Point ID (STPID) 17

1.1.5. STRIDE Response ID (SRID) 18

1.1.6. Notification of Traffic ID (NID) 18

1.2. Memory Requirements 19

1.2.1. Messaging Memory 20

1.2.2. Tracing Memory 22

1.2.3. Transport Settings 23

1.2.4. Memory Management 23

1.2.5. Multi-Process Target 24

1.3. Using the API 25

1.3.1. Creating an STID 25

1.3.2. Creating a STRIDE Message 25
1.3.3. Registering Messages 27
1.3.4. Overriding Registration 28
1.3.5. Subscribing to Messages 28
1.3.6. Reading and Sending Messages 28
1.3.7. Using Pointers 30
1.3.8. Returning Message Memory 31
1.3.9. Pointer Memory Policies 32
1.3.10. Trace Points 33
1.3.11. Data Format Conformance 33
1.4. Routing with Access Class Registration 33
1.4.1. Remote Messaging (RM) Overview 34
1.4.2. Access Class Intercept Module 34
1.5. Connecting to the Host 35
1.5.1. Connection Settings 35
2. Remote Messaging (RM) 36
2.1. Implementing a Remote Messaging Service 37
2.1.1. How It Works 37
2.1.2. Issues to Consider 38
2.1.3. Translating between STRIDE and Native Message IDs 39
2.2. Using a Remote Message Stub (RMS) 39
2.2.1. Remote Message Stub Thread Setup 40
2.2.2. Remote Message Stub Thread Messages 40
2.2.3. How to Handle Responses 41
2.2.4. Binding Native Commands with Native Responses 42
2.2.5. Wait Event 42
2.2.6. Subscriptions and Broadcasts 42
2.2.7. Pointers 43

2.2.8. Message Tracing 43
2.2.9. Remote Message Stub Thread Example 43
2.3. Using a Remote Message Proxy (RMP) 45
2.3.1. Remote Message Proxy Routing 46
2.3.2. Binding Native Commands with Native Responses 46
2.3.3. Receiving STRIDE Responses – Sending Native Responses 47
3. Runtime API Services 48
3.1. Setup and Shutdown 49
srInit() 50
srUninit() 51
srCreateSTID() 52
srDeleteSTID() 54
3.2. Messaging 55
srRegister() 56
srRegisterAccessClass() 58
srRead() 60
srReadComplete() 63
srSendCmd() 65
srSendRsp() 67
srBroadcast() 70
srSubscribe() 73
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 112
3.8. I-block 113
srIBlockOutReady() 113
3.9. Runtime Thread Entry and Exit Points 114
srThread() 115
srThreadInit() 116
srThreadUninit() 117
srThreadProc() 118
3.10. Host Override Routines 119
srHostShutdownIM() 120
3.11. Connecting 122

srCONNECT_OPEN_T_SMID 123
srCONNECT_CLOSE_T_SMID 124
srCONNECT_STATUS_B_SMID 125
srCONNECT_STATUS_T_SMID 127
3.12. Database Loading Routines 128
srHOST_LOAD_DB_O_SMID 129
srHOST_LOAD_DB_STATUS_B_SMID 130
srHOST_LOAD_DB_STATUS_T_SMID 133
3.13. Trace Buffers 135
srTRACE_BUFFER_B_SMID 135
3.13.1. Trace Filtering 137
3.14. Subscriber Information 139
srSUBSCRIBERS_LOCAL_B_SMID 140
srSUBSCRIBERS_REMOTE_B_SMID 141
3.15. Marshaling Errors 142
srERROR_MARSHAL_B_SMID 143
4. STRIDE Runtime Internals 144
4.1. STRIDE Runtime Thread and Procedure 144
4.2. STRIDE Runtime Modules 144
4.3. STRIDE Runtime Files 146
Appendix A: STRIDE Runtime API (sr.h) 148
Appendix B: STRIDE Runtime Configuration (srcfg.h) 157

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 activites, 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*.)

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

Figure 1: Standard Data Types

Standard Defines

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

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

Figure 2: Standard Defines

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.

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:

Public API Naming Conventions

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.

Private API Naming Conventions

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

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 releated functionality (example: "err" is used for the component tag for the files and APIs releated 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 accessable 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 Identifer. 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 Abstaction 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.

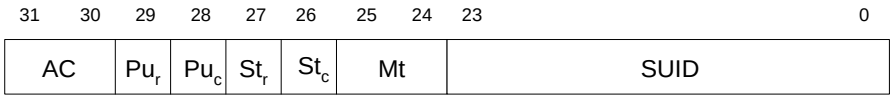
The PAL defines the set of OS functionality required by the platform to support the STRIDE Runtime. The *pal.h* header file defines the PAL functionality. The PAL also defines functionality required by the STRIDE Runtime to transmit and receive packets of data (called I-blocks) using the platform's transport 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 SMID is a unique message ID and a set of attributes associated with the message. The SCM defines the SUID portion as the low order 24 bits of the SMID (bits 0 thru 23), allowing unique identifiers between zero and 2^{24}-1. The attributes are stored in bits 24 thru 29. Bits 30 and 31 are reserved and must be set to 0 for all user-defined SMIDs. Each SCL-compliant message must be assigned a unique message ID. The format of the STRIDE message ID is illustrated below:



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:

Message Type (Mt) Values

Meaning	Value
One way command	0
One way response	1
Two way command/response	2
Broadcastg	3

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 uses the ST attribute when determining whether to route locally or remotely across platform boundaries. The following tables describe the ST attribute settings:

SendType for Command (STc) Values

Meaning	Value
By Pointer (combined with NULL data value means no payload)	0
By Value	1

SendType for Response (STr) Values

Meaning	Value
By Pointer (combined with NULL data value means no payload)	0
By Value	1

When a payload is passed by pointer, a Pointer Usage (PU) attribute is required. Otherwise the value of the PU is ignored. 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 a common pool. 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) Values

Meaning	Value
Pool	0
Private	1

Pointer Usage for Response (PUr) Values

Meaning	Value
Pool	0
Private	1

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

Access Class (AC) Values

Meaning	Value
Message	0
Function	1
System/Application	2
Reserved	3

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.

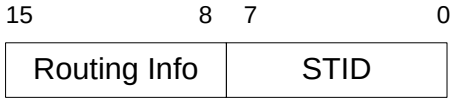
Generally, each thread will have its own STID.

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	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 meory for stack usage. This memory used by the STRIDE Runtime is requested through the *palMemSegmentOpen()* and *palMemSegmentClose()* routines. **Note:** This does not include space required for interrupt services routines that use the same stack as the interrupted thread. In case of multi-process target is enabled, this memory should be shared among applications.
Dynamic	All memory requirements for payload storage are requested through the *palMemAlloc(..)* and *palMemFree(..)* 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. If the STRIDE Runtime Memory Management is enabled, dynamic memory will be allocated from the *configurable memory*.
Configurable	The configurable memory used by the STRIDE Runtime is requested through the *palMemSegmentOpen()* and *palMemSegmentClose()* routines. The blocks of memory that are configurable are defined in the *srcfg.h* file. The constants contained in the file should be customized for each project based on its specific requirements. **Note:** In case of multi-process target is enabled, this memory should be shared among applications.

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 1: 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 dowloaded 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.

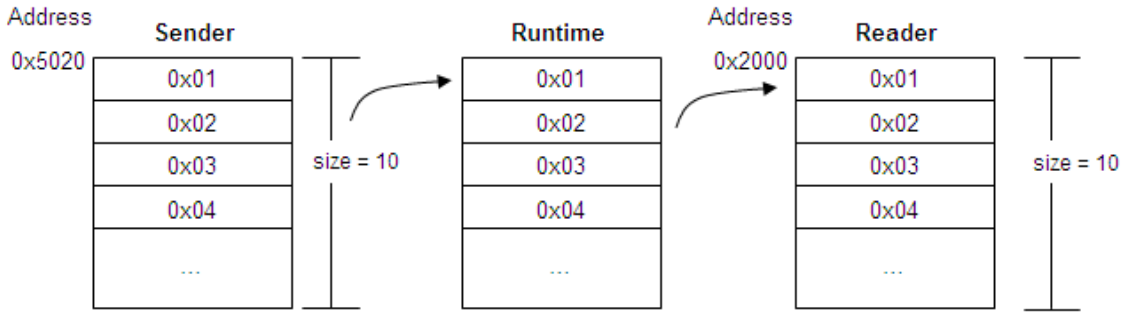
A User can send a one-way or two-way command using the *srSendCmd(..)* routine.

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.

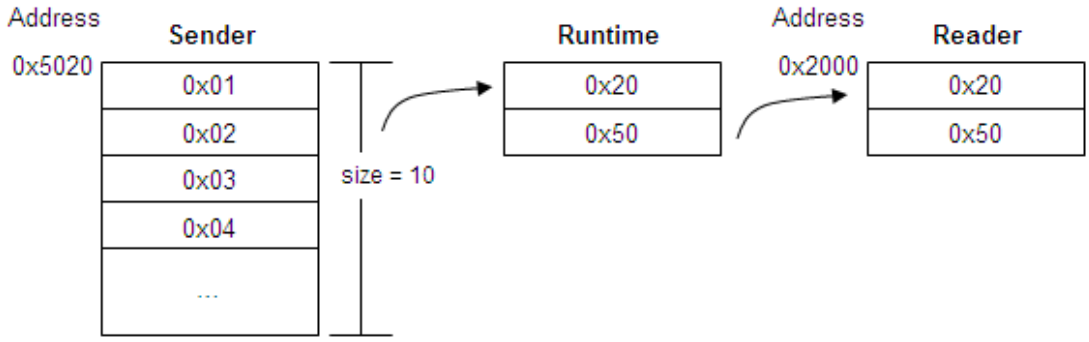
1.3.6.1. Send Type by Value

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 configued 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.

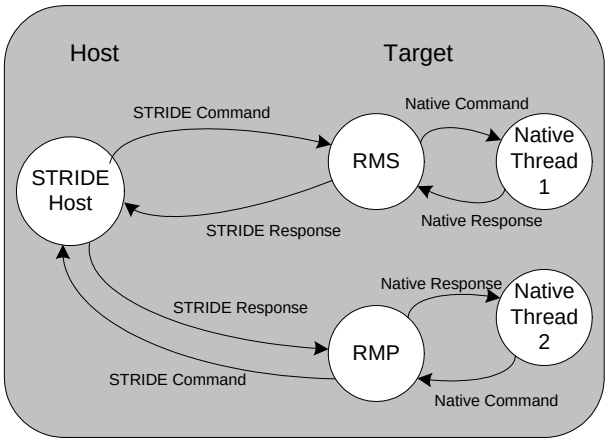


Figure 9: Remote Message Overview

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 */
                CustomerAddSubscirber( (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),
                             &MsgBuffer,
                             wSize );
            }
            else
            {
                /* send response */
                dwMsgId = dwMsgId & srSMID_SUID_MASK;
                srSendRsp( wRMS_STID,
                           dwMsgInst,
                           srDWORD(dwMsgId + srMT_ONE_RSP + srST_RSP_VAL),
                           &MsgBuffer,
                           wSize );
            }
        }
    }
}
```

2.3. Using a Remote Message Proxy (RMP)

A Remote Message Proxy (RMP) thread provides a way for your target to translate native commands sent on your target into STRIDE messages on your desktop. It will also allow you to receive STRIDE broadcasts sent from the host platform without allocating any subscriber resources on the target. Any broadcast message received over the link that does not have any subscribers will be delivered to the RMP. The RMP can then translate the received broadcast message into your native messaging.

An STID can become the RMP by calling the STRIDE Runtime API *srRegisterAccessClass* with the RMP Access Class type *srAC_REG_MESSAGES*. A valid STID and mail box ID are needed for proper routing.

By using the Force Broadcast Remote feature on the host along with the RMP on the target you can support native broadcasts from the host that will be forced over the link and intercepted by the RMP on the target. This is done without using any subscriber resources on either the host or target.

If the Force Broadcast Remote feature is enabled on the host then any broadcast that does not have a valid SUID associated with it will be routed remotely, regardless of the value of the SUID. Ensure that the SMID used has the Broadcast attribute set. Once on the target, the SUID is checked for a SUID entry. If there is no SUID entry then the STRIDE Runtime will check if there is a valid RMP registered. If so, the broadcast is sent to the RMP. The RMP is then free to route the broadcast as it sees fit to the native threads.

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 49

Messaging – page 55

Pointers – page 79

Tracing – page 96

Query – page 105

Access Class (Remote Messaging) Routines – page 112

I-block – page 113

Runtime Thread Entry and Exit Point – page 114

Host Override Routines – page 119

Connecting – page 122

Database Loading Routines – page 128

Trace Buffers – page 135

Trace Filtering – page 137

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
dwNID	Input	The Application Notification Identifier
szName	Input	Character pointer to a NULL terminated string
pwSTID	Input/Output	The STRIDE Transact Identifier
bNew	Input	Request new STID srTRUE = Create new STID dynamically srFALSE = Use STID passed in parameter list

Return Value	Description
srOK	Success
srERR_STID_ALLOC	STID allocation failed
srERR_STID_USED	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
wSTID	Input	STRIDE Transact ID from the srCreateSTID() call

Return Value	Description
srOK	Success
srERR_STID_INVALID	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
wSTID	Input	STRIDE Transact Identifier
eBox	Input	The associated mailbox for the message
dwSMID	Input	The unique SMID
bOn	Input	Indicates if registration is On or Off srTRUE = On srFALSE = Off

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

eAC	Input	Access Class of type srAccessClass_e, defined as:
		`typedef enum`
		`{`
		`  srAC_REG_MESSAGES  = 0,`
		`  srAC_REG_FUNCTIONS = 1`
		`} srAccessClass_e;`
		
		Values are *srAC_REG_MESSAGES* or *srAC_REG_FUNCTIONS*

wSTID	Input	The associated STRIDE Transact Identifier for the Access Class
eBox	Input	The associated mailbox for the Access Class
bOn	Input	Set or unset registration srTRUE = On srFALSE = Off

Return Value	Description
srOK	Success
srErr	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
wSTID	Input	STRIDE Transact ID from *srCreateSTID()* call
eBox	Input	Box from which to read
wMaxRead	Input	The max number of bytes allowable to read
pdwSMID	Output	The SMID read; 0 if nothing in box
pyBuffer	Output	Input data; address of or payload
pwSize	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 attrributes 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
wSTID	Input	STRIDE Transact ID from the *srCreateSTID()* call
dwMsgInst	Input	The sender's Message Instance Identifier

Return Value	Description
srOK	Success
srERR_STID_INVALID	STID is invalid
srERR_PTR_INVALID	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
wSTID	Input	STRIDE Transact ID from the *srCreateSTID()* call
eRspBox	Input	The Response Box Identifier
dwSMID	Input	The unique SMID
pyPayload	Input	Address (pointer) of the message payload
wSize	Input	The size of the payload

Return Value	Description
srOK	Success
srERR_STID_INVALID	STID is invalid
srERR_STID_INACTIVE	STID is inactive
srERR_REG_NONE	No local registration
srERR_PAL_MEM_ALLOC	PAL memory allocation failed
srERR_PTR_ALLOC	Pointer allocation failed
srERR_SEND_PRIV	Attempt to send private memory without pointers
srERR_PTR_POOL	Expected pool pointer memory
srERR_PTR_OVERWRITE	Payload pointer does not match pointer entry
srERR_QUEUE_FULL	Message queue full
srERR_PAL_NOTIFY_SYS	PAL notification to STRIDE Runtime failed
srERR_PAL_NOTIFY_USER	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
wSTID	Input	STRIDE Transact ID from the *srCreateSTID()* call
dwMsgInst	Input	The sender's Message Instance Identifier
dwSMID	Input	The unique SMID
pyPayload	Input	Address (pointer) of the message payload
wSize	Input	The size of the payload

Return Value	Description
srOK	Success
srERR_STID_INVALID	STID is not valid
srERR_STID_INACTIVE	Calling STID routing to inactive STID
srERR_RMT_FAIL	Attempt to route remote failed
srERR_PAL_MEM_ALLOC	PAL memory allocation failed
srERR_PTR_ALLOC	Pointer allocation failed
srERR_SEND_PRIV	Attempt to send private memory without pointers
srERR_PTR_POOL	Expected pool pointer memory
srERR_PTR_OVERWRITE	Payload pointer does not match pointer entry
srERR_PTR_DUPLICATE	Duplicate pointer set up
srERR_PTR_LOCKED	Pointer in lock state
srERR_QUEUE_FULL	Message queue full
srERR_PAL_NOTIFY_SYS	PAL notification to STRIDE Runtime failed
srERR_PAL_NOTIFY_USER	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
wSTID	Input	STRIDE Transact ID from the *srCreateSTID()* call
dwSMID	Input	The unique SMID
pyPayload	Input	Address (pointer) of the message payload
wSize	Input	The size of the payload

Return Value	Description
srOK	Success
srERR_STID_INVALID	STID is not valid
srERR_STID_INACTIVE	Calling STID routing to inactive STID
srERR_RMT_FAIL	Attempt to route remote failed
srERR_PAL_MEM_ALLOC	PAL memory allocation failed
srERR_PTR_ALLOC	Pointer allocation failed
srERR_SEND_PRIV	Attempt to send private memory without pointers
srERR_PTR_POOL	Expected pool pointer memory
srERR_PTR_OVERWRITE	Payload pointer does not match pointer entry
srERR_QUEUE_FULL	Message queue full
srERR_PAL_NOTIFY_SYS	PAL notification to STRIDE Runtime failed
srERR_PAL_NOTIFY_USER	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
wSTID	Input	STRIDE Transact Identifier
eBox	Input	The Box ID to send message to
dwSMID	Input	The unique SMID
bOn	Input	Indicates if Subscription is On or Off srTRUE = On srFALSE = Off

Return Value	Description
srOK	Success
srERR_STID_INVALID	STID is invalid
srERR_SMID_ATTR	SMID attributes are incorrect
srERR_SUID_RANGE	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
wSTID	Input	STRIDE Transact Identifier
dwAuxData	Input	The auxiliary data to set

Return Value	Description
srOK	Success
srERR_STID_INVALID	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
wSTID	Input	STRIDE Transact Identifier
pdwAuxData	Output	The auxilliary data read

Return Value	Description
srOK	Success
srERR_STID_INVALID	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             *pwHandle );
```

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.

	Each user sending a command payload containing a pointer is required to call the *srPtrSetup* routine. The owner of a response payload containing a pointer is required to call the *srPtrSetup* routine.

Parameters	Type	Description
wSTID	Input	STRIDE Transact Identifier from the srCreateSTID() call
dwSMID	Input	The unique SMID
wOffset	Input	The pointer field offset within the payload
pwOffsetTable	Input	The pointer field offset table, containing offset depth and offsets into parent payloads
wSize	Input	The byte size of the memory block pointed to.
eMsgDir	Input	The message direction: wMSGDIR_COMMAND = 0 wMSGDIR_RESPONSE = 1
ePtrDir	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
ePtrUsage	Input	The pointer usage attribute: srPTR_USAGE_PRIVATE = 0 srPTR_USAGE_POOL = 1
pwHandle	Output	The unique handle of the pointer

Return Value	Description
srOK	Success
srERR_STID_INVALID	STID is invalid
srERR_SUID_RANGE	SUID is not in the valid range
srERR_SMID_ATTR	SMID attributes are incorrect
srERR_PTR_MSG_DIR	Pointer message direction 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 *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
wSTID	Input	STRIDE Transact ID from the *srCreateSTID()* call.
wParentHandle	Input	Handle to the "parent" pointer that holds the nested (child) pointer
wOffset	Input	The pointer field offset within the parent pointer payload
wSize	Input	The byte size of the memory block pointed to.
ePtrDir	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
ePtrUsage	Input	The pointer usage attribute srPTR_USAGE_PRIVATE = 0 srPTR_USAGE_POOL = 1
pwHandle	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;

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

srWORD     wSTID, wParentHandle, 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
wSTID	Input	STRIDE Transact ID from the *srCreateSTID()* call
dwSMID	Input	The unique SMID
wHandle	Input	Handle of the pointer from srPtrSetup or srPtrSetupChild

Return Value	Description
srOK	Success
srERR_STID_INVALID	STID invalid
srERR_PTR_INVALID	Pointer handle invalid
srERR_PTR_TEARDOWN	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
wSTID	Input	STRIDE Transact Identifier frm the srCreateSTID() call
dwMsgInst	Input	Message instance
pyMemory	Input	Pointer memory address
pwHandle	Output	Handle to the pointer in the dwMsgInst

Return Value	Description
srOK	Success
srERR_STID_INVALID	STID invalid
srERR_PTR_INVALID	Pointer handle invalid
srERR_PTR_ADDRESS	Pointer address not found
srERR_PTR_LOCKED	Pointer in lock state
srERR_PTR_DIR	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
wSTID	Input	STRIDE Transact Identifier from the srCreateSTID() call
wHandle	Input	Pointer handle
wSize	Input	New size for the pointer setup information

Return Value	Description
srOK	Success
srERR_STID_INVALID	STID is invalid
srERR_PTR_INVALID	Pointer handle invalid
srERR_PTR_SIZE	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
wSTID	Input	STRIDE Transact Identifier from the srCreateSTID() call
dwSMID	Input	The unique SMID
pyPayload	Input	Command payload
wSize	Input	Command payload size
pdwCmdInst	Output	Command pointer instance

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

dwCmdInst must be deleted by calling srPtrDeleteInst.

srPtrCreateRspInst()

Create response pointer instance

Prototype

```
srWORD  srPtrCreaterRspInst ( 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
wSTID	Input	STRIDE Transact Identifier from the srCreateSTID() call
dwSMID	Input	The unique SMID
pyPayload	Input	Command payload
wSize	Input	Command payload size
dwCmdInst	Input	Command pointer instance
pdwRspInst	Output	Response pointer instance

Return Value	Description
srOK	Success
srERR_PTR_OVERWRITE	Payload pointer does not match pointer entry
srERR_PTR_ALLOC	Pointer allocation failed
srERR_PTR_INVALID	Pointer handle invalid
srERR_PTR_DUPLICATE	Pointer setup duplicate
srERR_PTR_LOCKED	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
wSTID	Input	STRIDE Transact Identifier from the srCreateSTID() call
dwPtrInst	Input	Pointer instance

Return Value	Description
srOK	Success
srERR_PTR_INVALID	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
wSTID	Input	STRIDE Transact Identifier
wSTPID	Input	The unique STRIDE TracePoint ID
pyPayload	Input	Pointer to a payload
wSize	Input	The size of the payload
eLevel	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
srOK	Success
srERR_STID_INVALID	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
wSTID	Input	STRIDE Transact Identifier.
szString	Input	A null terminated string.
eLevel	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
srOK	Success
srERR_STID_INVALID	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
dwPtrInst	Input	Handle returned by srPtrCreateCmdInst or srPtrCreateRspInst depending on eTraceType
eTraceType	Input	Interface trace type

Return Value	Description
srOK	Success
srERR_STID_INVALID	STID is invalid
srERR_TRACE_TYPE	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
szMsg	Input	A string, which can be formatted. Cannot be null.
...	Input (Optional)	Variable argument list to format the string *szFmt*.

Return Value	Description
srOK	Success
srERR	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
szMsg	Input	A string, which can be formatted. Cannot be null.
...	Input (Optional)	Variable argument list to format the string *szFmt*.

Return Value	Description
srOK	Success
srERR	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

eAC	Input	Access Class of type srAccessClass_e, defined as:
		`typedef enum`
		`{`
		`  srAC_REG_MESSAGES  = 0,`
		`  srAC_REG_FUNCTIONS = 1`
		`} srAccessClass_e;`
		
		Values are *srAC_REG_MESSAGES* or *srAC_REG_FUNCTIONS*

pwSTID	Output	The associated STRIDE Transact Identifier for the Access Class
peBox	Output	The associated mailbox for the Access Class

Return Value	Description
srOK	Success
srErr	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
DwNID	Input	Notification Identifier
pwSTID	Output	STRIDE Transact ID associated with NID

Return Value	Description
srOK	Success
srERR_NID_NOT_FOUND	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
szName	Input	Name used when creating the STID
pwSTID	Output	STRIDE Transact Identifier (STID)

Return Value	Description
srOK	Success
srERR_NAME_NOT_FOUND	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
dwSMID	Input	The unique SMID

ptSMIDInfo	Output	ptSMIDInfo is of type srSMIDInfo_t, defined as:
		`typedef struct`
		`{`
		`srBOOL bReg;`
		`srBYTE yRegSTID;`
		`srBOX_e eRegBox;`
		`srBOOL bRegOverride;`
		`srWORD wSubIdsCnt;`
		`} srSMIDInfo_t;`
		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
srOK	Success
srERR_SMID_OUT_OF_RANGE	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
wSTID	Input	STRIDE Transact Identifier
eBox	Input	The mailbox to check
ptBoxInfo	Output	The number of pending messages, the next message's SMID, and the size of the next message in the box.

Return Value	Description
srOK	Success
srERR_STID_INVALID	STID is invalid

Example

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

srWORD    wResult;
srBoxInfo  tBoxInfo;

  …

wResult = swQueryBox( 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 uninitialize 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()

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 request timeouts, the Runtime will enter the Listen state.

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

Command	Description
No payload	Used to initiate a connection

Response	Description
srConnectOpenRsp_t	Contains the results of the connection request.
eConnection	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, wMsgSize;
srDWORD             dwMySMID, dwMsgInst;
srConnectOpenRsp_t  tConnectOpenRsp;

//assumes STID already created
wResult = srSendCmd( wMySTID,  srBOX_1, CONNECT_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
srMT_TWO	Two-way message
srAC_SYS	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
srMT_BRD	Broadcast message
srST_RSP_VAL	Response payload sent by value
srAC_SYS	STRIDE Runtime system message

Response	Description
srConnectOpenRsp_t	Contains the results of the connection request

eConnection	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 = srSubcribe( 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
srMT_TWO	Two-way message
srST_RSP_VAL	Response payload sent by value
srAC_SYS	STRIDE Runtime system message

Response	Description
srConnectOpenRsp_t	Contains the results of the connection request.

eConnection	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
srMT_ONE	One-way message
srST_CMD_VAL	Command payload sent by value
srAC_SYS_STRIDE	Runtime system message

Response	Description
srHostLoadDatabaseCmd_t	Contains the database filename and path
szDatabase	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
srMT_BCST	Broadcast message
srST_RSP_VAL	Response sent by value
srAC_SYS_STRIDE	Runtime system message

Response	Description
srHostLoadDBStatusRsp_t	Database status information
szDatabase	String containing name and path to database
eStatus	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
srMT_TWO	Two-way message
srST_RSP_VAL	Response sent by value
srAC_SYS_STRIDE	Runtime system message

Response	Description
srHostLoadDBStatusRsp_t	Database status information
szDatabase	String containing name and path to database
eStatus	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
srMT_BRD	Broadcast message
srST_RSP_PTR	Response payload sent by pointer
srPU_RSP_POL	Response payload pointer uses pool memory
srAC_SYS	STRIDE Runtime system message

Response	Description
srTraceBufferRsp_t	Contains the trace buffer
wTraceBufSize	Indicates the size of the conformant array
TraceBuffer[1]	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 subsubscription
```

```
...

//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

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
srMT_ONE	One-way message
srST_CMD_VAL	Command payload sent by value
srAC_SYS	STRIDE Runtime system message

Command	Description
srTraceFilterSTIDCmd_t	Contains the STID filter
wSTID	Unique STID identifier
bGlobalDisable	Global disable any tracing for STID
bEnableMID	Enable/disable MID tracing
bPayloadMID	Enable/disable MID payloads
bEnableSTPID	Enable/disable trace point tracing
bPayloadSTPID	Enable/disable trace point payloads
eLevel	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
srMT_BRD	Broadcast message
srST_RSP_VAL	Response payload sent by value
srAC_SYS	STRIDE Runtime system message

Response	Description
srRemoteSubRsp_t	Contains the remote subscriber information
WSRID	STRIDE Response Identifier
WSUID	STRIDE Unique Identifier
BOn	Indicates whether 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_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
srMT_BRD	Broadcast message
srST_RSP_VAL	Response payload sent by value
srAC_SYS	STRIDE Runtime system message

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

Example

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

rWORD                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. 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
srMT_BRD	Broadcast message
srST_RSP_VAL	Response payload sent by value
srAC_SYS	STRIDE Runtime system message

Response	Description
srErrMarshalRsp_t	Contains the marshaling error information
wCode	Marshaling error code (for a list of marshalling error codes, refer to sr.h on page 148).
bCmd	Boolean flag to indicate direction: srTRUE = Command srFALSE = Response
dwSMID	Message identifier
wRmtSRID	STRIDE Response Identifier
bMarshalOut	Direction of the marshalling error: TRUE = outgoing marshalling error (target to host) FALSE = incoming marshalling 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. STRIDE Runtime Internals

The STRIDE Runtime is orgainzed 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.

4.1. 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.

4.2. STRIDE Runtime Modules

The STRIDE Runtime is organized into nine funtional 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	srtime

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

4.3. 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
srapirgl.c	Regional API source code
srapirgl.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 prototyes 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
srmsgrt.c	Message Routing source code
srmsgrt.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
srsuid.c	SUID source code
srsuid.h	SUID prototypes and defines
srthread.c	Runtime Thread source code
srthread.h	Runtime Thread source code prototypes and defines
srtime.c	Timer source code
srtime.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)
srtestpp.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

#include "pal.h"

#if (defined(__WIN32__) || defined(_WIN32)) && !defined(STRIDE_STATIC)

# ifdef STRIDE_EXPORTS
#  define srEXPORT __declspec(dllexport)
# else
#  define srEXPORT __declspec(dllimport)
# endif

#else

# define srEXPORT

#endif


#ifdef __cplusplus
extern "C" {
#endif

/*****************************************************************************
 *
 * 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 managment prototypes */
typedef void* (*srPoolMemAlloc_t)(srWORD wSize);
typedef void (*srPoolMemFree_t)(void* pvMem);


/*****************************************************************************
 *
 * 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;

/* Function Double */
typedef void (*srFunctionDouble_t)(void);

/*****************************************************************************
 *
 * 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 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 srUninit( void );

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

srEXPORT srWORD srDeleteSTID( srWORD  wSTID );


/*****************************************************************************
 *
 * Messaging API
 *
 *****************************************************************************/

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

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

srEXPORT srWORD srReadComplete( srWORD    wSTID,
                                srDWORD   dwMsgInst );

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

srEXPORT srWORD srSendRsp( srWORD     wSTID,
```

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

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

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

srEXPORT srWORD srSetAuxData( srWORD  wSTID,
                              srDWORD dwAuxData );

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


/*****************************************************************************
 *
 * Pointer API
 *
 *****************************************************************************/

srEXPORT srWORD srPtrSetup( srWORD        wSTID,
                            srDWORD       dwSMID,
                            srWORD        wOffset,
                            srWORD*       pwOffsetTable,
                            srWORD        wSize,
                            srMsgDir_e    eMsgDir,
                            srPtrDir_e    ePtrDir,
                            srPtrUsage_e  ePtrUsage,
                            srWORD*       pwHandle );

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

srEXPORT srWORD srPtrTearDown( srWORD    wSTID,
                               srDWORD   dwSMID,
                               srWORD    wHandle );

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

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

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

srEXPORT srWORD srPtrCreateRspInst( srWORD    wSTID,
                                    srDWORD   dwSMID,
                                    srBYTE*   pyPayload,
                                    srWORD    wSize,
                                    srDWORD   dwCmdInst,
                                    srDWORD*  pdwRspInst );
```



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


/*****************************************************************************
 *
 * Tracing API
 *
 *****************************************************************************/

srEXPORT srWORD srTracePoint( srWORD      wSTID,
                              srDWORD     dwTPID,
                              srBYTE*     pyPayload,
                              srWORD      wSize,
                              srLevel_e   eLevel );

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

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


/*****************************************************************************
 *
 * Printing API
 *
 *****************************************************************************/

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

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


/*****************************************************************************
 *
 * Query API
 *
 *****************************************************************************/

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

srEXPORT srWORD srQueryName( const srCHAR*    szName,
                             srWORD*          pwSTID );

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

srEXPORT srWORD srQueryBox( srWORD        wSTID,
                            srBOX_e       eBox,
                            srBoxInfo_t*  ptBoxInfo );

srEXPORT srWORD srQuerySTID( srDWORD* pdwNID,
                             srWORD   wSTID );

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

/*****************************************************************************
 *
 * Remote Message Stub/Proxy API
 *
 *****************************************************************************/
```

```
/* Deprecated in favor of srRegisterAccessClass() */
srEXPORT srWORD srSetRMS( srWORD    wSTID,
                          srBOX_e   eBox );

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


/*****************************************************************************
 *
 * Runtime Thread Entry / Exit Points
 *
 *****************************************************************************/

srEXPORT void srThread( void* param );
srEXPORT void srThreadInit( void );
srEXPORT void srThreadUninit( void );
srEXPORT srBOOL srThreadProc( void );


/*****************************************************************************
 *
 * Transport API
 *
 *****************************************************************************/

typedef srBOOL (*srRmtRouteOutCallback_t)( const srBYTE *pyBuffer, srWORD wSize );

srEXPORT srRmtRouteOutCallback_t srRmtRouteOutReg( srRmtRouteOutCallback_t ptFuncCb );


/*****************************************************************************
 *
 * Host Specific Runtime Routines
 *
 *****************************************************************************/

#ifdef srHOST
srEXPORT void srHostIMPoolMemOverride(srPoolMemAlloc_t pfMemAlloc,
                                      srPoolMemFree_t  pfMemFree);
#endif /* srHOST */


/*****************************************************************************
*
* Function Double Routines
*
*****************************************************************************/


srEXPORT srBOOL _srGetFnDbl(const srCHAR* szName, srFunctionDouble_t* ptFnDbl);
#define srDOUBLE_GET(fn, pfnDbl) _srGetFnDbl(#fn, (srFunctionDouble_t*)pfnDbl)

srEXPORT srBOOL _srSetFnDbl(const srCHAR* szName, srFunctionDouble_t tFnDbl);
#define srDOUBLE_SET(fn, fnDbl) _srSetFnDbl(#fn, (srFunctionDouble_t)fnDbl)

srEXPORT srBOOL _srRegFnDbl(const srCHAR* szName, srFunctionDouble_t* ptFnDblRef);

#ifdef __cplusplus
}
#endif


/*****************************************************************************************
*
 *
 * C/C++ features support
 *
```

```
*******************************************************************************
/

#if !defined(srCRT_HAS_WCHAR_T)
#define srCRT_HAS_WCHAR_T           1    /* wchar_t defined (1=enabled, 0=disabled)*/
#endif

#if !defined(srCRT_HAS_LONG_LONG)
#define srCRT_HAS_LONG_LONG         1    /* long long defined (1=enabled, 0=disabled)*/
#endif

#if !defined(srCRT_HAS_LONG_DBL)
#define srCRT_HAS_LONG_DBL          1    /* long double defined (1=enabled, 0=disabled)*/
#endif

#if !defined(srCRT_HAS_VAR_ARGS)
#define srCRT_HAS_VAR_ARGS          1    /* variable arguments (1=enabled, 0=disabled)*/
#endif

#ifdef __cplusplus

#if !defined(srCPP_HAS_NAMESPACE)
#define srCPP_HAS_NAMESPACE         1    /* C++ namespaces (1=enabled, 0=disabled)*/
#endif

#if srCPP_HAS_NAMESPACE
# define srCPP_NAMESPACE_BEGIN(name) namespace name {
# define srCPP_NAMESPACE_END  }
# define srCPP_NAMESPACE_USE(name) using namespace name;
# define srCPP_NAMESPACE_QUALIFIER(name) name::
#else
# define srCPP_NAMESPACE_BEGIN(name)
# define srCPP_NAMESPACE_END
# define srCPP_NAMESPACE_USE(name)
# define srCPP_NAMESPACE_QUALIFIER(name)
#endif

#endif /* __cplusplus */


#endif  /* SR_H */
```

Appendix B: 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              32   /* total number of subscribers at one time */
#define srCFG_TOTAL_PTRS               256  /* total number of pointer entries         */

#define srCFG_SUID_TABLE_TYPE          1    /* 1 = Search based, 0 = Index based       */
#define srCFG_SUID_TABLE_SIZE          32   /* number of SUID Table entries            */
#define srCFG_SUID_OVERRIDE            1    /* override (1 = enabled, 0 = disabled)    */
#define srCFG_SUID_OVERRIDE_STORAGE    0    /* override registration storage allocation*/

#define srCFG_TOTAL_SUIDS_QUED         128  /* total number of SUIDS queued at one time*/
#define srCFG_STID_NAME_SIZE           15   /* max size of a STRIDE Transact ID name   */


/*****************************************************************************
 *
 * Tracing
 *
 *****************************************************************************/

#define srCFG_TRACING_ENABLED          1    /* tracing enabled  (1=enabled, 0=disabled)*/
#define srCFG_TOTAL_TRACING_MEMORY     1024 /* 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 = microsecs, 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       512  /* 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)*/

/******************************************************************************
 *
 * Test Settings
 *
 ******************************************************************************/

#define srCFG_MAX_TEST_NAME_SIZE       128  /* max number of bytes to store test name
(name, label...) */
#define srCFG_MAX_TEST_VALUE_SIZE      512  /* max number of bytes to store test value
(comment, description, payload...) */

#define srCFG_MAX_UNIQUE_TEST_POINT    256  /* max number of simultaneously (un)expected
                                               unique test points */

/******************************************************************************
 *
 * 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


/*******************************************************************************
 *
 * Debug Settings
 *
 ******************************************************************************/

#define srCFG_ERROR_CHECK_LEVEL        2    /* levels(0,1,2), none(0)                */


#endif /* SRCFG_H */
```