STRIDE™ Communication Language Reference

Revision 5.6 (2008-09-24)

Published by

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

Contents

1 Introduction

The STRIDE Communication Language, or SCL, is an Interface Definition Language (IDL) used by the STRIDE Communication Model (SCM) to identify and define messages, function calls, and trace points within an application. Interfaces are identified and annotated in order to expose enough information about them so that they can be marshaled across platform boundaries. The implementation code of a defined interface must adhere to the contract defined in SCL. This document describes, in detail, the syntax and semantics of this contract.

The SCL is a superset of the ANSI C language. SCL extensions to the C language are a set of pragmas that allow interface semantics to be more precisely prescribed than the C Language allows.

- Message-based interfaces require unique identification of the message, definition of the sender/receiver relationship, and description of the data (if any) that is part of the message payload.
- Functional interfaces require unique identification of the function and a description of the parameters and return values.
- Trace points require unique identification of the tracepoint, assignment of a name, and description of the optional associated data.

SCL is a superset of the ANSI C Standard (ISO/IEC 9899:1999). Primarily, the extensions consist of new pragma directives. This section formally describes the SCL Pragmas. The context for this description is the ANSI C Standard (ISO/IEC 9899:1999). Many terms used within this document have meaning as defined by the standard.

An SCL Compliant compiler automatically defines the symbol "_SCL."

1.1 Notation and Definitions

1.1.1 Syntax

We borrow the notation used to describe the language syntax from the ANSI C Standard. Syntactic categories (nonterminals) are indicated by *italic type*, and literal words and character set members (terminals) by **bold type**. A colon (:) following a nonterminal introduces its definition. Alternative definitions are listed on separate lines.

When syntactic categories are referred to in the main text, they are not italicized and words are separated by spaces instead of hyphens.

1.2 Concepts

The SCL Pragmas are designed to allow annotation of C language constructs in such a way as to identify and define trace points, messages and function calls so that they can be

transparently intercepted and remoted. Some pragmas are necessary because C language constructs lack important interface details. Other pragmas are necessary because they allow STRIDE to do a better job of representing information to the user or because they allow STRIDE to take advantage of increased opportunities for efficiency. Finally, some exist simply to make it easier for users to adopt STRIDE without changing existing source code.

Pragmas are used to elaborate all of the following areas:

- The set of messages and functions within the application that are candidates for interception and remoting.
- Use of pointers. Pointer-related ambiguities include memory ownership, data directions, and pointers that point to a sequence of elements rather than a single element.
- The convention of using null-terminated strings in C. When a pointer is declared as “char *” it must be further elaborated as pointing to a single character or a series of characters ending in NULL.
- Pointers to functions. Functions whose addresses are passed as parameters must be identified.
- Pointers to data which should be treated by STRIDE as “void*” rather than its declared type. This may be necessary because of self-referential data structures or simply because of efficiency concerns.
- Unions. The discriminant (if any) must be identified; the relationship between active union members and discriminant values must be prescribed.
- Convenience pragmas to constrain the allowable values of a data item.
- Convenience pragmas to allow STRIDE to treat a data item as if it was defined to be of another type.
- Pragmas to allow STRIDE to identify certain structures (e.g., v-tables) used in C programs to simulate C++ inheritance and virtual function mechanisms.
- The STRIDE-defined constructs known as trace points.
- The special construct known as conformant arrays (aka variable length arrays).

1.2.1 Remotable Functions and Messages (Interfaces)

SCL includes pragmas used to identify messages or functions that may be remoted. Only messages or functions identified with these pragmas are remotable candidates. These remotable candidates are also referred to as interfaces. Almost all other pragmas operate on, or depend upon, the set of interfaces identified as remotable.

By far, the most important aspect of a remotable interface is the memory layout of the data that is passed from message sender or function caller. Most pragmas allow the data format to be described in detail. Throughout this document, the data exchanged for a given instance of a message or function is referred to as the payload.

Since STRIDE represents communication via both messaging and function calls, we introduce two terms to generically identify the sender/receiver of a message or the caller/callee of function. These terms are *user* and *owner*. Within a message, the user is the agent sending the message and owner is the agent that receives the message. Within a function call, the user is the caller and the owner is the callee.

1.2.1.1 Payloads

The data passed between user and owner is referred to as a payload. Not all messages or functions contain a payload. Consider the function

```
void f(void);
```

This function has no return value and no parameters, thus no payload.

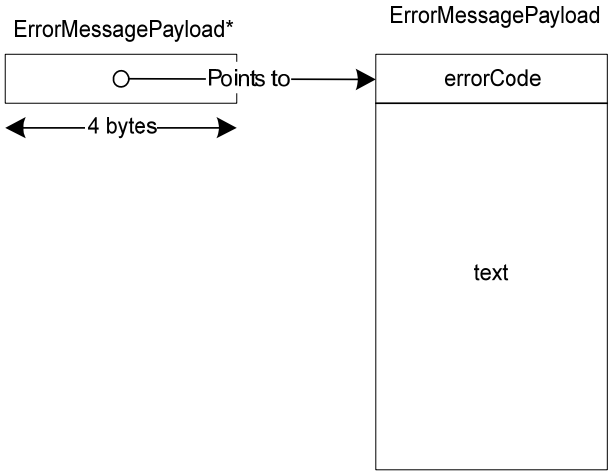
Message Payloads

Message payloads are defined by a C datatype. For example, we may have message ERROR that has an associated structure containing the error code and a string. The relevant C Language declarations might be:

```
#define ERROR_MSG  23  // the message id for ERRROR

// The structure for the message data.
typedef struct _ErrorMessagePayload
{
   int        errorCode;
   char[100]  text;
} ErrorMessagePayload;
```


Message payloads are typically passed by reference (but can also be passed by value). Assuming the message payload above is passed by reference we can depict the memory layout of the entire message payload as:



Note that in this instance, the payload actually consists of two distinct memory blocks – a pointer and the structure pointed to. The distinct blocks in a payload are referred to as the payload memory blocks. Note also that the blocks form a tree structure with each contiguous memory block representing a node in the tree. This illustrates a more general concept, that the payload blocks form a network and that network must be a tree (i.e., it starts with a node designated as the root and all other nodes are reachable from the root and there can be no cycles)

Within a two-way message, the structure of data passed in each direction may be different. For this reason, payloads must be further characterized according to whether they are passed from user to owner or owner to user. Payloads passed from user to owner are called Command Payloads. Payloads passed from owner to user are called Response Payloads.

Function Payloads

Functions are much like two-way messages, data can be passed from user (caller) to the owner (callee) at the time of the call, and the owner (callee) may pass data back to the caller (user) at the time of the return. As with message payloads, the data passed from user to owner is referred to as the Command Payload and the data passed from owner to user is referred to as the Response Payload.

Command Payloads are constructed from the formal parameters of a function. Response Payloads are constructed from the return value of a function (certain details of the response payload are omitted in this discussion but taken up later).

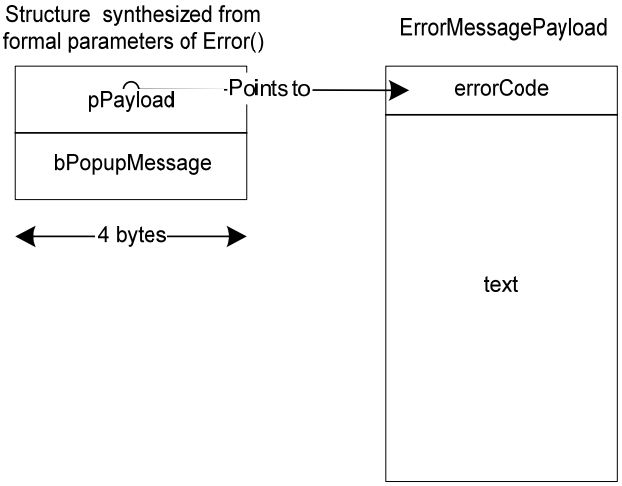
The Command Payload for a function is a synthesized C structure. The members of the structure correspond to the formal parameters of the function (both the order and types of the members exactly correspond).

The Response Payload for a function corresponds to the return value. Below is the depiction of the Command and Response Payload memory layout for a simple function Error():

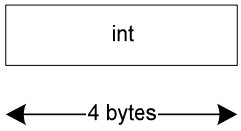
```
// The structure for the message data.
typedef struct _ErrorMessagePayload
{
   int         errorCode;
   char[100]   text;
} ErrorMessagePayload;

int Error(ErrorMessagePayload *pPayload, BOOL bPopupMessage);
```

Command Payload



Response Payload



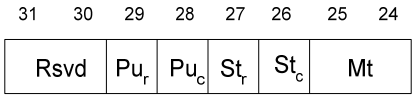
1.2.2 Messages

1.2.2.1 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.

SMID Attributes



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 (CMD)	0
One-way (RSP)	1
Two-way (TWO)	2
Broadcast (BRD)	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 following tables describe the ST attribute settings:

Send Type for Command (STc) Attributes

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

Send Type for Response (STr) Attributes

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

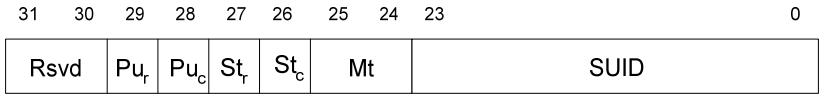
Meaning	Value
Pool (POL)	0
Private (PRI)	1

Pointer Usage for Response (PUr) Attribute

Meaning	Value
Pool (POL))	0
Private (PRI)	1

The two highest order bits of the SMID are reserved. .

These attributes, along with the 24 bit SUID are all packed into the STRIDE Message ID (SMID) as illustrated below:



In all user-constructed SMIDs the Rsvd bits are required to be 0.

1.2.2.2 STRIDE Unique ID (SUID)

The SUID is composed of the low order, 24-bit value found in a SMID. The SUID is unique across the space of messages and functions. In this way, it is possible to uniquely identify a message or function simply by knowing its SUID.

1.2.3 Pointers within Payloads

Without exception, every pointer that is part of a payload must have several defining attributes:

- Pointer Data Directional Attributes
- Memory Ownership Attributes
- Pointed to Element Count

1.2.3.1 Pointer Data Directional Attributes

The table below defines the allowed directional attributes and their meanings.

Pointer Data Directional Attributes

Values	Meaning
IN	When a pointer is tagged with the IN directional attribute it means that it points to a block of memory that must be allocated and assigned values by the user. When an owner receives a pointer that is tagged with the IN directional attribute, the owner can assume that it points to a block of memory with values as set up by the user. A block of memory pointed to by an IN pointer is said to be an IN block. Any changes an owner makes to values within an IN block will never be reflected back to the user.
OUT	When a pointer is tagged with the OUT directional attribute it means that it points to a block of memory allocated by the user, but whose values are setup by the owner. A block of memory pointer to by an OUT pointer is said to be an OUT

Values	Meaning
	block. When a user allocates an OUT block, the user need not define or setup any particular initial values. When such a block is received by an owner, the owner must assume that all memory within the block is not initialized and holds undefined values. The owner is obligated to write to the OUT block to setup all relevant data values. All values set up by the owner within an OUT block will be reflected back to the user.
INOUT	When a pointer is tagged with the INOUT directional attribute, it means that it points to a block of memory that must be allocated by the user, and is expected to be assigned values by both the user and owner. A block of memory pointed to by an INOUT pointer is said to be an INOUT block. The user for the INOUT block is obligated to setup initial values within the block. All values within the block will be reflected to the owner. The owner for an INOUT block can assume that all data values setup by the user are reflected in the block received. If the owner makes any changes to the data within the INOUT block the values will be reflected back to the user.
RETURN	When a pointer is tagged with the RETURN directional attribute it means that the owner is responsible for the allocation of the block, the setup of its initial values and communication of the pointer value back to the user. A block of memory pointed to by a RETURN pointer is referred to as a RETURN block. When a pointer is tagged with the RETURN attribute, both the value of the pointer and the pointed to block of memory are set up by the owner. Both are communicated back to the user at the return of the function call. At the time of a return the user will receive both the value of the RETURN pointer and the block pointed to. The block will have values as setup by the owner.
INRETURN	When a pointer is tagged with the INRETURN directional attributes it means that value of the pointer and the value of the pointed to block are setup by both user and owner. There is no single block of memory pointed to by an INRETURN pointer. Rather there is one block on the call (or message send) and a second block on the return (or completion of the two-way message). Thus the first block has the characteristics of an IN block and the second the characteristics of a

Values	Meaning
	RETURN block. A pointer tagged with INRETURN must point to a block allocated and setup with initial values by the user. All values setup by the user will be reflected to the owner upon receipt of the call or message. The owner receiving such a pointer may do any of the following: • Change neither the pointer nor the values of the pointed to block. In this case the user will receive exactly the values setup for the call or message send. • Change values within the pointed to block. In this case, the user will receive the same pointer value sent, but the values within the pointed to block will change according to the owner changes. • Change the values of the pointer (i.e., allocate another block) and set up values within the new block. The user will receive both the new value for the pointer and pointed to block

1.2.3.2 Memory Ownership Attributes

In addition to directional attributes, all pointers also must have memory ownership attributes. The allowable attributes are shown below.

Pointer Memory Ownership Attributes

Values	Meaning
`PRIVATE`	Ownership/management responsibility of the memory to which the pointer points belongs to the allocating agent. Depending on the directional attribute of the pointer the allocating agent may be either the user or owner. The pointed to memory may be either global, automatic or heap based since ownership is solely the responsibility of the allocator.
`POOL`	Ownership/management responsibility of the memory to which the pointer points is transferred along with the pointer. The receiving agent is responsible for de-allocation of the memory to which the pointer points. For this reason, pool memory must be dynamically allocated and cannot be static or automatic. The receiving agent can be either owner or user depending up on the directional attributes of the pointer. Pool memory ownership further implies the existence of a centralized “Pool” of memory to which both owner and user have access and a set of common allocation/deallocation routines.

1.2.3.3 Allowed Combinations of Memory Ownership and Directional Attributes

Not all directional attribute values or memory ownership attributes are possible for every pointer. Some directional attributes may be applied only to a pointer residing in a block with certain attributes. And some memory ownership attributes are prohibited for some directional attribute values.

Directional Attributes Allowed

Pointer Attribute	Allowable Memory Usage Attributes	
IN	PRIVATE or POOL	
OUT	PRIVATE	POOL is not allowed for an OUT pointer because memory ownership cannot be transferred.
INOUT	PRIVATE	POOL is not allowed for an INOUT pointer because memory ownership cannot be transferred.
RETURN	PRIVATE or POOL	
INRETURN	PRIVATE or POOL	

Directional Attributes Allowed

Block Type in Which Pointer Resides	Allowable Pointer Attributes	Allowable Memory Usage Attributes
IN	IN	PRIVATE or POOL
OUT	RETURN	PRIVATE or POOL
INOUT	IN	PRIVATE or POOL
	OUT	PRIVATE
	INOUT	PRIVATE
	RETURN	PRIVATE or POOL
	INRETURN	PRIVATE or POOL
RETURN	RETURN	PRIVATE or POOL
INRETURN	IN	PRIVATE or POOL
	RETURN	PRIVATE or POOL
	INRETURN	PRIVATE or POOL

Special Cases

All top levels pointers (i.e., pointers which reside in the root block of a command payload) have allowable pointer attributes as IN, OUT or INOUT.

All blocks in a response payload must be RETURN. Response payloads include:

- The return value of a function
- The response payload of a two-way message
- The payload of a one-way response message
- The payload of a broadcast message

If the return value of a function is a pointer, then that pointer must have the RETURN pointer attribute, either implicitly or explicitly. It can have either POOL or PRIVATE memory usage.

1.2.3.4 Sized pointers

The C language allows a pointer to point to a series of elements, rather than a single element. As a result, it is not clear how many elements a pointer declared as "T *" points to. However, this is an important detail of a payload. A pointer must be characterized as pointing to a single element or a series of elements. Those pointing to a series of elements are called "sized pointers." Sized pointers are broken down into two additional

subtypes: Those which point to a fixed sized series and those who point to a series whose length is characterized by another field referred to as the count field. The pragmas scl_ptr() and scl_sized_ptr() allow these characterizations.

Sized Pointer Memory Allocation Policies

In most cases, a block of memory pointed to is of known, fixed size. This is typically the case for pointers that point to a single element. However, this is generally not the case for sized pointers. Sized pointers point to a series of elements, and sometimes have an associated field whose value indicates the current element count. In some cases, it is not clear if memory is available for elements beyond the current count. Conventions must be adopted in order to clarify the user and owner memory management responsibilities. The conventions specify the required sizes of allocated blocks and responsibility for initial values.

Memory Conventions for Fixed Size Sized Pointer Pointing to Max Elements

Pointer Directional Attribute		User Obligations for Pointer Setup	Owner Assumption for Received Pointer	Owner Obligation For Returned Pointer	User Assumption For Receipt of Returned Pointer
IN	BlockSize	Max	Max	n/a	n/a
	ElemCount	Max	Max	n/a	n/a
OUT	BlockSize	Max	Max	Max	Max
	ElemCount	0	0	Max	Max
INOUT	BlockSize	Max	Max	Max	Max
	ElemCount	Max	Max	Max	Max
INRETURN	BlockSize	Max	Max	Max	Max
	ElemCount	Max	Max	Max	Max
RETURN	BlockSize	n/a	n/a	Max	Max
	ElemCount	n/a	n/a	Max	Max

Memory Conventions for Sized Pointer with Associated Size Field residing in IN block.

Pointer Directional Attribute		User Obligations for Pointer Setup	Owner Assumption for Received Pointer	Owner Obligation For Returned Pointer	User Assumption For Receipt of Returned Pointer
IN	BlockSize	$Count_{in}$	$Count_{in}$	n/a	n/a
	ElemCount	$Count_{in}$	$Count_{in}$	n/a	n/a
OUT	BlockSize	$Count_{in}$	$Count_{in}$	$Count_{in}$	$Count_{in}$
	ElemCount	0	0	$Count_{in}$	$Count_{in}$
INOUT	BlockSize	$Count_{in}$	$Count_{in}$	$Count_{in}$	$Count_{in}$
	ElemCount	$Count_{in}$	$Count_{in}$	$Count_{in}$	$Count_{in}$
INRETURN	BlockSize	$Count_{in}$	$Count_{in}$	$Count_{in}$	$Count_{in}$
	ElemCount	$Count_{in}$	$Count_{in}$	$Count_{in}$	$Count_{in}$
RETURN	BlockSize	n/a	n/a	$Count_{in}$	$Count_{in}$
	ElemCount	n/a	n/a	$Count_{in}$	$Count_{in}$

$Count_{in}$ denotes the value of the sized field at the time the user called the owner

Memory Conventions for Sized Pointer with Associated Size Field residing in OUT or RETURN block.

Pointer Directional Attribute		User Obligations for Pointer Setup	Owner Assumption for Received Pointer	Owner Obligation For Returned Pointer	User Assumption For Receipt of Returned Pointer
IN	BlockSize ElemCount	Prohibited in SCL	Prohibited in SCL	Prohibited in SCL	Prohibited in SCL
OUT[1]	BlockSize ElemCount	Max 0	Max 0	Reduced[2] $Count_{out}$	Max $Count_{out}$
INOUT	BlockSize ElemCount	Prohibited in SCL	Prohibited in SCL	Prohibited in SCL	Prohibited in SCL
INRETURN	BlockSize ElemCount	Prohibited in SCL	Prohibited in SCL	Prohibited in SCL	Prohibited in SCL
RETURN	BlockSize ElemCount	n/a n/a	n/a n/a	$Count_{out}$ $Count_{out}$	$Count_{out}$ $Count_{out}$

[1] Under certain conditions, the User and Owner obligations and assumptions indicated by this row can be relaxed. A cooperating User and Owner may privately agree to a convention consisting of a Max size block that is less than that called for in the scl_sized_ptr() pragma. In other words, the user may safely allocate less than Max for the OUT block, if there is a private agreement between user and owner that owner will never return more than $Count_{out}$ max elements. This special case is supported by Stride Host components.

[2] Reduced means all Stride components which act as owners will inform the host runtime when the $Count_{out}$ is less than the Max. This allows the host runtime to then marshall only $Count_{out}$ back to user from owner. (If the Stride components did not take this action, then Max would be marshalled back) This reduction of OUT going payload by owner is not required by the interface contract. Rather, it is an optional efficiency provided by Stride Components acting as owners. Other components (those created by the embedded developer) acting as owner do not have to perform this convenience. In addition, this reduction of an OUT block payload does not change the udnerlying size of the OUT block allocated by the User. It only affects the fraction of the block written to when the ower returns the OUT block results. When the owner reduces the outgoing block, the bytes beyond the reduction threshold will not be written upon the return from owner. It is this Stride host component behavior that enables co-operating Users and Owners to implement the “private” maxsize agreement described in the previous footnote.

Memory Conventions for Sized Pointer with Associated Size Field residing in INOUT or INRETURN block.

Pointer Directional Attribute		User Obligations for Pointer Setup	Owner Assumption for Received Pointer	Owner Obligation For Returned Pointer	User Assumption For Receipt of Returned Pointer
IN	BlockSize	$Count_{in}$	$Count_{in}$	n/a	n/a
	ElemCount	$Count_{in}$	$Count_{in}$	n/a	n/a
OUT[3]	BlockSize	Max	Max	Reduced	Max
	ElemCount	0	0	$Count_{out}$	$Count_{out}$
INOUT	BlockSize	Max	Max	Reduced	Max
	ElemCount	$Count_{in}$	$Count_{in}$	$Count_{out}$	$Count_{out}$
INRETURN	BlockSize	$Count_{in}$	$Count_{in}$	$Count_{out}$	$Count_{out}$
	ElemCount	$Count_{in}$	$Count_{in}$	$Count_{out}$	$Count_{out}$
RETURN	BlockSize	n/a	n/a	$Count_{out}$	$Count_{out}$
	ElemCount	n/a	n/a	$Count_{out}$	$Count_{out}$

1.2.3.5 Default Directional and Memory Usage Attributes

In the absence of any pragma, all pointer directional attributes (and pointer types) default to IN/PRIVATE except for pointers in response payloads. Response payloads include the return value of a function, or the response payload of a two-way comand, a one-way respose, or a broadcast message type. Pointers in response payloads default to RETURN/PRIVATE. These defaults can be overridden by an explicit pragma applied to the pointer or its type symbol. The defaults can also be overridden by application of explicit directional attributes to some ancestor pointer. The rules for overriding the defaults are prescribed below.

Assume we have a pointer p. When a pragma is applied to p, it may have an effect on the directional attributes of some downstream pointers (downstream pointers are pointer instances located within any block that p could point to directly or indirectly). The effect is to all downstream pointers along any path from p up to, but not including, the first pointer that has had an explicit pragma applied and that pragma identifies a set of instances that are contained within the set that p may point to. We call these descendents of p the "default descendents." The effects are as follows:

- When p is designated as IN, there are no changes to the default descendents.
- When p is designated as OUT, all default descendents change to RETURN.
- When p is designated as INOUT, all default descendents change to INOUT.

[3] This row allows the same special user convention of a privately agreed to $Count_{out}$ max as the table above entitled "Memory Conventions for Sized Pointer with Associated Size Field residing in OUT or RETURN."

- When p is designated as RETURN, all default descendents change to RETURN.
- When p is designated as INRETURN, all default descendents change to INRETURN.
- The default memory attribute PRIVATE of a pointer never changes.

The following section shows various SCL examples that illustrate the use of the default rules. Each example has two forms. The original the and equivalent with explicit pragmas.

Example 1

Original (Valid)	**Explicit Equivalent**

`void f(int **p);`
`#pragma scl_function(f)`
	`void f(int **p);`
	`#pragma scl_function(f)`
	`#pragma scl_ptr(f.p, IN, PRIVATE)`
	`#pragma scl_ptr(*f.p, IN, PRIVATE)`

Example 1a

Original (Valid)	**Explicit Equivalent**

`void f(int **p);`
`#pragma scl_function(f)`
`#pragma scl_ptr(f.p, RETURN, PRIVATE)`
	`void f(int **p);`
	`scl_function(f)`
	`#pragma scl_ptr(f.p, RETURN, PRIVATE)`
	`#pragma scl_ptr(*f.p, RETURN, PRIVATE)`

The designation of RETURN for p changes the downstream defaults.

Example 1b

Original (Invalid)	**Explicit Equivalent**

`void f(int **p);`
`#pragma scl_function(f)`
`#pragma scl_ptr(*f.p, RETURN, PRIVATE)`
	`void f(int **p);`
	`scl_function(f)`
	`#pragma scl_ptr(f.p, IN, PRIVATE)`
	`#pragma scl_ptr(*f.p, RETURN, PRIVATE)`

The designation of RETURN for *f.p is incorrect because the parent, f.p, is IN by default rules.

Example 2

Original (Invalid)	**Explicit Equivalent**

`typedef struct { int *p;} S;`
`#pragma scl_ptr(S.p, RETURN, PRIVATE)`
`void f(S* ps);`
	`typedef struct { int *p;} S;`
	`void f(S* ps);`
	`#pragma scl_function(f)`


```
#pragma scl_function(f)
```

```
#pragma scl_ptr(f.ps, IN, PRIVATE)
#pragma scl_ptr(f.ps->p, RETURN, PRIVATE)
```

The scl_ptr pragma applies to all fields p within objects of type S. Thus, it eliminates the default rules for S.p. S.p is always RETURN unless another pragma is used to override (which is not the case for this example).

Example 2a

Original (Valid)

```
typedef struct { int *p;} S;
#pragma scl_ptr(S.p, RETURN, PRIVATE)
void f(S* ps);
#pragma scl_function(f)
#pragma scl_ptr(f.ps, OUT, PRIVATE)
```

Explicit Equivalent

```
typedef struct { int *p;} S;
void f(S* ps);
#pragma scl_function(f);
#pragma scl_ptr(f.ps, OUT, PRIVATE)
#pragma scl_ptr(f.ps->p, RETURN, PRIVATE)
```

The scl_ptr pragma applies to all fields p within objects of type S. Thus, it eliminates the default rules for S.p. S.p is always RETURN unless another pragma is used to override (which is the case for this example).

Example 3

Original (Valid)

```
typedef struct S1 {int *pi;} S1;
typedef struct S2 {S1* ps1;} S2;
typedef struct S3 {S1* ps1_2; } S3;
#pragma scl_ptr(S3.ps1_2, INOUT, PRIVATE)
#pragma scl_ptr(S3.ps1_2->pi, OUT, PRIVATE)
void g(S2* gps2);
void h(S3* hps3);
#pragma scl_function(g)
#pragma scl_function(h)
#pragma scl_ptr(h.hps3, INOUT, PRIVATE);
```

Explicit Equivalent

```
typedef struct S1 {int *pi;} S1;
typedef struct S2 {S1* ps1;} S2;
typedef struct S3 {S1* ps1_2; } S3;
void g(S2* gps2);
void h(S3* hps3);
#pragma scl_function(g)
#pragma scl_ptr(g.gps2, IN, PRIVATE)
#pragma scl_ptr(g.gps2->ps1, IN, PRIVATE)
#pragma scl_ptr(g.gps2->ps1->pi, IN,
PRIVATE)
#pragma scl_function(h)
#pragma scl_ptr(h.hps3, INOUT, PRIVATE)
#pragma scl_ptr(h.hps3->ps1_2, INOUT,
PRIVATE)
#pragma scl_ptr(h.hps3->ps1_2->pi, OUT,
PRIVATE)
```

Illustrates the use of a pragma for S3.ps1_2->pi, that does not change the default of S1.pi.

Example 4

Original (Invalid)

```
typedef struct S1 {int *pi;} S1;
#pragma scl_ptr(S1.pi, IN, PRIVATE)
S1 f(void);
#pragma function(f)
```

Explicit Equivalent

```
typedef struct S1 {int *pi;} S1;
S1 f(void);
#pragma function(f)
#pragma scl_ptr(f().pi, IN, PRIVATE)
```

scl_ptr() is applies to every pi within any S1, thus effectively eliminating the default for S1.pi. Subsequently, this is invalid SCL because f1().pi must be RETURN.

Example 4a

Original (Valid)	Explicit Equivalent

`typedef struct S1 {int *pi;} S1;`
`#pragma scl_ptr(S1.pi, IN, PRIVATE)`
`S1 f(void);`
`#pragma function(f)`
`#pragma scl_ptr(f().pi, RETURN, PRIVATE);`
	`same`

scl_ptr() is applies to every pi within any S1, thus effectively eliminating the default for S1.pi. However, the pragma applied to f().pi is more specific and thus overrides it.

Example 5

Original (Valid)	Explicit Equivalent

`typedef struct S1 {int *pi;} S1;`
`typedef S1* S1_PTR;`
`#pragma scl_ptr(S1_PTR, INOUT, PRIVATE)`
`void f(S1_PTR p);`
`#pragma function(f)`
	`typedef struct S1 {int *pi;} S1;`
	`typedef S1* S1_PTR;`
	`void f(S1_PTR p);`
	`#pragma function(f)`
	`#pragma scl_ptr(f.p, INOUT, PRIVATE)`

scl_ptr() is applied directly to a type name.

Conventions for Count Fields

Given a payload that contains a sized pointer (i.e., a pointer with an associated size field), it is possible that the agent setting up such a payload will assign a value to the count field that is out of range. An out-of-range value is one that is either less than 0 or greater than the prescribed maximum size (in the pragma).

There is one more special case of an out-of-range count field. It is possible that a count field resides in a block that is distinct from the block in which the sized pointer resides. Furthermore, it is possible that the pointer to the count field's block is null. Since there is no count value, this is defined to be an out-of-range condition and treated as such.

When the count field is out of range in a payload, the effect is TBD.

1.2.3.6 Restriction: Outpointers in Two-way message command payloads

The command payload of a two-way message may not have any ptr that is qualified as OUT, INOUT (and therefore RETURN and INRETURN well).

1.2.4 Pointers to Functions (Callbacks)

When a payload contains a field that is a pointer to a function, it is possible that that receiver of such a payload may make a call using the value received. Because the set of possible functions that might be passed is very large, and because all remotable methods must be identified with a SUID, it is a requirement that each such payload value be constrained to identify the specific set of functions that might be passed. Two pragmas are used for making these associations: scl_fptr_named() and scl_fptr_anonymous().

Payload fields that are of type pointer to function that are not explicitly associated with a list of candidates are treated as if they have been declared as void *.

1.2.5 Unions

Unions are C language constructs that have a set of members, of which at most one can be stored in the union object at any time. This is called the "active" member. For each union that is part of a payload, there must be a means to determine which member is active. There are two basic methods for identification of the active member:

- One union member is designated as always active. The union will be treated as if this member is permanently active, there is no discriminant, nor any way to change the active member.
- A secondary field is designated as the discriminant and its value determines which (if any) of the union members are active.

1.2.5.1 Discriminants

Unions within the source code are easily identified by the "union" keyword. Discriminants are not easy to identify, the scl_union() pragma is necessary to identify them. The scl_union_activate() pragma is optionally used to define a mapping between discriminant values and union members. The details of both scl_union() and scl_union_activate() are found in later sections.

Within a discriminated union, the discriminant value determines the active member of the union. Thus, there is a mapping between discriminant values and union members. A number of mapping choices are supported.

- In the simplest mapping, a discriminant value of n directly identifies the n^{th} union member as active. In other words, a value of 0 would indicate the first union member, a value of 1 the second, etc.
- If the discriminant is an enumerated type, then it is possible to set up an association between enumeration constants and active members such that the n^{th} enumeration constant (in declared order, not value order) maps to the n^{th} union member.
- It is also possible to create an explicit map between discriminant value sets and active members by specifying that a particular value or set of values maps to a specific member.

The following constraints are enforced for the mapping between discriminant values and union members

- A particular discriminant value may map only to a single union member.
- A particular field may act as a discriminant for more than one union
- A single union may only have a single discriminant field

- The type of the discriminant field must be an integer or enumerated type or must have been cast (using scl_cast()) to an integer or enumerated type.
- scl_values() applied to the discriminant field affects the default mapping between discriminant values and union members.

Default Mapping

As mentioned previously, the scl_union() pragma is used to identify the discriminant for a union and scl_union_activate() is used to map discriminant values to union members. If there are no scl_union_activate() pragmas for a particular discriminated union, then the mapping between the discriminant and members is said to be default. The default mapping depends on the type of the discriminant:

- If the discriminant is one of the standard integer types, or has been cast to one of the standard integer types using scl_cast(), then the value of the discriminant identifies the position of the active member. That is, a value of 0 indicates that the first union member, a value of 1 the second, etc.
- If the discriminant is an enumerated type or has been cast to an enumerated type, or has had a set of constant values prescribed using scl_values() then each constant has both a value and a position within the list. It is the position, rather than the value, that identifies the active member. When the discriminant takes on the value of the constant from position n, the n^{th} union member is active. In the case the two constants from the same list have the same value (but different positions), an error is recognized.

Explicit Mappings

A union that has at least one scl_union_activate() pragma applied to it is said to have an explicit mapping. When a union has an explicit mapping there is no default mapping, rather all mapping between discriminant values and union members is prescribed by the set of scl_union_activate() pragmas for the unions. The details of scl_union_activate() are found in later sections.

Internal and External Discriminants

A union's discriminant is either internal or external. A union has an external discriminant if the discriminant field is not contained within the union.

An internal discriminant is one that is located inside the union. If a union has an internal discriminant the following must be true for the union to behave properly when marshaled across platform boundaries.

- Every member of the union must have a field that corresponds to the internal discriminant. All such fields must be of exactly the same type or had exactly the same scl_cast() or scl_values() specifications applied. Furthermore, all such fields must be positioned in exactly the same memory location within the union. If they are located in a payload block other than the one containing the union, then the

expression “path” leading to each must be the same in the sense that all corresponding pointers across all the members have exactly the same offsets.

The SCL language translator will verify that the above conditions are met.

Conventions for “out of range” discriminant values

Given a mapping between discriminant values and union members, it is sometimes possible that certain discriminant values have no corresponding active member. In this case, the discriminant value is said to be out of range. For a union with an out of range external discriminant, the receiver of such a payload must assume that the contents of the union are undefined since there was no active member.

The receiver of a payload containing a union with an internal discriminant that is out of range must assume that the contents of the union are undefined with one exception: the discriminant, which will contain the out of range value to allow the receiver to determine that no union member is active.

There is one more special case of out of range discriminant. It is possible that a discriminant resides in a block that is distinct from the block in which the union resides. Furthermore, it is possible that the pointer to the discriminant’s block is null. Since there is no discriminant value, this is defined to be an out of range condition and treated as such.

When the discriminant is an internal discriminant, the receiver of such a payload can assume that the values of the pointers along the path leading up to the discriminant are valid, at least up to the pointer that is null. This makes it possible for the receiver to successfully test for the out of range condition caused by the absence of the block in which the discriminant resides.

1.2.5.2 Unions Containing Pointers

Union members may contain pointers. There are no additional restrictions for pointers contained within unions that have a fixed active member. However, pointers within a discriminated union must meet certain additional conditions. These conditions are related to both the direction of the block in which the union resides, and the block in which the discriminant resides. These conditions allow both user and owner to successfully interpret the memory layout of payloads. The conditions are prescribed below:

Discriminated Union residing in an IN block

Block Attribute in which Discriminant Resides	Allowable Attributes For Pointers that are Members Located within the Union Block	Explanation
IN	**IN,OUT, INOUT (*OUT and INOUT only possible if the union resides in the root block*)**	Owner cannot modify active member so all possible attributes are allowed.
OUT	**n/a**	Combination prohibited since discriminant will be treated as out of range when union is passed from user to owner.
INOUT	**IN**	OUT/INOUT members not allowed since they do not make sense if owner changes active member.
RETURN	**n/a**	Combination prohibited since discriminant will be treated as out of range when union is passed from user to owner.
INRETURN	**IN**	OUT/INOUT members not allowed since they do not make sense if owner changes active member.

Discriminated Union residing in an INOUT block

Allowable Block Attribute in which Discriminant Resides	Allowable Attributes For Pointers that are Members Located within the Union Block	Explanation
IN	**IN, OUT, INOUT, RETURN, INRETURN**	Owner cannot modify active member so all possible attributes are allowed.
OUT	**n/a**	Combination prohibited since discriminant will be treated as out of range when union is passed from user to owner.
INOUT	**INRETURN, RETURN**	IN, OUT, INOUT members not allowed since they do not make sense if owner changes active member
RETURN	**n/a**	Combination prohibited since discriminant will be treated as out of range when union is passed from user to owner.
INRETURN	**INRETURN, RETURN**	IN, OUT, INOUT members not allowed since they do not make sense if owner changes active member.

Discriminated Union residing in an OUT or RETURN block

Allowable Block Attribute in which Discriminant Resides	Allowable Attributes For Pointers that are Members Located within the Union Block	Explanation
IN	**RETURN**	Descendents of OUT or RETURN block can only be RETURN
INOUT	**RETURN**	Same as above
OUT	**RETURN**	Same as above
INRETURN	**RETURN**	Same as above
RETURN	**RETURN**	Same as above

Discriminated Union residing in an INRETURN block

Allowable Block Attribute in which Discriminant Resides	Allowable Attributes For Pointers that are Members Located within the Union Block	Explanation
IN	**IN** **RETURN,** **INRETURN**	Owner cannot modify active member so all possible attributes (consistent with descendents of INRETURN) are allowed.
OUT	**n/a**	Combination prohibited since discriminant will be treated as out of range when union is passed from user to owner.
INOUT	**RETURN** **INRETURN**	IN members not allowed since they do not make sense if owner changes active member.
RETURN		Combination prohibited since discriminant will be treated as out of range when union is passed from user to owner.
INRETURN	**RETURN** **INRETURN**	IN members not allowed since they do not make sense if owner changes active member.

If a union does not contain any pointers then there is no restriction on the type of memory block that the discriminant may reside in.

1.2.6 Strings

SCL supports the C language convention of null terminated strings via the scl_string() pragma. String support includes strings composed of the C language char type and short type (can be signed or unsigned). Strings composed of char type elements are assumed to be ASCII strings. Strings composed of short type elements are assumed to be Unicode. The scl_string() pragma identifies the field that is being characterized as a string along with a fixed maximum limit on the string length. Although a string does not have an explicit length field, a current length is always implied by the position of the null (0) value. If the number of characters (not including the null character) is exactly the limit of the length of the string, then there is no terminating null character.

In most cases, the block of memory in which a field resides is of known, fixed size. This is true for string fields that are arrays. However, this is not necessarily the case for string

fields that are pointers. For example, assume a pointer, p, of type char* has been characterized as a string with a maximum limit of 100 characters. Subsequently p is assigned to point to a series of 10 characters, and an 11th character that is null (0). Further, assume that p is part of a payload that is passed from a user to an owner. What can the owner assume about the size of the storage unit that p points to? In other words, must the owner assume that p points to exactly 11 characters? Or may the owner assume that p points to exactly 100 characters as the prescribed maximum limit? The next section outlines conventions necessary to resolve these questions.

1.2.6.1 Memory Conventions for Pointers Defined as Strings

Memory conventions for strings are similar to those for sized pointers. In fact, the conventions correspond exactly to sized pointer conventions where both the size field and the pointer have the same directional attributes.

Memory Conventions for Pointers Defined as Strings

Pointer Directional Attribute		User Obligation for Memory Size of pointed to String	Owner Assumptions for size of received data	Owner Obligation For Returned String	User Assumption For Receipt of Returned String
IN	BufferSize	Up to NULL*	Up to NULL*	n/a	n/a
	Predictable Values	Up to NULL*	Up to NULL*	n/a	n/a
OUT	BufferSize	Max	Max	Max	Max
	Predictable Values	None	None	Up to NULL*	Up to NULL*
INOUT	BufferSize	Max	Max	Max	Max
	Predictable Values	Up to NULL*	Up to NULL*	Up to NULL*	Up to NULL*
INRETURN	BufferSize	Up to NULL*	Up to NULL*	Up to NULL*	Up to NULL*
	Predictable Values	Up to NULL*	Up to NULL*	Up to NULL*	Up to NULL*
RETURN	BufferSize	n/a	n/a	Up to NULL*	Up to NULL*
	Predictable Values	n/a	n/a	Up to NULL*	Up to NULL*

Up to NULL* : The allocated buffer for the string extends only to the NULL character. In other words there is no commitment that the buffer extends beyond the NULL character nor are those characters beyond the NULL character prescribed to have any particular value. If there is no NULL character within the first Max characters, then the buffer is presumed to be exactly Max elements long and there is no trailing NULL character.

1.2.7 BitFields

Designation of fields that are also bitfields is allowed anywhere their corresponding underlying type is allowed. This means pragmas like scl_values() can be applied to bitfields; bitfields can be size fields & discriminants, etc.

1.2.8 Brew Class Objects

The scl_brew_class() pragma was invented as a convenience to Brew API developers. Brew developers simulate C++ classes and virtual functions within ANSI C by following a fixed set of conventions. The conventions are used to create classes referred to as "Brew Classes." A brew class is formed from a C structure type that is used to represent the C++ virtual function table. The structure must contain members which are of type pointer to function. Furthermore, these members must point to a function type whose first parameter is a pointer. Additional details TBD

1.2.9 Conformant Arrays

Conformant arrays are a convention used by C programmers to support variable length structures. SCL supports the notion of a structure whose last member is an array whose actual length is determined by another field within the structure called the count field. This gives the effect of a variable length C structure.

The scl_conform() pragma is used to identify a structure that is used as a conformant array. We call the structure "the conformant array structure." The array field is called the "conformant array" and it must be the last member of the conformant array structure. Furthermore, it must be declared as an array with a single dimension, typically of length 1, e.g. int x[1]. However, the declared length of the array is immaterial and will be ignored by the scl_conform pragma. The count field must also be a member of the conformant array structure.

The count field and the conformant array must be members of the same structure (namely, the conformant array structure).

1.2.9.1 Restrictions on the use of conformant array structures

The use of conformant array structures is restricted to certain contexts. This is primarily because their use is via programmer defined conventions. They have no direct support in the C language.

1. The type of a formal parameter, or return value of a function may not be a conformant array structure. (However a pointer to such is allowed).

2. A field within a structure or union may not be a conformant array structure type (Although a pointer to such a type is allowed).

3. An array element type cannot be a conformant array structure type. (Although it may be a pointer to such a type.)

4. The type pointed to by a sized pointer may not be a conformant array structure type. (Although a pointer to such type is allowed.)

1.2.9.2 Memory Conventions for Conformant Arrays

Conformant arrays are similar to sized pointers in that one field's actual size depends upon the value of a related count field. However, they are different in that the conformant array count field's possible location, relative to the array, is more limited than the size field of a sized pointer. Since the conformant array count field must be a member of the same structure (the conformant array structure), this means that the conformant array and count must always reside in the same payload block.

Because the count field of a conformant array structure determines the size of the structure (and of the conformant array) conventions must be adopted in order to clarify the user and owner memory management responsibilities. The conventions specify the required sizes of allocated blocks containing conformant arrays when passed between agents.

The table below prescribes the memory conventions for conformant arrays. The block size depends on the memory attribute that contains the conformant array structure along with either the value of the count field or the maximum declared count of the conformant array. The actual formulas used for the calculation are listed immediately after the table.

Attribute of Block containing Conformant Array Structure	Block Size Allocated by User Determined By	Owner Assumed Block Size Determined by	Owner Obligation For Returned Block Size Determined by	User Assumption For Returned Block Size Determined By
IN	$Count_{in}$	$Count_{in}$	n/a	n/a
OUT	MaxCount	MaxCount	MaxCount	MaxCount
INOUT	MaxCount	MaxCount	MaxCount	MaxCount
INRETURN	$Count_{in}$	$Count_{in}$	$Count_{out}$	$Count_{out}$
RETURN	n/a	n/a	$Count_{out}$	$Count_{out}$

When Block Size is determined by $Count_{in}$, the block size is:

```
Block Size =
   sizeof(conformant_array_structure) +
   Count_in * sizeof(conformant_array_element_type) –
   declared_length_of_array * sizeof(conformant_array_element_type)
```

When Block Size is determined by MaxCount the block size is:

```
Block Size =
   sizeof(conformant_array_structure) +
   MaxCount * sizeof(conformant_array_element_type) –
   declared_length_of_array * sizeof(conformant_array_element_type)
```

When Block Size is determined by $Count_{out}$ the block size is:

```
Block Size =
   sizeof(conformant_array_structure) +
   Count_out * sizeof(conformant_array_element_type) –
   declared_length_of_array * sizeof(conformant_array_element_type)
```

1.2.10 Treatment of Function Parameters Declared as Arrays

Given the function declaration:

```
int f(char c[20]);
```

According to C language rules, the type of c is converted from “array of char” to “pointer to char”. The length of the array (in this case 20) has no meaning.

The treatment of arrays as function parameters is in line with the ANSI C standard. The table below illustrates the treatment for several cases:

Actual Declaration	**Treated as if it were declared**
`int f(char c[20]);`	`int f(char *c);`
`int f(char c[10][20]);`	`int f(char (*c)[20]);`
`int f(char c[10][20][30]);`	`int f(char (*c)[20][30]);`

In these examples, the type of the formal parameter c is automatically converted from "array of X" to "pointer to X.". This is agreement with ANSI C language rules.

Furthermore, this allows pragmas to be applied to c as if it were a pointer – which it is, since it is automatically converted to such.

1.2.11 Pointers to Incomplete Types

A pointer to an incomplete type, that is never completed, is treated as if it were cast to "void *" using scl_cast().

1.2.12 Treatment of Unnamed Parameters in Function Prototypes

STRIDE automatically gives unnamed parameters in function declarations a name. These names are synthesized to allow the parameters to be referenced by name from within pragmas.

Unnamed parameters are named, from left to right order, with the name p<N> starting with N=1. If another parameter is already named p<N>, then p<N+1> is tried, then p<N+2>, etc. until a non-colliding name is found.

For example,

```
int f(int , int p3, float,  double p1);
```

is treated as if it were declared

```
int f(int p2 , int p3, float p4,  double p1);
```


1.2.13 Significance of SCL Pragma Source Code Location

1.2.13.1 Location of pragma relative to C source

A pragma that makes use of any C-language identifier or macro name (e.g., within an absolute specifier or constant expression) must appear after the declaration of the macro or identifier.

Macros within pragmas are expanded according to macro context of their source location. (This is important because macros are not constant, they can come and go with #define and #undef.)

1.2.13.2 Location of pragmas relative to each other

The source location of pragmas relative to each other is has no effect on the semantics of the SCL Specification. The semantics are the same regardless of order.

Order Exception For scl_cast() and scl_ptr_opaque() Pragmas

There is one exception to the rule that the source location of pragmas relative to each other has no effect on semantics. The location of scl_cast() and scl_ptr_opaque() relative to other pragmas is important.

The first rule of pragma location is that the pragmas must appear in the source code after the "C" language symbols they reference have been declared. For example, in the fragment below, the scl_ptr() pragma must appear after the declaration of "S" and member "p" because the scl_ptr() pragma references them.

```
typedef struct {int *p;) S;
#pragma scl_ptr(S, p, OUT, PRIVATE)
```

The problem with scl_cast() and scl_ptr_opaque() are that they have an effect that is equivalent to changing the C source that has been seen so far. Consider

```
typedef struct { int *p; } S1;
typedef struct {int *q; } S2;
typedef struct { S1 * pS1; } S;
#pragma scl_ptr(S, pS1->p, OUT, PRIVATE);
    // at the time of parsing this pragma, and before the next one,
    // S.pS1->p is validated as a  legal absolute specifier
    // based on the C declarations seen so far.

#pragma scl_cast(S.pS1, S2 *)
    // This pragma alters the view of the preceding C source,
    //  invalidating some absolute specifier combinations
    // and creation new valid combinations
```

In keeping with the rule that the C language symbols must be declared before use in a pragma, it follows that scl_cast() must come prior to pragmas that refer to the casted symbols.

scl_cast() may not be applied to a set of runtime values that is a superset of runtime values specified for any previous pragma (See section 1.2.16 Absolute Specifiers for a description and definition of runtime value sets.) Colloquially, this means that scl_cast() and scl_ptr_opaque() can't "cast away" information that has been prescribed by pragmas that have appeared earlier in the source code. [4]

1.2.14 SCL Translation Units

An SCL specification need not all be translated all at once. It may be broken up in to units called workspace header files that are each translated independently. A workspace header file, together with all other included source, comprises a translation unit.

Independently translated units are combined into a single compiled SCL specification. The number of units is determined by the number of workspace header files.

The order of translation units in a valid SCL specification does not affect the outcome of the final compiled SCL specification. A single unchanged translation unit can be re-compiled and re-combined into a previously compiled valid SCL specification and the result is a new specification that is equivalent to the original.

The following defines how translation units are combined:

- Assume there are two translation units, 1 and 2. The following are extracted and isolated from unit 1:
    - All the interfaces and tracepoints defined by (or synthesized for) all scl_func(), scl_function(), scl_fptr_list(), scl_ptr_flist(), scl_tp(), scl_tracepoint(), scl_cclass() and scl_brew_class() pragmas.
    - The set of all types referenced by any of the interfaces and tracepoints recursively along with any types referenced by these types. A type T is considered referenced if all the following are true:
        - T is not a function type and there exists at least one data element of type T or T* that exists within the interface, message or tracepoint payload and T is not an incomplete type within the translation unit.
        - T is a function type
    - The set of all pragmas that reference any of the defined interfaces or referenced types.

[4] An alternative way of stating this principle is if the target specifier of an scl_cast() pragma is a right subspecifier of any absolute specifier found in ay pragma which appears before it in a the source code then this is flagged as an error.

- The set of all object-like macro names and enumerated constant names.
- All other information from the translation unit is discarded. This would include unreferenced types along with any pragmas that reference them.

- The same information from unit 2 is isolated.

The isolated information from unit 1 and unit 2 is combined in the following fashion:

1. Interfaces and trace points are combined. The rules for any name and/or SUID collisions have been specified in other sections of this document.

2. Object-like macro names and enumerated constants are combined. When combined, two object like macro's with the same name must have identical values or an warning is recognized and the value of the constant in the Symbolic Constants Collection is changed to "`ERROR_VALUE_CONFLICT`". The same is true for enumerated constants with the same name.

3. If the isolated information from the two compilation units each contain a type with the same name (call them T_1 and T_2 respectively), then the following must hold:

   - T_1 and T_2 must have compatible types. Moreover, if they are structure, union or enumeration types, then they must follow the rules for compatible types defined in separate translation units (Section 6.2.7 of ISO/IEC 9899). If they do not have compatible type, an error will be raised.

   - Every pragma applied to T_1 must have a corresponding equivalent pragma applied to T_2 or an error will be raised. Two pragmas are equivalent if

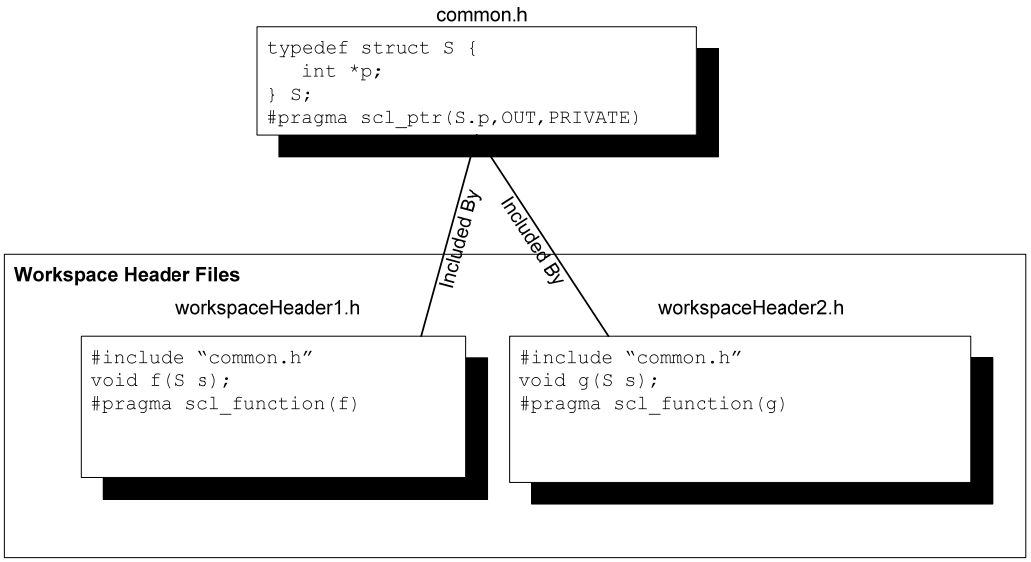
     - they are the same pragma
     - any absolute specifiers within the pragmas are the same (i.e. have the same path and designate the same field)
     - All other attributes across the two pragmas are the same.

If there are additional compilation units, then the information from the n^{th} compilation unit is combined with the result of combining all the n-1 units in the same way.

TBD: We have recently extended the language to not generate a merge error on typedef names that resolve to the same type. The formal definition of this needs to be completed.

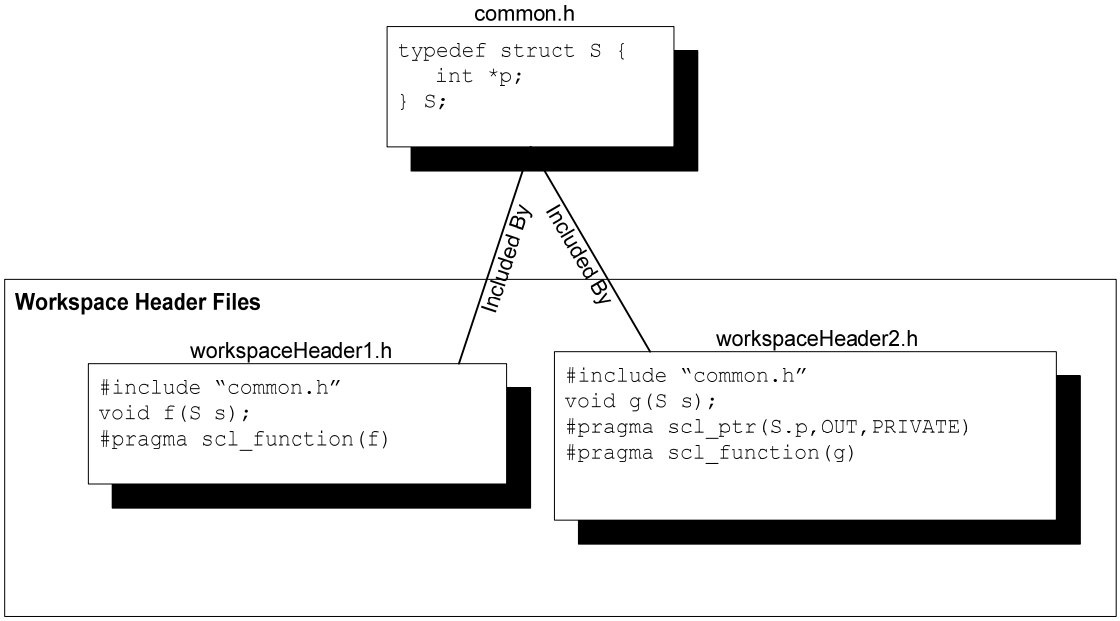
Example 1: (Valid SCL) Pragmas for a given type must be included into each translation unit

In the example below, the scl_ptr() pragma becomes a part of both translation units. This is a valid SCL Specification.



Example 2: (INVALID) Pragmas for a given type not included into each translation unit

In the example below, the scl_ptr() pragma is not a part of the workspaceHeader1 translation unit. Thus an error will be recognized when the two translation units are combined.

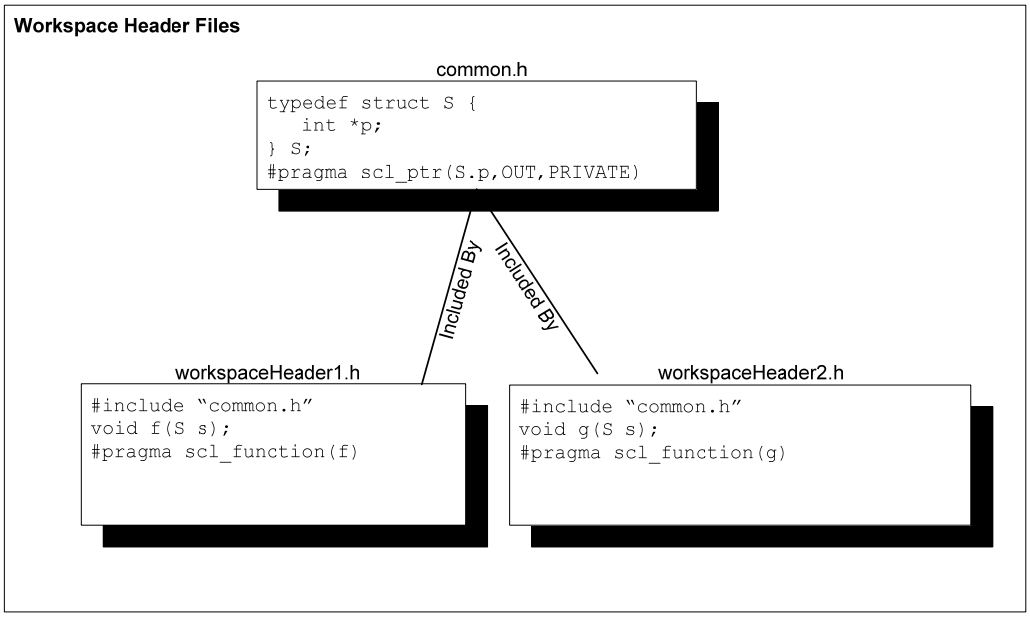


Compilation of the above workspace yields an error because the type S does not have the same pragmas applied in the two translation units.

Example 3: Some workspace header files have no effect on the output SCL Specification

The rules for combining separately translated units require that each unit have at least one interface, trace point, object-like macro or enumerated constant for it to contribute to the output SCL specification. In the example, below we have three workspace header files (workspaceHeader1.h, workspaceHeader2.h and common.h). This is a valid SCL Specification. All its translation units compile and can be successfully combined. However, the workspace header file common.h contributes nothing to the translation process because it defines no interfaces, trace points, object-like macros or enumerated constants. It can be removed from the set of workspace header files and the SCL Specification remains the same. (Note that if a change was made to common.h, then it was re-compiled, this would have no effect on the compiled SCL Specification.

When a translation unit's contents are discarded in this way, the compiler must issue a warning stating that the its contents did not contribute to the compiled SCL specification.



1.2.15 Self-Referential (Recursive) Data Structures

A C language struct or union may have members that are pointers which either directly or indirectly point to an item of the same type. Such a struct or union is said to be self-referential or recursive. Each runtime instance of a recursive set must have identical SCL attributes. The SCL Language translator will issue an error if this is not the case.

1.2.16 Absolute Specifiers

Most pragmas require a set of C language runtime values to be designated as those to which the pragma applies. The values are fields within a particular message or function payload. The values to which a pragma applies depends on the expression used to identify them. Informally, the identification can be any of the following:

- All instances declared using a particular set of typedef names.
- Instances of a field or fields within a particular function payload – either Command or Response.
- Instances of a field or fields within a particular message payload – either Command or Response.

The following syntax formally specifies the language used to identify the values. The complete expression that designates the values is always referred to as an absolute specifier.

Absolute Specifier

```
absolute-specifier :
     type-specifier
     function-return-val
     message-response-qualification
     message-command-qualification
     formal-param
     type-specifier-field

type-specifier:
    * type-specifier
    ( type-specifier )
    single-identifier

single-identifier :
    identifier


function-return-value :
    func-identifier ()
    * function-return-value
    ( function-return-value )
     function-return-value -> identifier
     function-return-value . identifier

message-response-qualification :
    ( message-designation ) ()
    message-rsp-field-designation

msg-rsp-field-designation
    ( message-designation ) () . identifier
    * message-rsp-field-designation
    ( message-rsp-field-designatio n)
     message-rsp-field-designation -> identifier
     message-rsp-field-designation . identifier

message-command-qualification :
    ( message-designation )
    message-cmd-field-designation

message-cmd-field-designation
    ( message-designation ) . identifier
    * message-cmd-field-designation
    ( message-cmd-field-designation  )
    message-cmd-field-designation -> identifier
    message-cmd-field-designation . identifier

message-designation :
    message-name
    suid-expression

integer-constant-expression
   must evaluate to a message suid

formal-param :
    func-identifier . param-name
    * formal-param
    ( formal-param )
    formal-param -> identifier
    formal-param . identifier

func-identifier :
    identifier
```



```
param-name :
    identifier

type-specifier-field :
    leftmost-specifier-name
    *  type-specifier-field
    type-specifier-field. identifier
    type-specifier-field -> identifier
    ( type-specifier-field )

leftmost-specifier-name :
    identifier
    enum identifier
    struct identifier
    union identifier
```

Constraints

- An *absolute specifier* denotes the syntax for designating a set of C-language runtime values. It is used throughout later sections of this document within many of the pragmas.
- A *single identifier* is an identifier that is a type name. Depending upon the pragma, there may be further constraints on the allowable set of types designated.
- A *func identifier* is an identifier that must resolve to a function that is identified with the scl_func() or scl_function() pragma.
- A *message name* is the name given to a message resulting from the appearance of and scl_msg() pragama. See later sections for specifics of how message names are formed.
- A *param name* is an identifier that must resolve to a formal parameter of the function corresponding to func identifier.
- A *leftmost specifier* name is the leftmost identifier in an expression that uses at least one of the *, -> or . operators and is not a function name. Unless otherwise noted, it must be a typedef name that is either a structure type, union type, or a pointer type that points to another pointer type, or points to a structure or union type. If there is no typedef name that matches the identifier, then the space of tag names is searched. If this identifier is a struct or union tag name, then the identifier designates that struct or union type.
- A *function return value* consists of a function name immediately followed by () that designates the value returned by the function.
- *suid expression* is an integer constant expression that evaluates to the numeric SUID value of a message previously identified by the appearance of an scl_msg() pragma

Semantics

An absolute specifier identifies a set of C runtime values to which a pragma applies. It is possible for two absolute specifiers (that are part of two different pragmas) to identify a set of values that overlap. Consider the following SCL:


```
typedef int* INT_PTR;
typedef struct S1 {INT_PTR *p;} S1;
typedef struct S2 {S1* ps1;} S2;
void  f(S2* ps2);
#pragma scl_function(f)
```

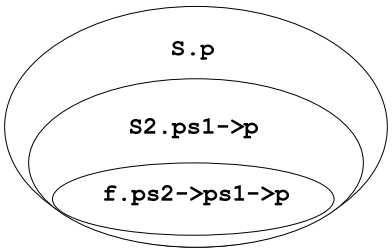
Given this example, the following are all valid absolute specifiers that can be used to identify instances of “p.”

```
S1.p
S2.ps1->p
f.ps2->ps1->p
```

Each identifies a different runtime value set:

Absolute Specifier	Corresponding runtime value set
`S.p`	All instances of the pointer p located within all instances of type S.
`S2.ps1->p`	All instances of the pointer p within all instances of type S that are pointed to by the ps1 field of all instances of S2.
`f.ps2->ps1->p`	The one and only p value located within the payload of f pointed to ps1, which is in turn pointed to by the parameter ps2.

The diagram below depicts the sets’ relationship. Notice that the sets intersect and that the intersection relationship is such that one set is always a proper subset of the other.



This is an important characteristic of absolute specifiers. When two absolute specifiers identify intersecting runtime value sets, they either identify exactly the same set, or one is a proper subset of the other.

1.2.16.1 Formation of an Absolute Specifier from Base and Relative Parts

Some forms of some pragmas allow the splitting of an absolute specifier into two parts, called the *base-specifier* and the *relative-specifier*. Although still supported, the use of base/relative form is discouraged. An absolute specifier is constructed from the base and relative parts by insertion of the entire base specifier expression immediately to the left of the first identifier in the relative specifier and placing a **.** (period) between them.

Examples of splitting an absolute specifier into base and relative parts is shown below. Under some circumstances, the split requires that the -> shorthand be normalized into the explicit ***** and **.** operations. An example of the various ways to split a given absolute specifier is shown below.

Absolute Specifier	**Base Specifier**	**Relative Specifier**
*(a.b).c->d.e->f	*(*(a.b).c->d.e)	f
	*(a.b).c->d	e->f
	((a.b).c)	d.e->f
	*(a.b)	c->d.e->f
	a	*(b).c->d.e->f

1.2.16.2 Designation of sized pointer elements in absolute specifiers

The absolute-specifier syntax deliberately limits the designation of indexed elements (either by array or pointer subscripting) so that only the first element of such a series can be designated. In this way, pragmas can be applied only to the first element of a series. Application of a pragma to the first element is generalized to mean application of the pragma to every element of the series in an identical manner.

When a pointer, p, is identified as a sized pointer (using scl_ptr_sized()), and another pragma is applied to an entity that p points to through the use of p in the absolute specifier, then the pragma application applies to all elements to which p points (since p is a sized pointer, it points to a series of elements).

For example, in the SCL below, the pointer, p, is a sized pointer. When p is used within an absolute specifier to designate an element pointed to, the pragma applies to all the elements in the series pointed to by p, not just the first.

```
typedef struct S {int *pInt; int y;} S;
int f(int size, S* p);
#pragma scl_function(f)
#pragma scl_ptr_sized(f.p,IN, PRIVATE, f.size, 10)
#pragma scl_ptr(f.p->pInt, IN, POOL)
    // this pragma applies to all instances of pInt
    // pointed to by f.p, not just the first
```

1.2.16.3 Designation of Array elements in absolute specifiers

The absolute specifier notation does not allow the designation of array elements using **[** and **]**. In fact, the notation deliberately prohibits the designation of the n^{th} array element, but does allow the first element of an array series to be identified. Application of a pragma to the first element means application of the pragma to every element of the array in an identical manner.

SCL allows this by leveraging the notion of equivalence of pointers and arrays. In C, every expression of the form:


```
a[i]
```

can be replaced with its equivalent “pointer” form:

```
*(a+i)
```

Using this method, absolute specifiers can be used to generically designate the first array element and items within it without using **[** and **]**. Consider the example below. It applies the `INOUT` directional attribute to every `p` instance in the array contained by S for the interface `f`.

```
typedef struct S { int *p } S;
typedef struct S2 { S sArray[10]; } S2;
void f(S2 s2);
#pragma scl_function(f)

#pragma scl_ptr_sized(f.s2.sArray->p, INOUT, PRIVATE, 23);
    // applies pragma to all 10 instances of p
```

1.2.16.4 Qualification of Message Payload Examples

Message payloads may be qualified on a per-message basis by designation of the applicable message using either the message name or a numeric expression for the message SUID. See 1.2.16 Absolute Specifiers for the formal definition. This section simply clarifies the formal definition through use of examples. .

Items within message payload may be designated for qualification using two basic methods. The primary difference is how the message is identified. All messages have a name and the name may be used to associated the qualification back to the scl_msg() statement that defined the message. In addition, all messages have a SUID (a 31 bit integer) that uniquely identifies the message. This number may also be used to provide the association between the scl_msg() pragma and this qualification.

Examples of Qualifying Scalar Message Payloads

```
// command example scalar payload
#define Cmd 1
#pragma scl_msg(Cmd, int)
enum E {Red, Green, Blue};
#pragma scl_values(Cmd, Red, Green, Blue);
```

or

```
//command example scalar payload
#define Cmd 1
#pragma scl_msg(Cmd, int)
#pragma scl_cast(Cmd, unsigned int)
```

or

```
// response example scalar payload
enum E {Red, Green, Blue};
#define Rsp srMT_ONE_RSP | 1
#pragma scl_msg(Rsp, int)
#pragma scl_values( (Rsp)(), Red, Green, Blue)
```


or

```
// response example sclalar payload with enum
enum E {Red, Green, Blue};
enum Messages{ Rsp =  srMT_ONE_RSP | 1};
#pragma scl_msg(Rsp, int)
#pragma scl_values( (Rsp)(), Red, Green, Blue)
```

or

```
// qualification using synthsized name
enum E {Red, Green, Blue};
#pragma scl_msg( 1 + 5, int)
#pragma scl_values( (Msg0x6), Red, Green, Blue)
```

or

```
// qualification using numeric expression
enum E {Red, Green, Blue};
#define Cmd 1 + 5
#pragma scl_msg( Cmd, int)
#pragma scl_values( (2 + 4), Red, Green, Blue)
```

or

```
// two way msg qualification
Enum E {Red, Green, Blue};
#define TwoWay srMT_TWO | 1
#pragma scl_msg(TwoWay, int, long)
#pragma scl_values((TwoWay), Green, Blue)
#pragma scl_values((TwoWay)(),  Red)
```

Examples of Qualifying Non-Scalar Message Payloads

If a message payload is not a scalar type it must be a structure or union type. Pointer types are not allowed. Note that the structure or union payload may contain a pointer type, but the type representing the root block of the payload may not be a pointer type. Again the message name or SUID numeric expression must always have parenthesis around it.

```
// command msg non-scalar payload
struct S { int *p, int i};
#define Cmd 1
#pragma scl_msg(Cmd, struct S)
#pragma scl_ptr((Cmd).p, IN, PRIVATE);
```

or

```
// enum two way command using common type
struct S { int *p, int i};
enum E {TwoWay = srMT_TWO | 1};
#pragma scl_msg(TwoWay, struct S, struct S)
#pragma scl_ptr(TwoWay.p, IN, PRIVATE);
#pragma scl_ptr(TwoWay().p, RETURN, PRIVATE);
```


1.2.17 Interaction between Pragmas

By virtue of the absolute specifier used, pragmas may identify runtime value sets that overlap. This section prescribes semantics and handling of these cases.

The SCL pragmas use a variety of ways to identify the objects to which the pragma applies. Some pragma's identify the set applied to using a constant expression, some a function identifier, some an absolute specifier. In addition, each pragma typically further restricts the object set by disallowing (or allowing) only certain types, value ranges, etc. In this section we categorize the value sets that pragmas can be applied to. Use of these value sets will enable the identification of pragmas that have the potential to identify value sets that may overlap. The table below defines the sets and gives them a shorthand name.

Set Name Shorthand	**Meaning**
Ptr	A subset of the set of all runtime instances of type pointer, excepting pointer to function and void * and pointer to other incomplete type that is never completed.
Ary	A subset of the set of all runtime instances of type array.
U	A subset of the set of all runtime instances of type union.
S	A subset of the set of all runtime instances that are of type struct.
FID	A single identifier from the set of all function identifiers.
CE	A constant expression.
MID	***CE*** whose value has been promoted the status of ***MID*** by an scl() pragma.
STPID	A ***CE*** whose value has been promoted to the status of ***STPID*** by an scl_tracepoint() pragma
pF	A subset of the set of all runtime instances that are of type ptr to function
Int	A subset of the set of all runtime instances of one of the standard integer types AND the set of all runtime instance that scl_ptr_opaque has been applied to.
Count	The set of all runtime instances of Count fields identified by scl_ptr_sized pragmas.
Disc	The set of all runtime instance of discriminants identified by scl_union pragmas
UMember	The set of all runtime instances that have been identified as an activated member via a scl_union_activate pragma

We now identify the set that each pragma may affect and determine which pragmas can have overlapping value sets.

Pragma	Sets It May Affect	May Overlap With	Remarks
scl_func	***FID***	scl_function	
scl_function	***FID***	scl_func	
scl_values	***Int***	scl_cast	
scl_cast	***Ptr , Int, pF***	scl_ptr scl_ptr_sized scl_string scl_ptr_opaque	scl_cast can be applied to any pointer type along with integral types.
scl_ptr	***Ptr***	scl_ptr_opaque scl_ptr_sized scl_string scl_cast	
scl_ptr_opaque	***Ptr***	scl_ptr scl_ptr_sized scl_string scl_cast	
scl_ptr_sized	***Ptr***	scl_ptr scl_ptr_opaque scl_string scl_cast	
scl_string	***Ptr***	scl_ptr scl_ptr_opaque scl_ptr_sized scl_cast	
scl_string	***Ary***	scl_conform	
scl_union	***U***	scl_union_activat e	
scl_union_activate	***U***	scl_union	
scl_fptr_list	***pF***	scl_ptr_flist scl_cast scl_brew_class scl_cclass	
scl_ptr_flist	***pF***	scl_fptr_list scl_cast scl_brew_class scl_cclass	
scl_tracepoint	***CE***		
scl_tracepoint_forma t	***STPID***		
scl_cclass	***S, pF***	scl_fptr_list	The absolute specifier for an

Pragma	Sets It May Affect	May Overlap With	Remarks
		scl_ptr_flist scl_conform, scl_brew_class scl_cast	scl_cclass pragma identifies a struct, but an scl_cclass pragma also adds attributes to member of that struct that are of type ptr to function. In this way, scl_cclass applies to both ***S*** and ***pF*** at the same time.
scl_brew_class	***S, pF***	scl_fptr_list scl_ptr_flist scl_conform, scl_cclass, scl_cast scl_ptr_opaque	Applies to both ***S*** and ***pF*** in a way similar to scl_cclass.
scl_conform	***S, Ary***	scl_string, scl_brew_class, scl_cclass,	The absolute specifier for an scl_conform pragma identifies as struct, but also affects the last member of that struct which must be an array. In this way, scl_conform applies to both ***S*** and ***Ary*** at the same time.

1.2.17.1 Interaction between different pragmas

This section addresses the interaction between different pragmas. Two pragma instances are considered to be different pragmas if the pragma names are different. There are two basic cases of interaction between different pragmas:

1. The pragmas can be applied to the same value set.
2. The pragmas can be applied to overlapping, but not the same, value sets.

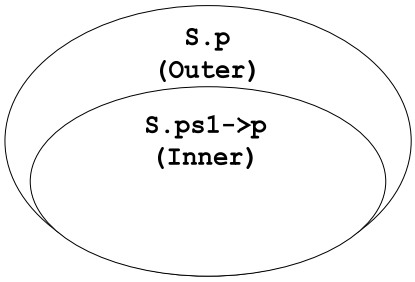
Different Pragmas applied to the same value set

It is an error if two different pragmas are applied to the same value set with the following exceptions:

- scl_ptr() and scl_string() may be applied to the same value set and the effect is additive.
- scl_union_activate() may be applied to the same value set as other pragmas and the effect is additive.
- Other pragmas may be applied to the same value set as scl_cast(), with the exception of scl_ptr_opaque(), providing that the other pragma appears after scl_cast() in the source. It is an error if scl_ptr_opaque() is applied to the same value set as scl_cast() when appearing after the cast in the source.
- Related scl_union/scl_union_activate pragmas must identify exactly the same runtime value set for the union. Otherwise they are not related.
- scl_values and scl_ptr_opaque may bed applied to the same value set and the effect is additive.
- It is an error if scl_func and scl_function are applied to the same ***FID***.

Different Pragmas applied to overlapping value sets

When two different pragmas are applied to overlapping value sets, it is always the case that one of the value sets is a proper subset of the other. This is depicted below. We call the larger, less specific set the “Outer” set and the smaller, more specific set the “Inner.” We will refer to the outer as the less-specific specifier while the inner will be the more-specific specifier.



The table below defines the semantics of the interaction between all pragmas that may be applied to overlapping value sets.

Outer	**Inner**	**Semantics**
scl_ptr_opaque	<any except scl_values)	Error
scl_ptr_opaque	scl_values	Inner is additive with outer
<any except scl_ptr_opaque>	scl_cast	Inner overrides outer
scl_cast	scl_ptr_opaque	Error
scl_cast	<any scl_ptr_opaque>	scl_cast must have appeared first in the source. The other pragma refers to the transformed program structure in place after scl_cast has been applied.
scl_ptr, scl_ptr_sized	scl_ptr_opaque	Inner overrides outer
scl_ptr_opaque	*scl_ptr, scl_ptr_sized, scl_string*	*Error. works same as cast to void*
scl_ptr_sized	scl_ptr	Inner overrides outer
scl_ptr	scl_ptr_sized	Inner overrides outer
scl_ptr	scl_string	Inner is additive with outer
scl_string	scl_ptr	Inner is additive with outer
scl_ptr_sized	scl_string	Error
scl_string	scl_ptr_sized	Error
scl_string	scl_ptr_opaque	Inner overrides outer
scl_conform	scl_string	Error
scl_string	scl_conform	Error
<any except scl_cast or scl_union>	scl_union_activate	Inner is additive with outer
scl_union_activate	<any except scl_cast or scl_union>	Inner is additive with outer
scl_fptr_list/ptr_flist	scl_fptr_list/ptr_flist	Inner overrides outer
scl_brew_class/cclass	scl_fptr_list/ptr_flist	Inner overrides outer
scl_fptr_list/ptr_flist	scl_brew_class/cclass	Inner overrides outer
scl_conform	scl_brew_class/cclass	Error
scl_brew_class/cclass	scl_conform	Error
scl_values	<d	Inner is additive with outer

1.2.17.2 Same pragma applied to overlapping value sets

This section addresses the interaction between same pragma overlapping value sets. Two pragma instances are considered to be the same pragma if the pragma name is the same. The parameter values need not be the same.

The table below details the semantics for handling the same pragma applied to the same or overlapping value sets. As the table shows, it is always an error for the same pragma to be applied to the same value set. And, with the exception of cast, in the case of pragmas whose value sets overlap, but are not the same, the inner (more specific) always overrides the outer (less specific).

Pragma	When Applied To Same Value Set	When applied to overlapping value Sets
scl_func	Error	n/a
scl_function	Error	n/a
scl_msg	Error	n/a
scl_values	Error	Inner overrides outer
scl_cast	Error	Inner overrides outer
scl_ptr	Error	Inner overrides outer
scl_ptr_opaque	Error	Inner overrides outer
scl_ptr_opaque	Error	Inner overrides outer
scl_ptr_sized	Error	Inner overrides outer
scl_string	Error	Inner overrides outer
scl_string	Error	Inner overrides outer
scl_union	Error	Inner overrides outer
scl_union_activate	Error	Inner overrides outer
scl_fptr_list	Error	Inner overrides outer
scl_ptr_flist	Error	Inner overrides outer
scl_tracepoint	Error	n/a
scl_tracepoint_format	Error	n/a
scl_cclass	Error	Inner overrides outer
scl_brew_class	Error	Inner overrides outer
scl_conform	Error	Inner overrides outer

Examples

It is an error if scl_ptr() is applied twice to the same value set, as shown in the following example:

```
typedef int* INT_PTR;
int f(INT_PTR p);
#pragma scl_function(f)
#pragma scl_ptr(f.p, OUT, PRIVATE)
#pragma scl_ptr(f.p, IN, PRIVATE)
    // Application of same pragma to the same value set twice.
    // This is an error even if the second pragma was exactly the
    // same as the first.
```

The same pragma can sometimes be applied using value sets that overlap, as shown in the following example:

```
typedef int* INT_PTR;
int f(INT_PTR p);
#pragma scl_function(f)
#pragma scl_ptr(INT_PTR, OUT, PRIVATE)
#pragma scl_ptr(f.p, IN, PRIVATE)
    // Application of same pragma to overlapping value sets.
    // This is OK and the pragma containing the more specific
    // absolute specifier will take effect within f.
```


1.2.17.3 Pragma Derivations

When a pragma with an outer (less-specific) specifier is used, its inner (more-specific) specifiers derive the same pragma. This interaction typically leads to confusing errors for the user. So this section will describe the overall pragma derivation scheme in how it relates to pragmas of different types.

General Pragma Derivations

This section describes the typical derivation process. Most SCL pragmas adhere to the following processing:

```
typedef struct { int* p; } S1;
typedef struct { S1* pS; } S2;
typedef S1 S3;
#pragma scl_ptr(S1.p, "OUT", "PRIVATE")
```

For the above snippet of code, S1.p was used in a pragma with an "OUT" direction. This resulted in two other pragmas being derived:

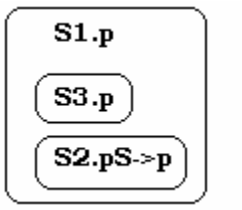
```
#pragma scl_ptr(S2.pS->p, "OUT", "PRIVATE") // derived (indirectly)
#pragma scl_ptr(S3.p, "OUT", PRIVATE")      // derived (directly)
```

These pragmas were derived because they are of the same type as the generalized pragma. However, there is nothing to prevent the user from actually declaring his own pragmas instead of these derived pragmas. In a sense, the user could create pragmas as:

```
#pragma scl_ptr(S2.pS->p, "INOUT", "PRIVATE")
#pragma scl_ptr(S3.p, "IN", PRIVATE")
```

We say that the generalized pragma of S1.p is a less-specific pragma in relation to the other two. The other two pragmas (S2.pS->p and S3.p) are more specific because they identify smaller sets of pointers.

If we think of this in terms of sets, we have the following:



All the S3.p pointers are S1.p pointers. All the S2.pS->p pointers are S1.p pointers. Applying a pragma to S1.p automatically applies to the other two unless they are overridden with a more-specific pragma specifier.

In general, a user may use a pragma with a less-specific specifier followed by a pragma with a more-specific specifier or visa-versa. This rule applies only to pragmas of the same type. Exceptions to this rule are scl_ptr_opaque and scl_cast.

Pragma Derivation Exceptions

The scl_cast pragma and scl_ptr_opaque pragma operate differently than the other pragmas. If these pragmas are used, pragmas with more-specific specifiers must be declared prior to pragmas with less-specific specifiers.

scl_ptr_opaque Derivations

This pragma is functionality the same as scl_cast(p, void*). It doesn't make sense to have two scl_ptr_opaque pragmas with a more-specific specifier and less-specific specifier because the pointer of the more-specific is no longer reachable.

Problems occur when a different pragma type uses a less-specific specifier and the scl_ptr_opaque is used with a more-specific specifier. This causes problems because the scl_ptr_opaque may not remove a prior derived pragma of a different pragma type.

```
typedef struct { char* pChar; } S4;
typedef struct { S4* pS4 } S5;
typedef struct { S5* pS5 } S6;
#pragma scl_string(S4.pChar, 10)
// #pragma scl_string(S5.pS4->pChar, 10)          // derived
// #pragma scl_string(S6.pS5->pS4->pChar, 10)     // derived
#pragma scl_ptr_opaque(S5.pS4) // generates an error
```

The error is generated because while a more-specific pragma may override a less-specific pragma, it may only do so for a pragma of the same kind. Currently, the only exception for this rule is scl_ptr/scl_ptr_sized (this are functionally the same except one carries a size). For this example, the scl_ptr_opaque cannot remove nor castaway the derived scl_string pragma for S5.pS4->pChar.

Notice in this example that if the scl_ptr_opaque were allowed to removed the derived pragma for S5.pS4->pChar that it would also have to remove the derived pragma with the S6->pS5->pS4->pChar specifier.

This restriction prevents compilation from constantly undoing derived processing. In complex programs with many pragmas it is not only difficult for the compiler, it is impossible for the user to determine what actually happened. Due to this restriction, at the insertion of each pragma the program specifiers are validated against all prior specifiers. Thus, we know that at any point in time the integrity of pragma compilation.

If we had wanted the prior program to work, the scl_ptr_opaque could have been declared prior to the other pragmas. This would have prevented the other pragmas from being derived.

```
typedef struct { char* pChar; } S4;
typedef struct { S4* pS4 } S5;
typedef struct { S5* pS5 } S6;
#pragma scl_ptr_opaque(S5.pS4)
#pragma scl_string(S4.pChar, 10)
```

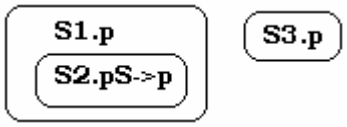
scl_cast Derivations

Cast Pragma Derivations are more troublesome than scl_ptr_opaque. An scl_ptr_opaque(p) pragma is functionally the same as scl_cast(p,void*). However, scl_cast is far more complex. This is because when one type is casted to another, it automatically derives all pragmas associated with the new type.

```
typedef struct { int* p; } S7;
typedef struct { int nCastMe; } S8;
#pragma scl_ptr(S7.p, "OUT", "PRIVATE")
#pragma scl_cast(S8.nCastMe, S7*)
// #pragma scl_ptr(S8.nCastMe->p, "OUT", "PRIVATE") // derived
```

In most programs containing a moderate number of pragmas a cast will typically result in some pragmas being derived from the cast. This presents problems in using scl_cast. Using scl_cast requires the more-specific specifiers be casted prior to the less-specific specifiers. A less-specific specifier casted to another type is "removed" from the set of pointers it resides in. Once casted, it is no longer a member of its prior "Outer" less-specific set. Thus for our earlier example, the following pragmas would result in:

```
#pragma scl_cast(S3.p, S9*)
#pragma scl_ptr(S1.p, "OUT", "PRIVATE")
//#pragma scl_ptr(S2.pS->p, "OUT", "PRIVATE") // derived
```



Notice that S3.p did not derive an scl_ptr pragma.

This is inheritance rule is also holds true when scl_cast is casts from more-specific to less-specific such as:

```
typedef struct { int* p; } S1;
typedef struct { S1* pS; } S2;
typedef S1 S3;
typedef struct { int* p9; } S9;
#pragma scl_ptr(S9->p9,"OUT","PRIVATE")
#pragma scl_cast(S3.p, S9*) // more-specific
//#pragma scl_ptr(S3.p->p9, "OUT", "PRIVATE")    // derived

#pragma scl_cast(S1.p, S9*) // less-specific
//#pragma scl_cast(S2.pS->p, S9*)                // derived
//#pragma scl_ptr(S2.pS->p->p9, "OUT","PRIVATE") // derived
```


Notice that no pragma was in conflict with the derived pragmas. However, the reverse (less-specific to more-specific casting) is not valid and will result in an error:

```
typedef struct { int* p; } S1;
typedef struct { S1* pS; } S2;
typedef S1 S3;
typedef struct { int* p9; } S9;
typedef struct { int* p10; } S10;
#pragma scl_ptr(S10->p10, ”INOUT”, ”PRIVATE”)
#pragma scl_ptr(S9->p,”OUT”,”PRIVATE”)
#pragma scl_cast(S1.p, S10*) // less-specific
//#pragma scl_cast(S3.p, S10*)                    // derived
//#pragma scl_ptr(S1.p->p10, ”INOUT”, ”PRIVATE”) // derived
//#pragma scl_ptr(S3.p->p10, ”INOUT”, ”PRIVATE”) // derived
#pragma scl_cast(S3.p, S9*) // more-specific generates error
```

By the time the user declared the second scl_cast for S3.p, the S3.p specifier was already casted AND it had already acquired derived pragmas associated with the casting. Thus, the user’s scl_cast of S3.p could not remove the derived pragma of a different type (in this case the scl_ptr(S3.p->p10,…)).

1.2.18 Macro Replacement Within Pragmas

Unless otherwise noted, macro replacement is always performed within SCL pragmas.

1.2.19 Target and Host Settings

The mapping of C-language datatypes into memory layouts is directly affected by the target settings of the SCL compiler. An SCL compiler must be configured to match the commercial compiler in use.

1.2.20 Declaration Before Use in Pragmas

Absolute specifier expressions and constant expressions within pragmas can only contain symbols (i.e., identifiers, macro names, etc) that have already been declared.

1.2.21 Trace Points

SCL supports the definition of trace points that can be activated from the target by specific calls to runtime APIs. Trace points are defined with scl_tracepoint(). Trace points have an optional payload. The formatting of the payload data may be described with scl_tracepoint_format(). Each trace point has an STPID. The STPID consists of a unique 16-bit ID. The STPID value of zero (0) is reserved; thus 65,535 unique trace points are available. There are no constraints on different application threads using the same trace point. The optional trace point payload can be used to associate application data with a trace point to be routed to and displayed by the host. Trace point payloads have several additional restrictions:

- they may not contain unions
- they may not contain pointers
- they may not contain conformant arrays
- TBD: What about void*? What if all their ptrs were prag'd with scl_ptr_opaque?...

The STPID of a trace point and the SUID of a SMID occupy different numeric spaces (i.e., an STPID and SUID can have the same value).

1.2.22 AutoSUID Generation

The pragmas scl_function(), scl_brew_class(), and scl_ptr_flist() allow SUID management to be performed within SCL.

1.2.23 Symbolic Constants Table

The symbolic constants collection (accessible via iScript.Constants) is a table that is created as part of a compilation and contains entries for macro names present in the source. Its primary purpose is to allow scripts to use the symbolic names for various constants just as the C source code does.

The symbolic constants table contains <name,value> pair entries that map symbolic names to constant values. An entry is created for each of the following:

- Every enumeration constant. The constant name and integral constant value together comprise the <name,value> pair.
- Every non-parameterized macro present in the source.
  - The name of the <name,value> pair is the macro name.
  - The value of the <name,value> pair can be either of the following:
    - The macro text portion of the #define, exactly as is, or
    - The number, if the SCL processor can resolve the text portion to a number. The text portion is resolvable if:
      - The value is a constant expression whose identifiers are either:
        - simple macros that evaluate to a number
        - parameterized macros that evaluate to a number
        - enumerated constant names.
      - The value can be represented by a signed 32 bit integer.

The table is constructed from the macro state in place at the end of the compilation unit. This is important because a single macro name can be assigned any number of different

values during the course of a compilation (via the #define and #undef pre-processing statements).

It is possible that when compilation units are combined, either as a result of a single-file compilation or the result of a complete re-compilation, that a symbol constant name that is present in two or more compilation units will have a different value. This situation is detected during the compilation and a warning is issued. Furthermore, because it is not possible to know which of the different values is the preferred value, the constant will be assigned the string value: “ERROR_VALUE_CONFLICT" within the symbolic constants table.

1.2.24 Typedef Propagation of SCL Attributes

If a pragma is applied to a set of runtime values designated by an absolute specifier whose leftmost identifier is type T, then that pragma also applies to any other types that are defined in terms of T. A typename T2 is defined in terms of T, if:

- T is used in the declaration specifiers of the typedef statement that defined T2
  -or-
- T2 is defined in terms of some other type, T3, that is defined in terms of T.

Example	**Explanation**
`typedef struct _S {int x;} S;`	S is defined in terms of struct _S
`typedef struct _S {int x;} S, *PS;`	S and PS are defined in terms of struct _S.

`typedef int* INT_P;`
`typedef INT_P* INT_PP;`
`typedef INT_PP *  INT_PPP;`
	INT_PP: is defined in terms of INT_P.
	
	INT_PPP is defined in terms of both INT_P and INT_PP.
	
	Any pragmas applied to INT_P will also apply to INT_PPP if not overridden by pragmas applied to INT_PP.

`typedef Struct _S {int x; } S;` `typedef S S2;`	S is defined in terms of struct _S. S2 is defined in terms of both S and struct _S;

1.2.25 Treatment of tag names

In ANSI C mode, the compiler should search the “ordinary” symbols to resolve a symbol (if there was no struct/union/or enum keyword immediately preceding it). Then it should search the tag name symbols. In C++ mode (a future option) tag names and ordinary identifiers are in the same namespace so this becomes a moot issue.

Implications:

Example 1: struct keyword is optional. Search will resolve if there is no collision.

```
struct S { int *p;};
scl_ptr(S, p, OUT, PRIVATE);      // no struct keyword necessary
```

Example 2: Keyword can be explicit

```
struct S { int *p;};
scl_ptr(struct S, p, OUT, PRIVATE);
```

Example 3: Ordinary namespace searched first

```
struct S { int *p;};
typedef struct {int x; int *p;} S;
scl_ptr(S, p, OUT, PRIVATE);     // applies to unnamed struct with
typedef name S
```

Example 4: Namespace search can be controlled by the keyword

```
struct S { int *p;};
typedef struct {int x; int *p;} S;
scl_ptr(struct S, p, OUT, PRIVATE);     // applies to struct S
```

Example 5: A minor drawback of this approach

The following is valid SCL. It compiles successfully and has the expected meaning:

```
struct S { int *p;};
scl_ptr(S, p, OUT, PRIVATE);      // no struct keyword necessary
```

If I make the following change this will no longer be considered valid SCL:

```
struct S { int *p;};
typedef struct {int x; } S;
scl_ptr(S, p, OUT, PRIVATE);   // syntax error: typedef name S has
no p
```


This syntax error can be resolved by introduction of the struct keyword:

```
struct S { int *p;};
typedef struct {int x; } S;
scl_ptr(struct S, p, OUT, PRIVATE);     // OK, talking about struct
S now
```

Note: This same treatment applies to all tag names (i.e., enum and union as well as struct).

1.2.26 Const objects

A data object that is const qualified may only reside in an IN, INOUT, RETURN or INRETURN block. A const qualified data object that resides in an OUT block is diagnosed as an error.

Any ptr that is const qualified must abide by an additional restriction: If it resides in an INOUT block it cannot be RETURN or INRETURN qualified.

1.2.27 Application of pragmas to parameters via typedef name for function type prohibited

Application of pragmas to parameters using typedef names, rather than concrete function names, is not allowed. For example:

```
typedef int (F) (int *p, int size);     // introduces typedef
                                        // name "F" for a function
                                        // type with 2 params

// ERROR!!!!! The pragma below is an error because F is a typedef
// name, not an actual function name

#pragma scl_ptr(F.p, IN, PRIVATE);
```

This restriction is a corollary of the Absolute Specifiers section beginning on page 38, since the leftmost identifier in an absolute specifier for a function must be a function identifier and cannot be a typedef name.

1.2.28 Application of pragma's to pointers to incomplete types

An error will result if a pragma is applied to a pointer field or pointer type, and the type pointed to is incomplete.

1.2.29 Unnamed fields of type struct or union

In the ANSI C language (but not C++) it is possible to have a field within a structure or union be unnamed. The following source fragment illustrates a structure with three such unnamed fields.


```
typedef struct S{
    union u{
        int i;
        char *c;
        float f;
    };
    union u2 {
        double d;
        int k;
    };
    struct s1 {
        int happy;
        int day;
    };
}S;
```

The challenge presented by such a construct how to identify the subfields of the unknown field. Normally subfields are identified using the name of the parent field. But in this case there is no such name and there are three unnamed members of S.

For the purposes of SCL and all subsequent specification, the SCL language translator will treat all such unnamed fields as if they had a name. The synthesized name will be "name<n>" where n is a digit or series of digits denoting the position of the unnamed field relative to the others. If a collision occurs between the synthesized name and any other existing name, the next n will be chosen until the collision is resolved.

This implies the above source fragment will be treated by the SCL language translator ***as if*** it were written below:

```
typedef struct S{
    union u{
        int i;
        char *c;
        float f;
    }unnamed1;
    union u2 {
        double d;
        int k;
    }unnamed2;
    struct s1 {
        int happy;
        int day;
    }unnamed3;
}S;
```

1.2.30 Naming of Unnamed Bitfields

Unnamed bit fields are given names by the SCL language translator following the same rules as unnamed fields of type struct or union. That is, synthesized name is "name<n>" where n is a digit or series of digits denoting the position of the unnamed field relative to the others. If a collision occurs between the synthesized name and any other existing name, the next n will be chosen until the collision is resolved.

1.2.31 Treatment of Zero-Length Array as Member of a Structure

An optionally allowed extension to ANSI C, SCL allows any member of a structure to be an array declared with 0 length. This feature is most useful for the last member of a struct. For the purpose of the following discussion, assume what have such a case:

```
struct S {
   char i;
   ... // all else omitted
   T ary[0];
};
```

For brevity we omit **`struct`** in the discussion below when referring to the type **`struct S`**. The declaration of the zero length array affects S in the following ways:

- The sizeof S will be adjusted upward if necessary so that it ends on a boundary suitable for the offset of the first element of ary, had one existed.
- The alignment requirements of S will be adjusted upward as necessary as if S had at least one element in ary.

1.2.32 Parameter Names in SCL Absolute Specifiers

A set of compatible declarations for a function f(), need not be consistent in parameter names (although the types must be consistent). The first parameter may have a different name in each declaration or may not even be declared. It is even possible that the first parameter in one declaration uses the same name as a different parameter, say parameter 2, in another declaration. For example the following are all compatible declarations for f()

```
int f(int x, int y, int z);
int f(int , int, int);
int f(int z, int y, int x);
```

Since parameters are designated by name in SCL absolute specifiers, this leads to an ambiguity. For the above declarations, x, can designate either the first or third parameter, depending upon whether we are considering the first or third declaration. The solution for SCL is that within an absolute specifier, the first declaration encountered permanently names the parameters, regardless of how many declarations exist. So for this example then, x is parameter 1 of f(), y is parameter 1 of f() and z is parameter 3 of f().

1.2.33 Additional restrictions for cast (Section TBD)

Entire section TBD. This is a marker for an area of SCL that is not very well defined and needs to be developed. Intuitively, this section identifies additional restrictions on the interaction between scl_cast() and other pragmas.

The following SCL should be rejected because the scl_cast() pragma invalidates the union/discriminant relationship that has previously been set up.

```
typedef struct
{
  int * pDisc;
  union { int x; } u;
} S;

#pragma scl_union(S.u, S.pDisc.[0])

void f1(S parm1);
#pragma scl_function(f1)
#pragma scl_cast(f1.parm1.pDisc, void *)
```

Reasoning:

f1.parm1.u will adopt the attributes of any pramgas applied to S.u via right subspecifier inheritance. It would be as if a user had entered the following::

```
scl_union(f1.parm1.u, *f1.parm1.pDisc)
```

However, in the context of f1.parm1.pDisc the type of pDisc is no longer int *, but has been cast to void*, making the type unsuitable to become a discriminant.

1.2.34 Test Classes

A test class is a C++ class developed with the specific intent of Stride testing. It is not part the application logic to be tested, rather it is developed with the intent of driving the application logic to be tested. A test class has the following characteristics:

- A test class is bound with the executable to be tested. It is part of the same application binary. In this way, test classes are usually part of test product builds rather than the final release product build.
- It is designed to separate test logic from the framework necessary to drive the test and record the results. A test class implements test logic. The Stride framework takes care of calling the test and managing results.
- Test class member functions contain test logic and each is treated as a test case.
- Fixtures (both setup and teardown) are supported. A setup fixture is code that is run immediately prior to every test case. A teardown fixture is code that is run immediately after ever test case.
- Test class initialization and deinitialization, is supported via the class constructor and destructor.
- Stride provide a set of pragmas to identify and elaborate test classes.

1.2.34.1 Test Class Details

A test class must be a public class. It may reside in any namespace. It cannot be nested within another class. I

1.2.34.2 Test Method Details

A test method must have the following characteristics

- It must have no parameters.
- It must be declared by they test class (it cannot be implicitly declared via inheritance from a base class).
- It must return void, bool or integral type (signed or unsigned version of long, int, short or char)

Test methods that return bool indicated test status by the return value. True for test success, false for failure.

Test methods that return an integal type indicate test status by the return value. Zero for success, non-zero for failure.

Test methods that return void must utilize the Stride runtime API's to indicate test status. If they do not use the API's to report status their status will always default to "in-progress."

1.2.34.3 Fixtures

In addition to test methods, a test class may have a setup and or teardown fixture. A setup or teardown fixture is identified as such using the scl_test_setup() and scl_test_teardown() pragmas. A setup fixture is a class method that is called immediately prior to every test method. A teardown fixture is a class method that is called immmediately after every test method.

1.3 SCL Pragmas Reference

This section details the allowed syntactic forms of all pragmas.

1.3.1 General Syntax issues

All SCL pragmas can optionally end in **;** .

All SCL Pragmas in C mode must always appear at the outer most scope (outside the scope of any function). It is an error if they do not.

All SCL Pragmas in C++ mode must always appear at the outer most scope. They may not appear within the scope of any namespace, class, fuction, etc. It is and error unless they are in the outer most scope outside of all namespaces.

1.3.2 scl_func

Syntax

```
#pragma scl_func ( SUID, function-name ) ;opt

function-name :
    identifier

SUID :
    integer_constant_expression
```

Constraints

- Functions whose names are overloaded may not have scl_func applied to them.
- Integer constant expression must evaluate to a number between 1 and 2^{24}.
- The integer constant that the SUID resolves to must be unique among both functions and messages, or an error is recognized.

Semantics

Identifies the function as one that is interceptable and remotable. Information about this function and its parameters will be maintained within STRIDE.

1.3.3 scl_function

Syntax

```
#pragma scl_function ( function-name ) ;opt

function-name :
    identifier
```

Constraints

In C mode any function may have scl_function successfully applied to it[5]. In C++ mode only functions declared in the global namespace that have all POD types as return values and parameters may have scl_function() sucessfully applied. Furthermore, the function cannot be overloaded in the compilation unit or it will be an error.

Semantics

Identifies the function as interceptable and remotable. Information about this function and its parameters will be maintained within STRIDE.

[5] However, care must be taken with the application of scl_function to functions with static scope. Unless they are defined in the same compilation unit as the intercept module they may result errors at target application link time (unresolved symbol references)

1.3.4 scl_msg

Syntax

```
#pragma scl_msg ( SMID) ;opt
#pragma scl_msg ( SMID_Cmd, cmd-payload-type ) ;opt
#pragma scl_msg ( SMID_Rsp, rsp-payload-type ) ;opt
#pragma scl_msg ( SMID_CmdRsp, cmd-payload-type, rsp-payload-type )
;opt

SMID:
     integer_constant_expression

SMID_Cmd:
    integer_constant_expression

SMID_Rsp:
    integer_constant_expression

SMID_CmdRsp:
    integer_constant_expression

cmd-payload-type :
    type-specifier

rsp-payload-type:
    type-specifier

void
```

Constraints

SMID must be a non-parameterized macro name, enumeration constant or constant expression that resolves to an integer constant expression. If the SMID is a non-parameterized macro name or enumeration constant, then the message name is exactly the characters of the macro name or enumeration constant. If the SMID is not a macro name or enumeration constant, then the message name will be synthesized based on the numeric value of the SMID. The synthesized name is “Msg0x<hexdigits>” where <hexdigits> is the smallest number of hexadecimal digits from the set 0..9a..e that represent the numeric value of the SMID. (Notice this precludes leading 0’s. from the hexdigits string as well as capitol letters.). The name of the SMID, whether explictly specified via macro or enumeration constant name or implicitly created from a numeric expression cannot collide with any other message or function name or it is an error.

SMID, SMID Cmd, SMID Rsp and SMID Cmd Rsp must all be integer constant expressions that evaluate to a number within the range 1 to $2^{30} – 1$ and follows the rule for a STRIDE Message ID as outlined in previous sections. All unused fields (bits) in the SMID Attributes must be set to 0 or it is an erorr. More specifically a one-way command message must have St_r and Pu_r set to 0 or and error is recognized. Likewise a one-way response and broadcast message must have St_c and Pu_c set to 0 or an error is recognized. The Rsvd field (2 bits) must also be 0 or an error will be recognized.

Type specifier (either cmd-payload-type or rsp-payload-type) must designate a type that is not a pointer type or may be “void” to indicate the absense of any payload value.

Integer constant expression evaluates to a 32-bit value, but with the SMID format described in earlier sections. Bits 24 through 29 determine the payload configuration, and thus the allowable form of the pragma. It is an error if there is mismatch between the attribute bits and the number of arguments supplied.

Message payloads may not contain types that are ptrs to functions that have been further characterized by scl_ptr_flist (or scl_fptr_list) pragmas.

Any pointers contained in a one-way command message payload must have direction IN or it is an error.

Any pointers contained in a one-way response message payload or broadcast message payload must have direction RETURN or it is an error.

Any pointers contained in a the command portion of a two-way message must have direction IN or it is an error. Any pointers contained in the response portion of a two-way message must have direction RETURN or it is an error.

Semantics

Identifies a message of interest. The message will be interceptable and remotable. Information about the payloads and the associated types will be maintained by STRIDE.

1.3.5 scl_values

Syntax

```
#pragma scl_values ( absolute-specifier-item, values ) ;opt
#pragma scl_values ( base-specifier-item, relative-specifier-item, values) ;opt

absolute-specifier-item:
    absolute-specifier

base-specifier-item:
    base-specifier

relative-specifier-item:
    relative-specifier

values:
    enum-type-specifier
    constants-list

enum-type-specifier :
    enum identifier
    identifier

constants-list :
    number-list
    cf
number-list :
    integer-constant-expression
```



```
number-list , integer-constant-expression
```

Constraints/Semantics

If the absolute specifier designates a type specifier, then it must be a name for one of the types from the allowed types table below. Otherwise, the absolute specifier identifies a set of instances whose type must be one of the types from the allowed types table below or a typedef name for such.

The object instances or type identified by the pragma are constrained to the specified set of values. Values outside this set are considered “out of range.”

The values may consist of a single identifier that is an enum specifier, or a typedef name for an enumerated type. If so, the effect of the pragma is equivalent to an explicit values list consisting of all the enumerated constants of that enumerated type in declaration order.

Each value in the number-list is assigned both a symbolic name and an integral value. The symbolic name can be any of the following:

- The macro name if the value is defined by a macro. (The macro name must be a non-parameterized macro.)
- The enumerated constant name if the value is an enumeration constant.
- If the constant is not a simple macro name or enumerated constant name (that is, it is some kind of expression), then the name is the constant expressed as a decimal number, including its sign if the value is negative.

Two values in the number-list may not have the same symbolic name. This will be recognized as a syntax error. Two values in the number-list may have the same value as long as the symbolic name is different.

If a constant in the values list is outside of the represent able range of the type or instance to which it is applied (according to either host or target platform characteristics) but can be converted according to C-language conversion rules, a compiler warning will be issued. At runtime, assignment of such a constant to the field should follow C-language assignment rules (i.e., the value of the constant is converted to the type of the destination value, then the assignment is made).

If a constant in the ***values*** list is not type-compatible with the type or instance to which it is applied, a compiler error will be issued.

scl_values() does not change the underlying type of the instances to which it is applied. If those instances are referenced by other pragmas, the type is unaffected; however, the instances will have constrained values.

<table>
<tr><th colspan="5">Allowed C-types Table</th></tr>
<tr><td>C type</td><td colspan="3">Type categories</td><td>Allowed C-types</td></tr>
<tr><td>short, int, long, long long (signed and unsigned)</td><td rowspan="5">Integral types</td><td rowspan="6">Arithmetic types</td><td rowspan="7">Scalar types</td><td>Y</td></tr>
<tr><td>char (signed and unsigned)</td><td>Y</td></tr>
<tr><td>_Bool</td><td>N</td></tr>
<tr><td>enum {…}</td><td>N</td></tr>
<tr><td>float, double, long double</td><td>Floating-point types</td><td>N</td></tr>
<tr><td>T *</td><td colspan="2">Pointer types</td><td>(Y/N)[1]</td></tr>
<tr><td>T […]</td><td colspan="2">Array types</td><td rowspan="2">Aggregate types</td><td>N</td></tr>
<tr><td>struct {…}</td><td colspan="2">Structure types</td><td>N</td></tr>
<tr><td>union {…}</td><td colspan="3">Union types</td><td>N</td></tr>
<tr><td>T (…)</td><td colspan="3">Function types</td><td>N</td></tr>
<tr><td>void</td><td colspan="3">Void type</td><td>N</td></tr>
</table>

Note: scl_values can be applied to opaque pointers only.

1.3.6 scl_cast

Syntax

```
#pragma scl_cast(absolute-specifier-item, type-specifier-to) ;opt
#pragma scl_cast(base-specifier-item, relative-specifier-item,
type-specifier-to) ;opt

absolute-specifier-item:
    absolute-specifier

base-specifier-item:
    base-specifier

relative-specifier-item:
    relative-specifier

type-specifier-to :
    type-specifier-to *
    identifier
```

Constraints

- The sizeof() the instance(s) or type being reassigned must be equal to the sizeof() the type to which it is reassigned.
- The scl_cast() pragma may only be applied to the integral type and pointer type instances, or typedef names for integral types or pointer types. The type specifier must designate a pointer type or integral type.
- scl_cast() cannot be applied to bit fields.
- An error will result if scl_cast() is applied to a set of runtime values that intersects with the set of values explicitly specified by any pragma that has appeared earlier (by way of lexical position) in the source code. Colloquially, this means that scl_cast() can't "cast away" information conveyed by previous pragmas. (Refer to section 1.2.16, Absolute Specifiers, beginning on page 38 for a description and definition of runtime value sets.)

Semantics

Informs STRIDE that it should treat the object instances (identified by an absolute specifier item) as if they are of the reassigned type. If the reassigned type is further elaborated via other SCL pragmas, the instances will be treated as if the pragmas are additive.

scl_cast() changes the runtime value set it identified by transforming the identified set's type. Any pragma may use the transformed identity. No pragma may use the original identity.

Special Exceptions

scl_cast() may be used to cast a data item of type `void*` or `unsigned char*` to a union type if all members of the union are of type pointer.

1.3.7 scl_ptr

Syntax

```
#pragma scl_ptr(absolute-specifier_ptr, direction, usage) ;opt
#pragma scl_ptr(base-specifier_ptr, relative-specifier_ptr,
                direction, usage) ;opt

absolute-specifier-ptr:
    absolute-specifier

base-specifier-ptr:
    base-specifier

relative-specifier-ptr:
    relative-specifier

direction :
    "IN"
    "OUT"
    "INOUT"
    "RETURN"
    "INRETURN"

usage :
    "PRIVATE"
    "POOL"
```

Constraints

The absolute specifier (or the absolute specifier formed by the base and relative specifier) must identify either:

- An identifier that is a type *specifier* and must be a pointer type that is not *pointer to void* or *pointer to function*

  -or-

- A set of instances that have pointer type that is not *pointer to void* or *pointer to function*

There are certain restrictions on the allowed combinations of direction and usage. See previous sections for details. All disallowed combinations will be enforced by the compiler.

Semantics

Informs STRIDE as to additional semantics (primarily direction and usage) associated with a pointer type or identified instances.

1.3.8 scl_ptr_opaque

Syntax

```
#pragma scl_ptr_opaque( absolute-specifier-ptr ) ;opt
#pragma scl_ptr_opaque( base-specifier-ptr, relative-specifier-ptr
) ;opt
```



```
absolute-specifier-ptr:
    absolute-specifier

base-specifier-ptr:
    base-specifier

relative-specifier-ptr:
    relative-specifier

base-specifier:
    as-defined
```

Constraints

The absolute specifier (or the absolute specifier formed by the base and relative specifier) must identify either:

- An identifier that is a type *specifier* and must be a pointer type that is not *pointer to function*
  -or-
- A set of instances that have pointer type that is not *pointer to function*

There are certain restrictions on the allowed combinations of direction and usage. See previous sections for details. All disallowed combinations will be enforced by the compiler.

- An error will result if scl_ptr_opaque() is applied to a set of runtime values that intersects with the set of values explicitly specified by any pragma that has appeared earlier (by way of lexical position) in the source code. Colloquially, this means that scl_ptr_opaque() can't "opaque away" information conveyed by previous pragmas. (Refer to section 1.2.16, Absolute Specifiers, beginning on page 38 for a description and definition of runtime value sets.)

Semantics

Informs STRIDE that the type or instances of the pointers identified are to be treated as if they were of type *pointer to void*.

1.3.9 scl_ptr_sized

Syntax

```
#pragma scl_ptr_sized(absolute-specifier-ptr, direction, usage,
                      max-elements) ;opt
#pragma scl_ptr_sized(absolute-specifier-ptr, direction, usage,
                      max-elements, absolute-specifier-count-field)
                      ;opt
#pragma scl_ptr_sized(base-specifier-ptr, relative-specifier-ptr,
                      direction, usage, max-elements, relative-
                      specifier-count-field) ;opt
#pragma scl_ptr_sized(base-specifier-ptr, relative-specifier-ptr,
                      direction, usage, max-elements) ;opt

absolute-specifier-count-field:
    absolute-specifier designating the count field

absolute-specifier-ptr:
    absolute-specifier designating the pointer

base-specifier-ptr:
    base-specifier

relative-specifier-ptr:
    relative-specifier for pointer

relative-specifier-count-field:
    relative-specifier for count

direction :
    "IN"
    "OUT"
    "INOUT"
    "RETURN"
    "INRETURN"

usage :
    "PRIVATE"
    "POOL"

max-elements:
    Integer-constant-expression
```

Constraints

The absolute specifier (or the absolute specifier formed by the base and relative specifier) must identify either:

- An identifier that is a type *specifier* and must be a pointer type that is not *pointer to void* or *pointer to function*
  -or-
- A set of instances that have pointer type that is not *pointer to void* or *pointer to function*

There are certain restrictions on the allowed combinations of direction and usage. See previous sections for details. All disallowed combinations will be enforced by the compiler.

There are certain memory allocation/initialization polices that must be followed for sized pointers. See previous sections for details.

When a sized pointer pragma prescribes a count field, there are certain restrictions on the count field relative to the location of the pointer.

- The count field may not reside in any block pointed to by the sized pointer in question.
- The count field may reside in the same memory block as the pointer.
- The count field may reside in any ancestor block or descendent of an ancestor block that is pointed to by a path containing only single pointers.
- The count field and pointer may also reside in different payloads (i.e., one may be in the command payload and the other may be in the return payload). When this is the case, the path through the payload blocks from root to count and/or size field must contain only single pointers (if there are any).
- If the count field resides in a union member, the sized pointer must also reside in exactly the same union member.

The type of the element count field must be one of the standard integer types.

max elements is an integer constant expression in the range 1 to a configuration dependent maximum. Furthermore, the *max elements* value must fit within the type of the element count field, or an error will result.

As a special case, if the pointer direction is IN and a count field has been specified, then the *max_elements* constant expression may evaluate to zero (0). Zero means that the maximum number of elements is not specified and the system limit should be used.

Semantics

Informs STRIDE as to additional attributes of pointers that are used to point to a series of elements allocated in contiguous memory.

When a scl_sized_ptr() is applied to a pointer it is said to be a “sized pointer.”

At runtime:

- If the actual value of the count field is zero, then the value of the (sized) pointer is considered to be undefined and no particular meaning can be ascribed to it.
- If the actual value of the count field is less than zero, it will be treated as if it were zero.
- If the actual value of the count field is greater than the (non-zero) max-elements, it will be treated as if it were equal to the max elements.

1.3.10 scl_string

Syntax

```
#pragma scl_string( absolute-specifier-string, max-count ) ;opt
#pragma scl_string( base-specifier-string, relative-specifier-
string, max-count ) ;opt

max-count :
    Integer-constant-expression

absolute-specifier-string:
    absolute-specifier

relative-specifier-string:
    relative-specifier

base-specifier-string:
    base-specifier
```

Constraints

The absolute specifier must identify either of the following:

- A type specifier that can be a pointer type or an array type;
    - If it is a pointer type, the type pointed to must be either char (with or without the signed/unsigned qualification) or short (with or without the signed/unsigned qualification), or a type name for such.
    - If it is an array type, the array type must be a single-dimension array with an element type of either char (with or without the signed/unsigned qualification) or short (with or without the signed/unsigned qualification), or a type name for such. The max-count must not exceed the declared length of the array or it is an error.
- A set of instances that are of pointer type or array type with the same constraints as above.

Max count is an integer constant expression that evaluates to within the range 1 to a configuration dependent maximum.

Semantics

If the element or pointed-to type is of type char (either signed or unsigned), then the string is assumed to be an ASCII string. If it is of type short (either signed or unsigned), then the string is assumed to be a UNICODE string.

The strings are considered to be null-terminated; the actual length of any string instance is the minimum of the max-count and the location of the null termination character.

Special case: the maximum length string (as constrained by the max count) does not have a null termination character.

1.3.11 scl_union

Syntax

```
#pragma scl_union ( union-absolute-specifier, fixed-active-member-index)
                        ;opt
#pragma scl_union ( union-absolute-specifier,
                    discriminant-absolute-specifier ) ;opt
#pragma scl_union ( base-specifier-union, union-relative-specifier,
                        discriminant-relative-specifier ) ;opt
#pragma scl_union (base-specifier-union, union-relative-specifier,
                        fixed-active-member-index) ;opt

    fixed-active-member-index :
        Integer-constant-expression

    union-absolute-specifier :
        absolute-specifier for the union

    discriminant-absolute-specifier :
        absolute-specifier for the discriminant

    base-specifier-union :
        base-specifier

    union-relative-specifier :
        relative-specifier for the union

    discriminant-relative-specifier :
        relative-specifier for discriminant
```

Constraints

The union absolute specifier identifies either of the following:

- A type specifier that must be of type *union*
  -or-
- A set of instances that must be of type *union*

The fixed active member index is an integer constant expression in the range of 0 – *n*-1, where *n* is the number of members in the union and identified in the n^{th} member.

The absolute specifier for the discriminant identifies a field whose value determines which member of the union is active. The type of this field must be either integer, enumerated type or typename for such.

When a union pragma prescribes a discriminant field, the following restrictions exist on the discriminant field relative to the union:

- The discriminant may be an internal discriminant and reside in exactly the same offset within every union member, as described in previous sections.
- The discriminant field may reside in the same memory block as the union.
- The discriminant field may reside in any ancestor block or descendent of an ancestor block that is pointed to by a path containing only single pointers.
- The discriminant field and union may also reside in different payloads (i.e., one may be in the command payload and the other may be in the return payload). When this is the case, the path through the payload blocks from root to count and/or size field must contain only single pointers, if there are any.

Semantics

Refer to the Unions section on page 22 for background on all union concepts.

1.3.12 scl_union_activate

Syntax

```
#pragma scl_union_activate (base-specifier, relative-member-
                            specifier, constant-value-list) ;opt

constant-value-list :
    Integer-constant-expression
    constant-value-list , Integer-constant-expression

relative-member-specifier:
    relative-specifier

base-specifier:
    base specifier;
```

Constraints

- The absolute specifier formed from the base specifier and relative member specifier must be a union member. The absolute specifier must be of the form <U>.<identifier>, where <identifier> is the name of the union member and <U> is itself an absolute specifier that identifies the union. For this scl_union_activate() pragma to be valid, <U> must have been used in an scl_union() pragma identifying <U> as a discriminated union. If no such pragma exists, then this scl_union_activate() instance will raise an error.
- The integer constant expressions in the constant value list must all evaluate to integer constants that are within the range represent able by the discriminant field (after application of any relevant scl_cast() or scl_values() pragmas).
- An error will result if there are two integer constant expressions within the same list that evaluate to the same value.

- An error will result if two different members of the same union have integer constant expressions that evaluate to the same value.
- Only one scl_union_activate pragma may be applied to any set of member instances.

Description

Refer to the Unions section on page 22 for background on all union concepts.

1.3.13 scl_fptr_list

This pragma is deprecated.

Syntax

```
#pragma scl_fptr_list ( absolute-specifier-ptr, candidate-name-list
                      ) ;opt
#pragma scl_fptr_list ( base-specifier-ptr, relative-specifier-ptr,
                      candidate-name-list ) ;opt

#pragma scl_fptr_list (absolute-specifier-ptr, assigned-SUID,
                      string-literal-name) ;opt
#pragma scl_fptr_list (base-specifier-ptr, relative-specifier-ptr,
                      assigned-SUID, string-literal-name ) ;opt

absolute-specifier-ptr :
    absolute-specifier

base-specifier-ptr :
    base-specifier

relative-specifier-ptr :
    relative-specifier

candidate-name-list :
    candidate
    candidate-name-list , candidate

candidate :
    identifier

assigned-SUID :
    integer-constant-expression

string-literal-name :
    "identifier "
```

Constraints

The absolute specifier (or combination of base and relative specifier) identifies a set of instances of type *pointer to function*.

Each candidate in the candidate names list is a function name that must appear in an scl_func() or scl_function() pragma. The signature of the function (i.e. the return type and parameter types) need not be the same as the type of the field. Their prototypes need not have the same number of parameters, nor types, nor do the return types need to match. If the prototypes do not match a warning (but not an error) will be issued.

assigned_SUID must be an integer constant expression in the range allowed for a SUID. Furthermore, the value must not collide with any other SUID or it is an error.

The string literal name is an identifier enclosed in double quotes. The identifier must be syntactically suitable for a function name. For each such string literal name STRIDE will treat it as if it's declaration had been seen and it had been identified with scl_func() or scl_function(). Each such string literal name must not collide with any other scl_func() or scl_function() pragma, in either name or SUID (in the case of scl_func()).

When a string literal name is used, its effect is to declare a function with the literal name (and parameters) and make the name available as a function to allow qualification of its parameters. For example:

```
typedef void (*PF)(int *pi);
typedef struct S{
   PF   pf;
} S;
scl_fptr_list(S.pf, "f", "g");
scl_ptr(f.pi, OUT, PRIVATE);
```

is correct SCL, because the scl_fptr_list() pragma has the effect of the source below. The source in ***blue*** is source that is internally synthesized as a result of the pragma:

```
typedef void (*PF)(int *pi);
typedef struct S{
   PF   pf;
} S;
scl_fptr_list(S.pf, "f", "g");
     void f(int *pi);
     #pragma scl_function(f)
     void g(int *pi);
     #pragma scl_function(g);
scl_ptr(f.pi, OUT, PRIVATE);
```

Semantics

The scl_fptr_list() pragma identifies a set of specific functions whose address may be passed as part of the payload field.

1.3.14 scl_ptr_flist

This pragma is deprecated.

Syntax

```
#pragma scl_ptr_flist( absolute-specifier-ptr, candidate-list ) ;opt
#pragma scl_ptr_flist( base-specifier-ptr, relative-specifier-ptr,
                       candidate-list) ;opt

absolute-specifier-ptr :
    absolute-specifier

base-specifier-ptr :
    base-specifier

relative-specifier-ptr :
    relative-specifier

candidate-list :
    candidate-list, candidate
    candidate-list, string-literal-name
    candidate
    string-literal-name

candidate :
    identifier

string-literal-name :
    "identifier"
```

Constraints

The absolute specifier (or combination of base and relative specifier) identifies a set of instances that have type pointer to function type.

Each candidate in the candidate names list is a function name that must appear in a scl_func() or scl_function() pragma. The type of the function (i.e. the return type and parameter types) need not be the same as the type of the field. Their prototypes need not have the same number of parameters, nor types, nor do the return types need to match. If the prototypes do not match a warning (but not an error) will be issued.

The string literal name is an identifier enclosed in double quotes. The identifier must be syntactically suitable for a function name. For each such string literal name STRIDE will treat it as if it's declaration had been seen and it had been identified with scl_func() or scl_function(). Each such string literal name must not collide with any other scl_func() pragma, in either name or SUID.

When a string literal name is used, its effect is to declare a function with the literal name (and parameters) and make the name available as a function to allow qualification of its parameters. For example:

```
typedef void (*PF)(int *pi);
typedef struct S{
   PF   pf;
} S;
```



```
scl_fptr_list(S.pf, "f", "g");
scl_ptr(f.pi, OUT, PRIVATE);
```

is correct SCL, because the scl_fptr_list pragma has the effect of the source below. The source in ***blue*** is source that is internally synthesized as a result of the pragma

```
typedef void (*PF)(int *pi);
typedef struct S{
   PF   pf;
} S;
scl_fptr_list(S.pf, "f", "g");
     void f(int *pi);
     #pragma scl_function(f)
     void g(int *pi);
     #pragma scl_function(g);
scl_ptr(f.pi, OUT, PRIVATE);
```

TBD: Default Candidate info

Semantics

See previous section 1.2.4 Pointers to Functions.

The scl_ptr_flist() pragma identifies a set of specific functions whose address may be passed as part of the payload field.

1.3.15 scl_msg_bind (removed)

The scl_msg_bind pragma has been removed. This entry exists as a placeholder to preserve section numbering of the pragmas.

1.3.16 scl_tracepoint

Syntax

```
scl_tracepoint ( STPID ) ;opt
scl_tracepoint ( STPID, tp-payload ) ;opt

tp-payload :
    type-name

type-name:
    identifier
```

The name scl_tp() is deprecated and follows the syntax for scl_tracepoint exactly.

Constraints

type name is a type specifier for a type that is the (optional) payload of the tracepoint. As long as the type either is not a pointer or does not contain a pointer (in the case of a struct or union type), the entire payload will automatically be marshaled from target to host and be available for display on the host. The format of the displayed values can be further

described, if desired using the scl_tracepoint_format() pragma (See the next section for details on scl_tracepoint_format()).

Trace point payloads have several additional restrictions:

- they may not contain unions
- they may not contain pointers unless the pointer (or its type) has had scl_ptr_opaque applied to it, or the pointer's type is void * or it is a pointer to an incomplete type that is never completed in the compilation unit.
- they may not contain conformant arrays

Semantics

Identifies the STPID with a trace point. The target application must be instrumented with calls to the srTracePoint() API for the trace points to be displayed.

1.3.17 scl_tracepoint_format

Syntax

```
scl_tracepoint_format ( STPID, format-string [, format-args] ) ;opt

format-string :
    string-literal

format-args :
    arg
    format-args , arg

arg :
    relative-specifier
```

The name scl_ tracepoint_format() is deprecated. It has exactly the same functionality as scl_tracepoint_format() with no other changes.

Constraints

STPID has been previously identified via the trace point pragma scl_tracpoint().

format-string is a string literal that contains the string to display when the trace point is activated. It may optionally contain format specifications identical to those used by the ANSI C library printf() routine that correspond to format args. Format args, if present, must be members of the type designated as the tracepoint payload type.

format string and format args follow the printf style of specifying data to format and display.

TODO: This section needs some cleanup

Semantics

Defines trace point payload formatting rules.

The scl_tracepoint format pragma defines a format string for a printf-style trace point.

1.3.18 scl_cclass

This pragma is deprecated in favor of scl_brew_class

Syntax

```
scl_cclass ( base-SUID, absolute-specifier-struct ) ;opt
scl_cclass ( base-SUID, absolute-specifier-struct [,
member1..,memberN] ) ;opt

base-SUID :
    SUID

absolute-specifier-struct :
    absolute-specifier

memberN :
    identifier
```

Semantics And Constraints

Semantics and constraints are exactly the same as _class() with the following differences:

base-SUID explicitly defines a starting point for SUID assignment for the member functions of the class.

1.3.19 scl_brew_class

Syntax

```
scl_brew_class (absolute-specifier-struct [, member1..,memberN] )
;opt

absolute-specifier-struct :
    absolute-specifier

memberN :
    identifier
```

Constraints

absolute-specifier designates a type name or set of instances that are of structure type, T. If no members are specified in the pragma, then members of the structure that are of type pointer to function and whose first parameter is of any pointer type, will be identified as class methods. Each identified member will be assigned a SUID that is base-SUID + n where n is the declaration order of the members starting with n = 1. The name of the member will also be changed so that it is prefixed with "T_".

If any members are specified then the identifier for each must exactly match a member of the structure T. Each such member must be of type pointer to function and whose first parameter is of whose first parameter is of any pointer type. Only the explicitly specified members of T will be identified and treated as class methods. The first parameter of every member will be treated as if its type were "void *" regardless of its actual type.

If only the absolute specifier for the struct is specified (i.e., no members are explicitly identified) and there are no members automatically identified as class methods then a warning will be issued.

It is an error if any explicitly specified member is either not found or does meet the requirements to become a class method.

Semantics

Converts a struct into a brew-based class.

1.3.20 scl_conform

Syntax

```
scl_conform ( container-struct-specifier, relative-specifier-count-
field, max-count ) ;opt

container-struct-specifier :
    absolute-specifier

relative-specifier-count-field :
    relative-specifier

max-count :
    integer-constant
```

Constraints

container struct specifier must identify either a type name for a structure or a field that is of type pointer to structure. The last member of the struct must be an array. This member is referred to as the "conformant array." The declared length of the array is ignored within SCL. The structure is referred to as the conformant array structure.

relative specifier count field identifies a member of the conformant array structure that, at run-time, specifies the number of elements in the conformant array. This field is relative to container struct specifier.

The count field must be located in the same payload block as the conformant array. In other words, its relative specifier cannot include indirection (either * or ->).

max count is an integer constant expression that gives the maximum number of elements and is in the range 1 to a configuration dependent maximum. Furthermore, the *max count* value must fit within the type of the element count field, or an error will result.

The element type of the conformant array may not contain a conformant array structure.

A conformant array structure may not be pointed to by a sized pointer. If a conformant array structure is pointed to by a sized pointer, a compile-time error will be issued.

A conformant array structure may not be a member of any other structure or union. If a conformant array structure is contained as a member of any other structure or union then a compile-time error will be issued.

A conformant array structure may not be passed as a function parameter. If a conformant array structure is used as a function parameter then a compile-time error will be issued.

Semantics

Specifies a struct type or pointer to such that acts as a conformant array structure.

Conventions For Out-Of-Range Count Fields

Given a payload that contains a conformant array structure, it is possible that agent setting up such a payload will assign a value to the count field that is out of range. An out of range value is one that is either less than 0 or greater than the prescribed maximum size. (A conformant array count field must reside in the same block as an array, so that there no possibility of the out of range condition caused by a null value of the pointer to the count field)

When the count field is out of range in a payload containing a conformant array, it is treated as if it were 0 and a runtime warning is issued.

Examples

The examples below are all based on this SCL

```
typedef struct S { int size; ary[1]} S;
#pragma scl_conform(S, size, 100);
```

SCL Examples (Valid)	**Explanation**
int f(S *ps);	ps is a pointer to a conformant array structure
int f(S **ppS);	ppS is a pointer to a conformant array structure
struct S2 { S *ps; int i; } s2;	ps is a pointer to a conformant array structure.
SCL Examples (Invalid)	**Explanation**
int f(S s);	Conformant array structure may not be a parameter to a function. Compiler error is issued.
struct S2 { S s; int i; } s2;	Conformant array structure may not be a member of another structure. Compiler error is issued.

1.3.21 scl_fptr_anonymous

Pragma to identify anonymous candidates.

There is no base/relative form for this pragma.

Syntax

```
#pragma scl_fptr_anonymous( absolute-specifier-ptr,
                            anon-candidate-list ) ;opt

absolute-specifier-ptr :
    absolute-specifier


anon-candidate-list :
    candidate-list, string-literal-name
    string-literal-name


string-literal-name :
    "identifier"
```

Constraints

The absolute specifier (or combination of base and relative specifier) identifies a set of instances that have type pointer to function type.

The absolute specifier must start with a function identifier. It may not be a type specifier. (The reason for this is to disallow the same anonymous callback to be used in more than one function.)

The string literal name is an identifier enclosed in double quotes. The identifier must be syntactically suitable for a function name. For each such string literal name STRIDE will treat it as if it's declaration had been seen and it had been identified with scl_func() or scl_function(). Each such string literal name must not collide with any other scl_func() pragma, in either name or SUID.

When a string literal name is used, its effect is to declare a function with the literal name (and parameters) and make the name available as a function to allow qualification of its parameters. For example:

```
typedef void (*PF)(int *pi);
typedef struct S{
   PF   pf;
} S;
int parent(S s);
#pragma scl_function(parent)
scl_fptr_anonymous(f.s.pf, "f", "g");
scl_ptr(f.pi, OUT, PRIVATE);
```

is correct SCL, because the scl_fptr_anonymous pragma has the effect of the source below. The source in ***blue*** is source that is internally synthesized as a result of the pragma

```
typedef void (*PF)(int *pi);
```



```
typedef struct S{
   PF   pf;
} S;
scl_fptr_anonymous(f.s.pf, "f", "g");
     void f(int *pi);
     #pragma scl_function(f)
     void g(int *pi);
     #pragma scl_function(g);
scl_ptr(f.pi, OUT, PRIVATE);
```


1.3.22 scl_fptr_named

For capturing named candidates.

There is no base/relative form of this pragma.

Syntax

```
#pragma scl_fptr_named ( absolute-specifier-ptr,
                              named-candidate-list ) ;opt

absolute-specifier-ptr :
    absolute-specifier

named-candidate-list :
    named-candidate-list, named-candidate

named-candidate :
    identifier
```

Constraints

The absolute specifier identifies a set of instances that have type pointer to function type.

Each named candidate in the named candidate list is a function name that must appear in a scl_func() or scl_function() pragma. The type of the function (i.e. the return type and parameter types) need not be the same as the type of the field. Their prototypes need not have the same number of parameters, nor types, nor do the return types need to match.

A warning is issued when all named candidates on the same list do not have matching types. To have matching types, the function types for all candidates must be compatible in the C Language sense (as defined by ISO/IEC 9899:1999)

1.3.23 scl_test_class

Identifies test classes.

Syntax

```
#pragma scl_test_class
                        (class_specifier[,init_method[,deinit_method]
                        ]) ;opt

class_specifier :
    ClassName
    namespace_specifier ClassName

namespace_specifier:
    ::
    namespaceName ::
    namespace_specfier namespaceName ::

init_method:
    class-method-identifier

deinit_method:
    class-method-identifier
```

Constraints

This pragma requires the compilation language to be C++. If the compilation language is not C++ and this pragma is encountered, then an error is issued and this pragma is ignored.

The class identified will become a Stride Test Class.

The test class identified must:

- Have a public constructor.
- The constructor may have parameters, but they must all be POD type.
- Have one or more member functions that suitable as a test method. For a member function to be a test method it must
    - be declared within the test class (method not declared, but inherited from a base class cannot be test methods)
    - have a return type of bool, an integral type (signed or unsigned long, int, short, char) or void.
    - have an empty parameter list. That is, declared as f() or f(void).
    - not be a templatized function
    - not be an overloaded operator
    - The class cannot be a pure virtual class
    - not be a static member.
- Cannot be a templated class.
- Cannot be a nested class.

The class specifier identifies the test class. If the test class is not in the global namespace it must include the namespace specifier for the class. If the test class is in the global namespace, that may be explicitly indicated by the global namespace specifier "::".

If an init_method (initialization method) is specified, it must be the method name of a test class method. This method will no longer be a test class method with this specification.

If a deinit_method (deinitialization method) is specified, it must be the method name of a test class method. This method will no longer be a test class method after this specification.

1.3.24 scl_test_setup

Identifies of the setup fixture of a previously-identified scl_test_cclass, scl_test_class, or scl_test_flist.

Syntax

```
#pragma scl_test_setup (test_specifier, function_name ) ;opt

test_specifier :
    cclass_specifier
    class_specifier
    test_unit_name

cclass_specifier :
    struct-identifier

class_specifier :
    ClassName
    namespace_specifier ClassName

test_unit_name :
    identifier

namespace_specifier:
    ::
    namespaceName ::
    namespace_specfier namespaceName ::

function_name:
    class-method-identifier
     routine-identifier
```

Constraints

This pragma identifies the setup fixture of an existing test_cclass (i.e. a struct with scl_cclass applied to it), test_class (i.e. a class with scl_test_class applied to it), or an existing test unit (i.e. name with scl_test_flist applied to it).

If the setup fixture is specified using a class specifier, the method must come from the pool of test methods. Once identified as a setup fixture the method is no longer a test method.

There may be only one setup fixture per test c-class (scl_test_cclass), test class (scl_test_class) or test unit (scl_test_flist).

The c-class specifier must match the struct specifier of a prior consumed scl_test_cclass pragma.

The class specifier must match the class specifier of an scl_test_class pragma that has already been consumed.

The test unit name must match the test unit name of an scl_test_flist pragma that has already been consumed.

1.3.25 scl_test_teardown

Identifies of the teardown fixture of a previously-identified scl_test_cclass, scl_test_class, or scl_test_flist.

Syntax

```
#pragma scl_test_teardown (test_specifier, function_name ) ;opt

test_specifier :
    cclass_specifier
    class_specifier
    test_unit_name

cclass_specifier :
    struct-identifier

class_specifier :
    ClassName
    namespace_specifier ClassName

test_unit_name :
    identifier

namespace_specifier:
    ::
    namespaceName ::
    namespace_specfier namespaceName ::

function_name:
    class-method-identifier
    routine-identifier
```

Constraints

This pragma identifies the teardown fixture of an existing test_class (i.e. a class with scl_test_class applied to it) or an existing test unit (i.e. name with scl_test_flist applied to it).

If the teardown fixture is specified using a class specifier, the method must come from the pool of test methods. Once identified as a teardown fixture the method is no longer a test method.

There may be only one teardown fixture per test c-class (scl_test_cclass), test class (scl_test_class) or test unit (scl_test_flist).

The c-class specifier must match the struct specifier of a prior consumed scl_test_cclass pragma.

The class specifier must match the class specifier of a prior consumed scl_test_class pragma.

The test unit name must match the test unit name of a prior consumed scl_test_flist pragma.

1.3.26 scl_test_flist

Associates a test unit name with a list of test functions.

Syntax

```
#pragma scl_test_flist ( test-unit-name , test-function-name { ,
test-function-name } ) ;opt

test-unit-name :
    string-literal-name

string-literal-name :
    "identifier"

string-literal-name :
    "identifier"

test-function-name :
    identifier
```

Constraints

Usage of this pragma requires an include of srtest.h.

The test-unit-name may not have been specified with a prior scl_test_flist pragma.

The test-unit-name may not be the name of an existing routine.

The test-function-name(s) not be a specifier(s) of other pragmas. It may not be used for pragmas like scl_ptr_flist. It also means that its return value may not be casted using scl_cast.

The function declared for test-function-name may not be declared static.

The function declared for test-function-name may not be repeated in the pragma.

The function declared for test-function-name may not exist in any other scl_test_flist pragma.

In C++ mode, the function declared for test-function-name must have used ‘extern “C”’.

The function declared for test-function-name must have a void parameter list.

The function declared for function-name must return a pass/fail results. The return type may be declared void or an integer type (bool is acceptable in C++ mode). If void is the return type, any calls to the test function default as successful.

Semantics

This pragma signifies a given test-unit-name is to be a collection of test functions.

1.3.27 scl_test_cclass

Associates a class (or C struct) as a composite type containing a list of test functions.

Syntax

```
#pragma scl_test_cclass ( struct-specifier , init-function-name { ,
deinit-function-name } ) ;opt

struct-specifier :
    identifier

init-function-name :
    identifier

deinit-function-name :
    identifier
```

Constraints

Usage of this pragma requires an include of srtest.h.

The struct-specifier may not have been specified with a prior scl_test_cclass pragma.

The struct -specifier must be of type struct or class.

The struct -specifier must be a plain old data type.

The struct -specifier may not be a template class.

The struct -specifier may not be a nested class.

The init-function-name must be an existing function.

The init-function-name must not have been pragmatized using any prior declared or later declared pragmas (i.e. scl_func, scl_function. scl_test_function, scl_ptr_flist).

The deinit-function-name may not have the same name as the init-function-name.

The deinit-function-name must be an existing function if specified with the pragma.

The deinit-function-name must not have been pragmatized using any prior declared or later declared pragmas (i.e. scl_func, scl_function. scl_test_function, scl_ptr_flist).

Semantics

This pragma signifies a given struct-specifier defines a collection of test functions.

1.3.28 Other Pragmas

In addition to the pragmas specific to SCL, the SCL language translator accepts a number of pragmas that are commonly accepted by other compilers. These are detailed here

1.4.1 #pragma once

The #pragma once, when placed at the beginning of a header file indicates that the file is written in such a way that including it several times has the same effect as including it once.

1.4.2 #pragma pack

The pack pragma is used to alter the layout of fields within a struct or union by overriding the alignment policy currently in effect. Thus the pack pragma may alter the layout so that the members are more closely packed than they would be otherwise.

Syntax

```
#pragma pack (alignment-value) ;opt
#pragma pack ();opt
#pragma pack (show) ;opt

alignment-value:
    1, 2, 4, 8, 16, 32, 64, 128
```

Constraints

By default the packing value is “not set”. All structures and unions are layed out according to the target characteristics in effect for the compilation.

Pragma pack with a specified alignment-value sets the pack alignment for subsequent declarations until the end of compilation or until another pack pragma is encountered. [6]

Pragma pack with no argument resets off any earlier specification of a pack pragma to the default value.

Pragma pack with the argument “show” will display a warning specifying the pack alignment-value currently in effect.

[6] A pack pragma has the potential to affect a structure layout only if the pack value is less than the alignment requirements of at least one of the structure members. The alignment requirements are for each type are part of the target characteristics specified for each compilation process.

1.4.3 #pragma warning

This functionality is applicable only when compiling with Microsoft compatibility. The warning pragma predominantly appears when including Microsoft include files. The compiler will ignore these warning pragmas during compilation. It will not generate a warning for these pragmas being unrecognized nor will it remember the warning's directive to generate a warning or to turn off future warnings.

#pragma warning("ABC") // does not generate a warning

#pragma warning(disable : 4412) // does not disable 4412 for later occurrences.

