Host Runtime Transport Specification

Version 1.0

Published by

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

Contents

1. About This Guide

1.1. Purpose

This document provides background and customization information about the Host Transport services. These host-based services represent the communication peer to the PAL services on the target device.

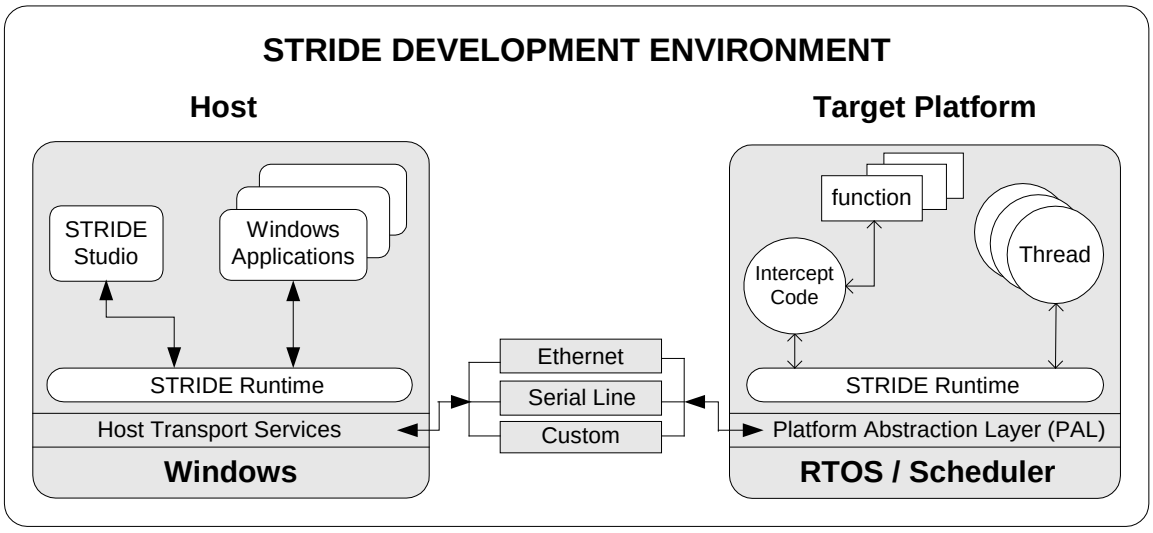

Figure 1. STRIDE Embedded Software Verification Platform

1.2. Document Conventions

This document uses the following conventions:

…	Indicates time passing, activity
	Indicates the developer should take special care to avoid errors
	Indicates additional information that could affect performance
	Indicates interface through use of messaging

1.3. Terms

I-block	STRIDE Communication Model (SCM) term for a packet of data transferred between platforms
message	A communication mechanism between two threads
module	A file containing one (1) or more functions

NID	Notification Identifier
pool memory	Memory allocated from a common pool used by application threads
private memory	Non-pool memory that is owned by a sending application thread
process	Implies a separate address space which typically does not apply to a task or thread
proxy	Software that uses an interface to connect a user to a remote device
public interface	Exposed to another component/unit
RFC	Remote function call
sender	The originator of a message
stub	Temporary code written to replace a unit that has yet to be written or is otherwise unavailable
task	Often used interchangeably with “thread”
thread	An independent entity running under the control of an Operating System
Transport DLL	A plug-in library that provides methods for transmitting STRIDE messages to/from the target.
Transport Server	Manages the connection between the Host and Target, using a single active Transport DLL to create the connection and send/receive data.

1.4. PAL

The PAL, or Platform Abstraction Layer, provides a consistent interface for the STRIDE Runtime regardless of the operating system or data transport used. This interface layer is necessary given the broad variety of operating systems and data transports that exist within embedded systems today.

A small set of functions, written according to the PAL specification, provides a virtual link between your operating system and platform transport mechanism to the STRIDE Runtime. Through the Platform Abstraction Layer, the STRIDE Runtime becomes independent of any specific operating system or transport. The PAL is designed to use standard concepts and services present in almost all operating systems and transport mechanisms. To complete the PAL, you’ll need to be familiar with concepts such as event signaling, scheduling, timers, critical section protection, memory allocation and data transfers, all of which are described in detail in the *STRIDE Platform Abstraction Layer Specification* PDF .

The “pal.h” header file provided with the STRIDE installation contains all the function prototypes necessary for writing the PAL. The pal.h header file is provided in the *STRIDE Platform Abstraction Layer Specification* PDF .

1.5. SLAP

The Simplified Link Application Protocol (SLAP) is a link protocol that is used to transmit and receive frames of data between two platforms. The sole purpose of the SLAP is to guarantee that frames are successfully transmitted between the two platforms.

The SLAP verifies the integrity of the data contained within the frame and is able to resynchronize quickly in the event of missed frames. This is accomplished through the use of “data stuffing”.

More detailed description of SLAP is provided in the *STRIDE Platform Abstraction Layer Specification* PDF .

1.6. Related Documents

The following publications are also available through STRIDE Online Help:

STRIDE Platform Abstraction Layer Specification

STRIDE Runtime Developer’s Guide

STRIDE Communication Language Reference Guide

2. Host Transport Services

2.1. Introduction

The Host Transport Services define an interface that enables the STRIDE Runtime on your target to send data to and receive data from the target. The host platform runs a version of the STRIDE Runtime -- the STRIDE Transport Server connects the Transport DLL to the STRIDE Runtime, thus providing indirect access to the target from STRIDE Studio, Autoscript, and other STRIDE applications. Several common transports are already supported within the STRIDE Transport Server, including serial and TCP/IP.

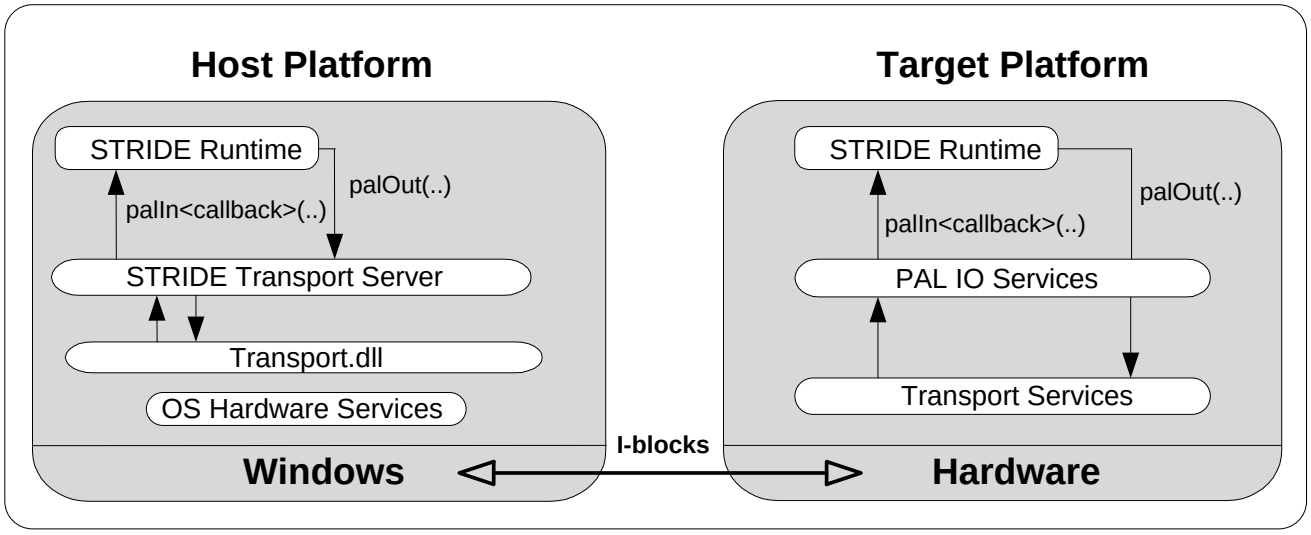

Figure 2. Transport Block Diagram

2.2. Host Services

The Host Transport Services allow the STRIDE Transport Server on the host to connect with the native target transport mechanism. The Host Transport Services are defined in "transport.h" and each Transport DLL must implement a concrete class derived from IStrideTransport. Each implementation of such a class must implement the four methods listed in Table 2.

Table 1. IStrideTransport Required Methods

Method Name	Description
Connect	Establish a connection with the device.
Disconnect	Close or terminate the connection with the device.
SendData	Send data from the host to the target.
ValidateProperties	Validate the current state of the transports properties.

In addition, the transport implementation must arrange to receive incoming data from the target. This is often done in a separate thread whose lifetime is tied to the connection state of the transport (i.e. the thread is only active when as the transport is connected). When this background thread receives data, it should call the ReturnData method on the StrideTransport instance to send the data to the Transport Server and eventually into the STRIDE Runtime.

The StrideTransport base class provides default implementations of the following methods. These default implementations should be sufficient for most needs, but the methods can be overridden as needed.

Table 3. IStrideTransport Inherited Methods

Method Name	Description
AddListener	Adds a subscriber to this Transport DLL. This method is called by the Transport Server to subscribe to the data and error events described by the IStrideTransportListener interface.
RemoveListener	Removes a subscriber.
Name	Returns the transport name.
Status	Returns the current ConnectStatus value.

Properties	Returns the current PropertyList container.
ReturnData	Send incoming data back to the Transport Server for routing to the host runtime. This method should be called by the Transport DLL whenever it receives a complete STRIDE message from the Target device.
DumpData	Send raw data buffer to the TransportServer for inclusion in the application log (under option). When the TransportDataDump property is enabled (in the Transport Server) , any buffers sent via this method will be written in human readable form to the STRIDE application log.
QueueData	Used to queue data (bytes) for eventual dumping. This is useful when received data is not known to be a complete STRIDE message until subsequent data is received.
DumpQueue	Causes any data in the current queue to be dumped (by calling DumpData).
SendEvent	Publishes an event to all current listeners. The event consists of a message string, a type, and a level (or severity).
ConnectNotify	Notifies all listeners when the status has transitioned to CONNECTED.
DisconnectNotify	Notifies all listeners when the status has transitioned to DISCONNECTED.

Each Transport DLL must also implement four global methods to provide a basic object factory and API version information.

Table 3. Required Global Functions

Method Name	Description
getTransport	Returns an instance of the StrideTransport object that your DLL implements. The current use model in the Transport Server requires that the created instance be a singleton (i.e. the same object instance must be returned by all calls to getTransport)
cleanupTransport	Called when the transport is unloaded to allow the singleton transport object instance to be

	freed and any other resources to be deallocated.
getAPIVersion	Must return the value of `TRANSPORT_API_VERSION` for which the transport was compiled.
getTransportVersion	Returns a version number for the Transport DLL. This value is not currently used by the Transport Server.

2.2.1. Stride Transport Methods

The “transport.h” header file defines the following interface to be implemented by a class in the Transport DLLs:

```
/**************************************************************************
* @class IStrideTransport
* Abstract class interface to be implemented by transport libraries.
**************************************************************************/
public:
    enum ConnectStatus {
        ConnectStatusDisconnected = 0,
        ConnectStatusConnected
    };
    typedef std::vector<std::wstring> ErrorList;
    typedef std::vector<IStrideTransportProperty*> PropertyList;

    virtual       bool           SendData(const unsigned char* data, long size) = 0;
    virtual       ErrorList      ValidateProperties() = 0;
    virtual       bool           Connect() = 0;
    virtual       bool           Disconnect() = 0;
    virtual       void           AddListener(IStrideTransportListener* pListener);
    virtual       void           RemoveListener(IStrideTransportListener* pListener);
    virtual const std::wstring & Name() const ;
    virtual       ConnectStatus  Status() const ;
    virtual const PropertyList & Properties() const ;
    virtual       bool           ReturnData(const unsigned char* data, long size)size);
    virtual       bool           DumpData(
        const unsigned char* data,
        long size,
        IStrideTransportListener::DumpType type);
    virtual       bool           QueueData(
        const unsigned char* data,
        long size,
        IStrideTransportListener::DumpType type);
    virtual       bool           DumpQueue(IStrideTransportListener::DumpType type);
    virtual void SendEvent(
        const std::wstring & message,
        const IStrideTransportListener::EventType & type,
        const IStrideTransportListener::EventLevel & level);

protected:
    typedef std::set<IStrideTransportListener*> ListenerList;

    IStrideTransport(const std::wstring & name) :
         m_Name(name),
```

```
        m_Status(ConnectStatusDisconnected);
    virtual ~IStrideTransport();
    virtual       bool          ConnectNotify();
    virtual       bool          DisconnectNotify();

    PropertyList      m_SupportedProperties;
    ListenerList      m_Listeners;
    ConnectStatus     m_Status;
    std::ostringstream   m_ReadDumpQueue;
    std::ostringstream   m_SendDumpQueue;
private:
    std::wstring m_Name;
};
```

2.2.1.1. Connect

Establish a connection

Prototype

```
bool Connect();
```

Description

The **Connect()** method is called to establish the connection for the transport. This often involves checking the current connection properties and opening the physical devices required for the connection. This method returns a bool status to indicate whether the connection was successfully started.

Parameters	Type	Description
None		

Return Type	Values	Description
bool	true	Connection established
	false	Connection attempt failed.

2.2.1.2. Disconnect

Terminate a connection

Prototype

```
bool Disconnect();
```

Description

The **Disconnect()** method is called to terminate the connection for the transport. This typically involves closing any physical devices used for the connection and freeing resources. This method returns a bool status to indicate whether the disconnection request was successful.

Parameters	Type	Description
None		

Return Type	Values	Description
bool	true	Disconnect succeeded
	false	Error encountered during disconnect

2.2.1.3. SendData

Send data from host to target

Prototype

```
bool SendData(const unsigned char* data, long size);
```

Description

The **SendData()** method is called by the Transport Server to send data from the Host Runtime to the target device. This method returns a bool status to indicate whether the data transfer was successful. This method must be implemented by each Transport DLL and it is only called by the Transport Server. If the Transport DLL is unable to send the data, it should publish an error event to the listeners (using SendEvent or one of the S2TP_ macros) and return false. The Transport Server will log any failed calls to SendData, but it will not attempt to resend the data. If retries are appropriate for a given transport, the Transport DLL must implement the retry logic in its SendData method.

Parameters	Type	Description
data	Input	Data buffer to send via the transport. This buffer is owned by the caller and should not be freed by the Transport DLL.
size	Input	Size of data in bytes.

Return Type	Values	Description
bool	true	Data sent
	false	Data submission failed

2.2.1.4. ReturnData

Return data from target to host

Prototype

```
bool ReturnData(const unsigned char* data, long size);
```

Description

The **ReturnData()** method is called by the Transport DLL to feed data from the target device into the STRIDE Runtime (via the Transport Server). This method returns a bool status to indicate whether the data was successfully submitted. The default implementation that exists in the IStrideTransport base class should be sufficient for most transports. The Transport DLL implementer must arrange to read incoming data from the connection (typically in a background thread) and then call ReturnData for all data that is received. This method requires complete STRIDE messages, thus framing (e.g. SLAP) is typically required to guarantee that complete STRIDE messages are give to this method.

Parameters	Type	Description
data	Input	Data buffer to submit
size	Input	Size of data in data, in bytes

Return Type	Values	Description
bool	true	Data submitted
	false	Data submission failed.

2.2.1.5. ValidateProperties

Validate the current transport property values.

Prototype

```
ErrorList ValidateProperties();
```

Description

The **ValidateProperties()** method is called by clients of the TransportServer to verify that the current property values are legitimate. The Transport implementation should check the current property settings for valid ranges and values as appropriate for the particular transport. This method returns an ErrorList type (vector of strings) that contains error information about invalid properties. If all the properties have valid values, this method should return an empty ErrorList.

Parameters	Type	Description
None		

Return Type	Values	Description
ErrorList	empty	All properties are valid.
	Non-empty	One more properties are invalid. The Elements of ErrorList contain specific information about each validation failure.

2.2.2. Transport Global Functions

2.2.2.1. getTransport

Create a singleton of the Transport Class object.

Prototype

```
IStrideTransport* getTransport();
```

Description

The **getTransport()** function is called by the Transport Server to get an instance of the class that implements the IStrideTransport interface in the Transport DLL. By convention, this method should create a new instance of the class upon first call and should return the same instance (singleton) with each subsequent call. The transport can free that instance when **cleanupTransport** is called.

Parameters	Type	Description
None		

Return Type	Values	Description
IStrideTransport*	Valid object	The object instance has been created
	NULL	There was an error creating the Stride Transport object.

2.2.2.2. cleanupTransport

Allow the transport DLL to free singleton instance.

Prototype

```
void cleanupTransport();
```

Description

The **cleanupTransport()** function is called by the Transport Server when it is no longer actively using that transport DLL. By convention, this method should free the singleton instance of the Stride Transport object that was created by getInstance. It can also free any additional resources that might have been allocated by getInstance.

Parameters	Type	Description
None		

Return Type	Values	Description
None		

2.2.2.3. getAPIVersion

Returns the Stride Transport API version of the Transport DLL.

Prototype

```
long getAPIVersion();
```

Description

The **getAPIVersion()** function is called by the Transport Server to get the TRANSPORT_API_VERSION value for which the Transport DLL was compiled. The Transport Server reads this value to make sure it is compiled against the same version of "transport.h" as the Transport DLL is.

Parameters	Type	Description
None		

Return Type	Values	Description
long	1	The first version of "transport.h". This is the only value that is currently supported.

2.2.2.4. getTransportVersion

Returns the Transport DLL version.

Prototype

```
long getTransportVersion();
```

Description

The **getTransportVersion()** function is called by the Transport Server to get a Transport specific version value. The Transport Server does not current use this value but might choose to in the future. As such, this value is currently determined by the Transport DLL author and has no direct impact on how the transport is loaded or used.

Parameters	Type	Description
None		

Return Type	Values	Description

long	Any	The current version of the transport, as determined by the author.

2.3. Building a Host Transport Services DLL

Developers can create custom Transport DLLs as necessary to implement target communication using protocols other than basic serial and TCP/IP. The following outlines the steps required to create a custom Transport DLL.

1. Create a new C++ Win32 DLL using Microsoft Visual Studio (express editions are fine).
2. Add STRIDE_DIR\inc to the project include path.
3. If using SLAP framing, add SLAP header and source files to the project.
4. Add STRIDE_TRANSPORT preprocessor definition.
5. Specify the Multithread DLL version of the MSVC runtime.
6. Declare a class that inherits from IStrideTransport.
7. Make sure that your class constructor calls the IStrideTransport constructor with a single wide-string argument – the name you want to give to your transport. To avoid confusion, we recommend appending "(debug)" to the name for debug conifigurations (this allows you to use both debug and release builds of the transport in the same transport server).
8. Setup your transports properties in the constructor as well.
9. Declare and implement the Connect method.
10. Declare and implement the Disconnect method.
11. Declare and implement the SendData method.
12. Declare and implement the ValidateProperties method.
13. Create a background thread to receive incoming data from the target. This thread should call the ReturnData method (already implemented by the IStrideTransport base class)
14. Set the project output to the STRIDE_DIR\transports directory so the DLL will be found by the transport server.
15. Build the project and verify that it is loaded by the Transport Server (you can use the Studio connection settings dialog for this).

2.3.1. Required Naming Convention

A Host Transport DLL must follow this naming convention (where <name> is selected by the transport author to insure a unique file name):

transport<name>.dll

2.3.2. Saving Settings

The Transport DLLs support get and set operations for all properties that they expose via the Properties( ) method. The transport server does not persist any property state for transports, but STRIDE Studio does. Users can select property settings and the active transport to use for connections in Studio. These settings are saved across invocations of Studio and are only valid when connecting using STRIDE Studio.

2.4. Existing DLLs

Several Transport DLLs are available with the standard STRIDE host installation. Transport DLLs are installed into <STRIDE_DIR>\transports directory.

2.4.1. transportRS232.dll

This transport uses a standard serial port with data framed by SLAP. It supports the standard COM port data settings via its properties (baud rate, data bits, etc.).

2.4.2. transportTCP.dll

This transport is used to connect to a target over a TCP connection. This DLL uses SLAP to frame the data before sending it, and decodes a SLAP frame on the receive side. The transport tries to establish a TCP/IP client connection to a listening device. The device address (or DNS name) and TCP port number are configurable properties of the transport.