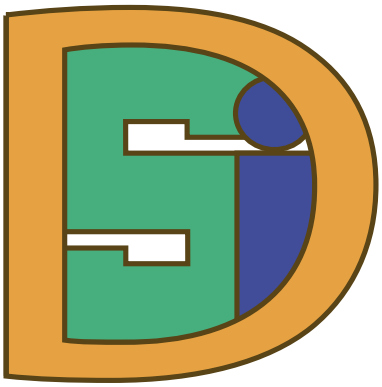

DSI API

Version 0.2

Harman/Becker Automotive Systems GmbH

Draft

DSI API: Version 0.2

Harman/Becker Automotive Systems GmbH

Becker-Göring-Straße 16, 76307 Karlsbad, Germany

Author:
Martin Häfner Martin.Haefner@harman.com
Florentin Picioroaga Florentin.Picioroaga@harman.com

Person in charge:
Martin Häfner Martin.Haefner@harman.com
Publication date 19.03.2012

Draft

Revision History

Revision 0.1	February 14th, 2012	MHaefner
Initial version		
Revision 0.2	March, 2012	FPicioroaga
Arranging document format to conform with the other documents in the DSI specification.		

Table of Contents

List of Figures

List of Tables

1 Introduction

DSI is an acronym for Distributed Service Interface, a message oriented middleware designed for embedded systems in the automotive industry. It consists of an interface resolution and lifecycle service executable, the so-called servicebroker, a C++ base library providing communcation primitives and base classes and a Java based DSI code generator.

The DSI base library only depends on the standard C++ library. It is intended to be compiled for a Linux-compliant hard- and software platform, i.e. Linux ARM or x86 using the GNU C++ compiler collection with a version greator or equal to 4.4. It makes use of classes from the current standard extension TR1.

This document describes the basic steps necessary to set up DSI client and server applications. The code generator generates C++ base classes for both the client implementation (also known as proxy) and the server (the stub) from a service interface description file (`.hbsi`). The developer may then implement the request and response handlers from these generated skeletons.

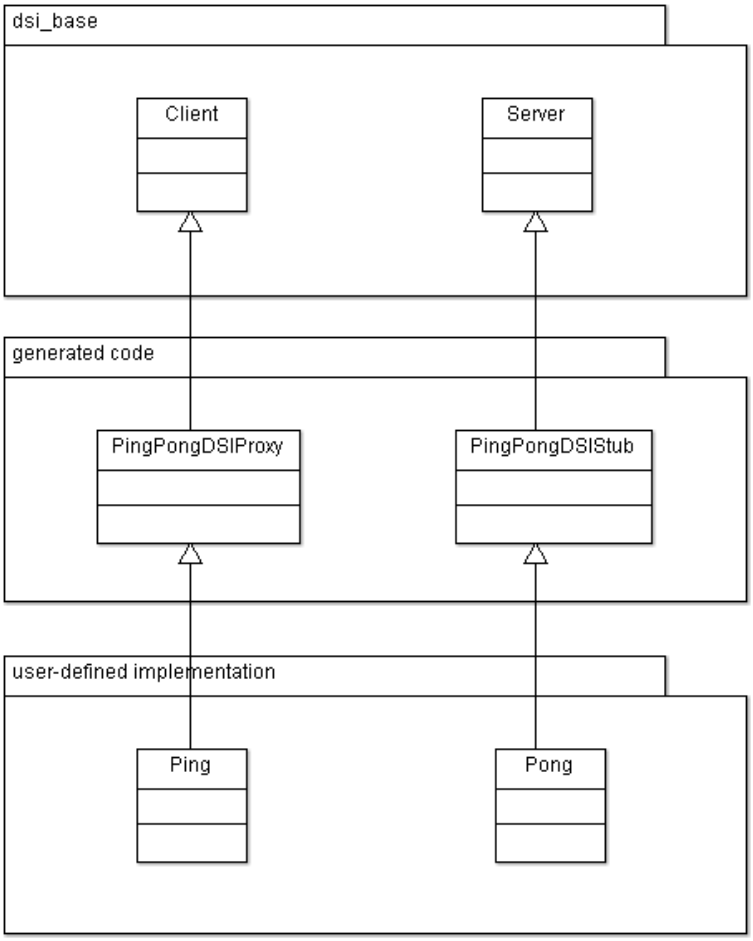

Figure 1.1. Static and generated code

2 Build setup

The very first step to build a DSI application is to generate code from an interface definition file using the code generator located in the java archive `dsi2gen.jar` which is based on Java 5. The code generator accepts an `.hbsi` input file and generates C++ headers and source files for

- the interface definition itself including the definition of all data types defined in the `.hbsi`
- client and server skeletons, also known as proxy and stub
- streaming functions for user-defined data types

The code generator has to be invoked according to:

```
java -jar dsi2gen.jar <hbsi-input-file> <output-directory>
```

The DSI library source code is currently build with `cmake` using `GNU make` as backend. Therefore, the source package contains a macro that can be invoked for generating an archive with the above described code fragments from an interface definition file. The `cmake` macro will be installed during the installation process of the DSI source package - via `make install` - and will be located in the directory `<CMAKE_INSTALL_PREFIX>/share/dsi`, e.g. `/usr/local/share/dsi` and can be referenced from there.

As example, the following `CMakelists.txt` will create a ping application linked against an `libPingPong.a`.

```
ADD_EXECUTABLE(ping ping.cpp)
    TARGET_LINK_LIBRARIES(ping PingPong dsi_base dsi_common dsi_servicebroker
rt pthread)

    DSI2_GENERATE(PingPong.hbsi)
```

The makefile snippit also shows which libraries have to be linked for a complete DSI application, no matter if client or server:

- **`dsi_base`** implementing the DSI-specific base classes
- **`dsi_common`** implementing generic utility classes not provided by the standard library
- **`dsi_servicebroker`** implementing the communication interface to the servicebroker
- **`rt`** standard library for Linux realtime extensions
- **`pthread`** standard library for enabling Linux threading support

3 Compile time

3.1 DSI Proxy

The proxy is the client of the DSI service. The proxy is always derived from the generated DSIProxy class (in our example this would be `CPingPongDSIProxy`).

- For each request in the interface, the proxy has a `requestABC(...)` method. ABC is the name of the method.
- For each response that might return from the server, the proxy has a `responseABC(...)` method the proxy can overwrite.
- For each attribute in the service, the proxy has a `notifyOnABC()` which sets a notification.
- For each attribute in the service, the proxy has an `onABCUpdate(...)` method that is called when the attribute changed its value.

The proxy does not have to implement all the response methods. It can implement that one it is interested in. Here an example that shows how to implement a proxy:

```
#include "dsi/CCommEngine.hpp"
#include "CPingPongDSIProxy.hpp"

class CPing : public CPingPongDSIProxy
{
   public:
   CPing()
   : CPingPongDSIProxy("pong")
   {
      // NOOP
   }

   void componentConnected()
   {
      // the service has connected
      requestPing(L"Hallo Welt");
   }

   void componentDisconnected()
   {
      // the service has disconnected
   }

   void responsePong(const std::wstring&amp;amp; message)
   {
      notifyOnStringAttribute(); // set a notification on StringAttribute
   }

   void onStringAttributeUpdate(const std::wstring&amp;amp; StringAttribute,
DSI::DataStateType)
   {
      // the attribute StringAttribute has an initial or new value
      assert( StringAttribute == "Hello World!" );
   }
};
```

```
int main()
{
   DSI::CCommEngine engine;

   CPing proxy;
   engine.add(proxy);

   return engine.run();
}
```

3.2 DSI Stub

The DSI stub is the implementation of the service, in other words the server implementation. The stub is always derived from the generated DSIStub class (in our example this would be `CPingPongDSIStub`). All request methods in the stub are pure virtual. So the derived class must implement them.

- For each request in the interface, the stub has a pure virtual method named `requestABC(...)`. ABC is the name of the request.
- For each response in the interface, the stub has a method called `responseABC(...)`.

```
#include "dsi/CCommEngine.hpp"
#include "CPingPongDSIStub.hpp"


class CPong : public CPingPongDSIStub
{
   public:
   CPong()
   : CPingPongDSIStub( "pong" )
   {
      setStringAttribute("Hello World!");
   }

   void requestPing(const std::wstring&amp;amp; message)
   {
      responsePong(message);
   }
};


int main()
{
   DSI::CCommEngine engine;

   CPong stub;
   engine.add(stub);

   return engine.run();
}
```

As stated above, the communication engine implements a multiplexing algorithm to serve multiple proxies and/or stubs at one time. Therefore, proxy or stub implemen-

tations have to keep the communication engine reactive which means that blocking requests should be avoided within this thread context and should be handled by other threads.

3.3 Communication Engine

DSI proxies and stubs communicate through the underlying transport with each other, typically local or TCP/IP sockets. The communication engine will act as a muliplexer for incoming requests or responses and also helps duplexing the DSI packages via socket reuse for resource efficiency reasons. Therefore, clients and servers always belong to a communication engine where they have to be registered on.

```
int main()
{
   DSI::CCommEngine engine;

   // add the proxy to the engine
   CPingPongDSIProxyImpl proxy("pong");
   engine.add(proxy);

   // add the stub to the engine
   CPingPongDSIStubImpl stub("pong");
   engine.add(stub);

   return engine.run();   // enter the event loop
}
```

In this example the `CCommEngine` object is being created, and the proxy and stub are being added to it. In this case, both the proxy and stub would run in the same application. You can add an arbitrary amount of proxies and stubs to an engine. The engine automatically enables TCP/IP transport if a TCP/IP enabled stub was added.

`engine.run()` runs the main event loop. This call will first return if `stop()` is called on the engine from any callback handler or any other thread.

4 Runtime

4.1 DSI protocol tracing

The DSI library supports the registration of a callback hook in order to allow the tracing of all outgoing and incoming DSI requests. A trace handler has to implement the C++ interface `IChannel` which resides in the namespace `DSI::Trace`. The interface must provide the following callback functions:

```
/**
* Open an output stream for the given interface.
*
* @return -1 if opening a stream for the handle does not work or any positive
integer
*         in case of success.
*/
virtual int open(const SFNDInterfaceDescription&amp;amp; iface, Direction dir,
uint32_t updateId = DSI::INVALID_ID) = 0;

/**
* Close the outputstream as described by @c handle.
*/
virtual void close(int handle) = 0;

/**
* @return true if input streaming is enabled for the correspondant
*         interface and direction for which the handle was opened, else false.
*/
virtual bool isActive(int handle) = 0;

/**
* @return true if payload streaming is enabled for the correspondant
*         interface, else false.
*/
virtual bool isPayloadEnabled(int handle) = 0;

/**
* Write a complete DSI frame. The @c info pointer may be null in case
* of non-data (dis-/connect request) or further-data requests (in case of large
DSI requests).
*/
virtual void write(int handle, const DSI::MessageHeader* hdr,
const DSI::EventInfo* info, const void* payload, size_t len) = 0;
```

A simple console based trace handler implements the tracing interface by writing all DSI requests and responses received or sent via the transport layer to `stdout`. The tracing subsystem has to be initialized by registration of a trace handler instance, as done in the following example:

```
#include "dsi/CStdoutTracer.hpp"

int main()
{
   DSI::Trace::CStdoutTracer tracer;
   DSI::Trace::init(tracer);

   DSI::CCommEngine engine;
```

```
   ...
}
```

This will e.g. for the delivered pong application generate a console output in the following form:

```
mhaefner@ubuntu:/export/dsi/build/debug/examples/pingpong$ ./pong

      ==> DSI v4.0 ConnectRequest
      serverId     : 500016.1000
      clientId     : 100013.1000
      interface    : ping.PingPong:0.2

      ==> DSI v4.0 DataRequest
      serverId     : 500016.1000
      clientId     : 100013.1000
      interface    : ping.PingPong:0.2
      requestType  : REQUEST
      requestId    : 0
      sequenceNr   : 2

      <== DSI v4.0 DataResponse
      serverId     : 500016.1000
      clientId     : 100013.1000
      interface    : ping.PingPong:0.2
      responseType : RESULT_OK
      requestId    : 2147483648
      sequenceNr   : 2

      ==> DSI v4.0 DisconnectRequest
      serverId     : 500016.1000
      clientId     : 100013.1000
      interface    : ping.PingPong:0.2

      mhaefner@ubuntu:/export/dsi/build/debug/examples/pingpong$
```

==> shows incoming requests or responses while <== shows the outgoing equivalents. The current implementation of the `CStdoutTracer` does not support payload deserialization though payload data in general may be transferred and displayed as well.

4.2 Logging subsystem

In embedded environments it is sometimes not possible to use a general purpose debugger like gdb for application debugging on the target. In these cases, the standard mechanism to log the application flow is by adding debugging statements, e.g. printf, to the code. DSI supports a set of macros that is used within the library code. These macros can be redefined by the end-user compiling the base library to fit his logging architecture. The default implementation supports logging to stdout/ stderr or to the syslog facility.

The logging subsystem within an DSI application context can be adjusted with the two functions `Log::setLevel(int)` and `Log::setDevice(int)` where valid values for level are in the range 0 (critical errors) to 4 (debugging information). The default level is `Critical` and the default device is set to `SystemLog`, that means only severe errors are logged into the syslog be default. See `include/Log.hpp` for distinct information.

The following macros are available and though could be redefined:

Macro name	Description
TRC_SCOPE_DEF(a,b,c)	Definition of a global trace scope by a 3-tuple of C++ identifiers. Typically the 3-tuple refers to the current context, e.g. TRC_SCOPE_DEF(dsi, CCommEngine, run).
TRC_SCOPE(a,b,c)	RAII object that traces the lifetime of the current trace context via entry and exit messages. It may refer to a previously defined scope definition.
DBG_MSG((msg))	This macro checks for the appropriate logging level and typically prints informational messages.
DBG_WARNING((msg))	This macro checks for the appropriate logging level and typically prints warning messages.
DBG_ERROR((msg))	This macro checks for the appropriate logging level and typically prints error messages.

Table 4.1. Logging macros

4.3 Threading

DSI proxies and stubs cannot be considered to be reentrant. Therefore, it is not safe to call any methods on stubs or proxies from within other threads the communication engine is not running in since proxies and stubs always belong to a certain communication engine. Crossing thread boundaries can only be done by using some kind of event-queueing mechanism, e.g. a message queue, socket or pipe or any other pollable device. You may add a pollable device to the communcation engine by calling `addGenericDevice(...)` with the file descriptor to poll on, an appropriate event handler functor and the expected data flow direction, i.e. the poll flag to be set for the file descriptor. By doing so, the event data may be transferred into the communication engine's thread context and be processed synchronously within the given event handler functor called from the communcation engine when appropriate data arrives.

4.4 Environment variables

The behaviour of DSI applications built using the DSI base classes can be controlled via multiple environment variables.

Variable name	Description
DSI_SERVICEBROKER	The server socket path of the servicebroker relative to the root mountpoint `/var/run/servicebroker`.

Variable name	Description
DSI_COMMENGINE_PORT	The IP port the communication engine should bind to in case of services requiring IPv4 transport. If this environment variable is not set, the port will be assigned by the OS kernel.
DSI_IP_ADDRESS	The IPv4 address in dotted quad notation that will be used when registerign services at the servicebroker. If not given, the servicebroker is able to dynamically resolve the correct IPv4 address in multi-node environments. Not setting this variable is typically the right choice. Note, that setting this variable does not affect the binding of server sockets to network interfaces, i.e. sockets are currently always bound to <const>INADDR_ANY</const> .
DSI_FORCE_TCP	Forces all services to be registered as TCP/IP services in addition to the registration for local transport. Moreover, services are connected via TCP/IP sockets even from local clients which could easily use local transport.
DSI_RECV_TIMEOUT	Timeout to be set on DSI transport sockets for receiving data.
DSI_SEND_TIMEOUT	Timeout to be set on DSI transport sockets for sending data.

Table 4.2. DSI related environment variables