DSI IPC Specification

Version 0.2

Harman/Becker Automotive Systems GmbH

Draft

DSI IPC Specification: Version 0.2

Harman/Becker Automotive Systems GmbH

Becker-Göring-Straße 16, 76307 Karlsbad, Germany

Author:
Marie Ries Marie.Ries@harman.com
Florentin Picioroaga Florentin.Picioroaga@harman.com

Person in charge:
Marie Ries Marie.Ries@harman.com
Publication date 19.03.2012

Draft

Revision History

Revision 0.1	February, 2012	MRies
Initial version		
Revision 0.2	March, 2012	FPicioroaga
Arranging document format to conform with the other documents in the DSI specification. Add the introduction chapter. Review documentation.		

Table of Contents

List of Figures

List of Tables

1 Introduction

This document describes the wire protocol, the serialization for the DSI communication.

As it can be observed in Figure 2.2, “DSI wire format when transporting messages” there are three main parts that are transmitted over the wire:

- The simple IPC DSI protocol data which is focused on transporting messages. Acts as a generic container for the service interface abstraction.
- Data needed to transport the concepts of the service interface abstraction.
- The user data, the payload that is transferred over the service interface.

Note

The DSI communication term used in this document comprises two aspects: the DSI IPC protocol (the transport protocol) and the service abstraction protocol.

Due to its serialization design the DSI communication can be flexibly used in combination with other IPCs.

For example the service interface abstraction data can be embedded in other IPC as the DSI IPC, e.g. service interface over DBus. In the same time the DSI IPC container can transport messages for other abstractions as the service interface.

2 IPC protocol

The DSI IPC protocol was designed based on the following requirements:

- high performance communication protocol
- scale down in terms of resource usage: CPU, RAM, code size
- save network bandwidth
- keep it simple

The DSI IPC is basically a point to point simple communication protocol that transports messages. As IPC is mainly used for exchanging control messages that are normally small it has been defined that a transmitted DSI packet has a maximal size of 4KB. As the size of a DSI message can be greater than 4 KB, a segmentation of messages in several packets might occur. This constraint is only an implementation configuration that can be changed and it is not part of the protocol itself, other values can be chosen for the maximum packet size.

In the following we will distinguish between the data of a message and the data of a packet, i.e. only a part of the message.

The following figure depicts the wire protocol when establishing IPC connections.

Note

The IPC command data field is used only for the `ConnectRequest` and `ConnectResponse` commands. Also, depending on the transport DSI runs, the IPC command data has a different meaning.

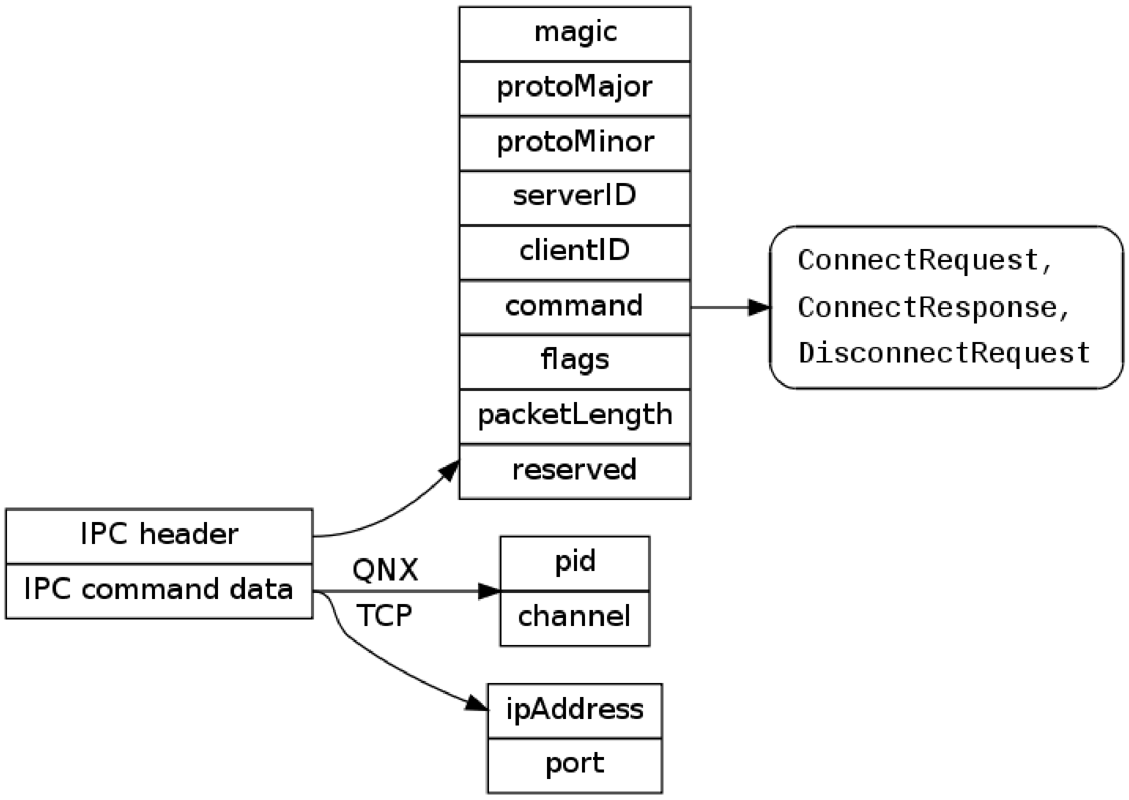

Figure 2.1. DSI wire format when managing IPC connections

2.1 DSI header

Each DSI packet starts with a DSI header:

```
struct MessageHeader
{
   /** The type of the message (unique for all messages) */
   int32_t type ❶;
   /** Protocol Major Version */
   uint16_t protoMajor ❷;
   /** Protocol Minor Version */
   uint16_t protoMinor ❸;
   /** The ID of the server which sends/receives this message. */
   union SPartyID serverID ❹;
   /** The ID of the client which sends/receives this message. */
   union SPartyID clientID ❺;
   /** The command. */
   DSICmd cmd ❻;
   /** Flasgs (bit 1 indicates if there is at least one more packet coming) */
   uint32_t flags; ❼;
   /** The length of this packet (without message header). */
   uint32_t packetLength ❽;
   /** reserved (fillup for 8 byte-alignment) */
   int32_t reserved[1]; ❾
};
```

❶ The element *type* is magic number that is unique for all messages (DSI_MESSAGE_MAGIC = 0x200).

❷ The element *protoMajor* describes the Major version number of the DSI protocol. (current value is 4)

❷ The element *protoMinor* describes the Minor version number of the DSI protocol. (current value is 0)

❹ The element *serverID* describes the ID of the server. The id is currently generated by the Servicebroker when a service registers in the DSI system.

❺ The element *clientID* describes the ID of the client. The id is currently generated by the Servicebroker when a client requests connection information from the Servicebroker.

❻ The element *cmd* indicates which kind of command is sent:

```
enum Command
{
   DataRequest=7,       ///< a data request transfer is initiated
   DataResponse=8,       ///< a data response transfer is initiated
   ConnectRequest = 9,  ///< a connection is requested
   DisconnectRequest = 10,  ///< a disconnection is requested
   ConnectResponse   = 11  ///< response to a connect request
} cmd;
```

The value of *cmd* determines the content of the message.

❼ The element *flags* indicates if there are more packets coming. Bit 1 indicates if there is at least one more packet coming.

❽ The element *packetLength* describes the length of this packet not including the length of the DSI header.

Note

The total message length is not known, the receiver only knows if there is more data to be expected for the current message over the *flags* field.

❾ The element *reserved* has no meaning. It has been introduced to achieve an 8-Byte alignment.

2.2 DSI communication build up and shutdown

2.2.1 Client connect (Cmd: ConnectRequest)

The `ConnectRequest` is sent from the client to the server to setup a new connection.

The data transferred with the `ConnectRequest` command depends on the kind of socket where the command is received.

If the `ConnectRequest` command is received on a Unix Domain Socket, the message contains the following data.

```
{
     uint32_t pid;         ///< process id in host-byte-order
     uint32_t channel;     ///< channel id (socket fd) in host-byte-
order
 }
```

❶ The pid is the process identification number of the client process
❷ The channel describes the file descriptor of the used socket.

If the `ConnectRequest` command is received on a TCP socket the message contains the following data:

```
   {
      uint32_t ipAddress;   ///< ip address in big endian byte-order
      uint32_t port;        ///< port in big endian byte-order
   };
```

❶ The `ipAddress` is the IP address of the client TCP server socket
❷ The `port` is the port number of the client TCP server socket.

2.2.2 Server connect response (Cmd: ConnectResponse)

The `ConnectResponse` is sent from the server to the client as response to a received `ConnectRequest.`

The `ConnectResponse` is sent over the same socket where the `ConnectRequest` has been received.

The data transferred with the `ConnectResponse` command depends on the kind of socket where the command `ConnectRequest` has been received.

If the `ConnectRequest` command has been received on a Unix Domain Socket, the `ConnectResponse` message contains the following data:

```
{
     uint32_t pid;         ///< process id in host-byte-order
     uint32_t channel;     ///< channel id (socket fd) in host-byte-
order
 }
```

❶ The pid is the process identification number of the server process
❷ The channel describes the file descriptor on the server

If the `ConnectRequest` command has been received on a TCP socket the `ConnectResponse` message contains the following data:

```
   {
      uint32_t ipAddress;   ///< ip address in big endian byte-order
      uint32_t port;        ///< port in big endian byte-order
   };
```

❶ The `ipAddress` is IP address of the server TCP server socket

❷ The `port` is the port number of the server TCP server socket.

2.2.3 Communication shutdown (Cmd: DisconnectRequest)

The `DisconnectRequest` is sent from the client to the server to disconnect an existing client /server connection.

Beside the DSI header no further data are transferred.

2.3 DSI service interface message

The service interface logic is using the DSI IPC communication mechanism to build high level concepts as attributes, requests, notifications that can be used in the business logic. The following figure describes the streaming format of the Harman/ Becker service interface request/response messages.

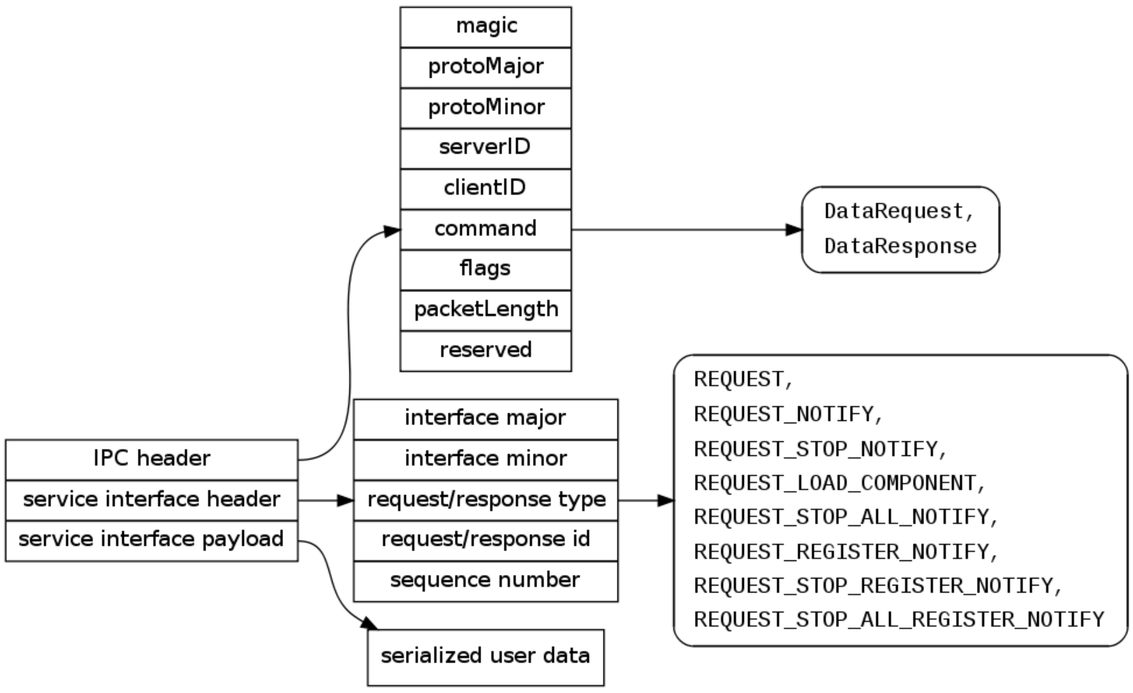

Figure 2.2. DSI wire format when transporting messages

2.3.1 Requests (Cmd: DataRequest)

All request messages contain the following data (preceded by the DSI header):

```
{
   uint16, // service interface major version
   uint16, // service interface minor version
   uint32, // request type ❶
   uint32, // request id   ❷
   int32   // sequence number ❸
}
```

❶ The following request types are currently defined (`0x00`-`0xFF` are reserved for internal use):

- 0x0100 (`REQUEST`)

  Standard request: triggers a method call on the server implementation.

- 0x0101 (`REQUEST_NOTIFY`)

  Generic request: register for notifications for a data attribute or for a response method. The attribute/response ID is sent as request ID.

- 0x0102 (`REQUEST_STOP_NOTIFY`)

  Generic request: clears any existing notification for a data attribute or for a response method. The attribute/response ID is sent as request ID.

- 0x0103 (`REQUEST_LOAD_COMPONENT`)

  Generic request: used by the HARMAN component framework to instantiate a component on demand.

- 0x0104 (`REQUEST_STOP_ALL_NOTIFY`)

  Generic request: clears all existing notifications for this client.

- 0x0105 (`REQUEST_REGISTER_NOTIFY`)

- Generic request: registers for updates of an information method. The information ID is sent as request ID. The sequence number identifies the used session.

- 0x0106 (`REQUEST_STOP_REGISTER_NOTIFY`)

  Generic request: clears specific update notification for an information method. The information ID is sent as request ID. The sequence number identifies the used session.

- 0x0107 (`REQUEST_STOP_ALL_REGISTER_NOTIFY`)

  Generic request: clears all existing update notification for this client and this session. The sequence number identifies the used session.

❷ The request ID is interface specific and identifies an attribute, request or response/information. The allowed kind of ID depends on the request type

- `REQUEST`

  The request ID identifies a request. Attribute or response/information IDs are not allowed.

- `REQUEST_NOTIFY`

  The request ID identifies the attribute, request or response/information to notify on.

- `REQUEST_STOP_NOTIFY`

  The request ID identifies the attribute, request or response/information to stop being notified on.

- `REQUEST_LOAD_COMPONENT`

  The request ID is ignored.

- `REQUEST_STOP_ALL_NOTIFY`

  The request ID is ignored.

- `REQUEST_REGISTER_NOTIFY`

  The request ID identifies the information. Attribute,request or response IDs are not allowed.

- `REQUEST_STOP_REGISTER_NOTIFY`

  The request ID identifies the information. Attribute,request or response IDs are not allowed.

- 0x0107 (`REQUEST_STOP_ALL_REGISTER_NOTIFY`)

  The request ID is ignored.

❸ The sequence number is used for session handling and/or to assign a response to a request. For one client the sequence number must be unique.

- `REQUEST`:

  if a response is attached the the request, the sequence number must be returned in the response message.

  If no response is attached the sequence number is ignored.

- `REQUEST_NOTIFY`:

  The sequence number is ignored.

- `REQUEST_STOP_NOTIFY`:

  The sequence number is ignored.

- `REQUEST_LOAD_COMPONENT`:

  The sequence number is ignored.

- `REQUEST_STOP_ALL_NOTIFY`:

  The sequence number is ignored.

- `REQUEST_REGISTER_NOTIFY`:

  The sequence number identifies the session.

- `REQUEST_STOP_REGISTER_NOTIFY`:

  The sequence number identifies the session.

- 0x0107 (`REQUEST_STOP_ALL_REGISTER_NOTIFY`)

  The sequence number identifies the session.

If the request has parameters, they are streamed as described below and will directly follow the generic request data.

2.3.2 Responses, informations and attributes (Cmd: DataResponse)

All response messages contain the following data (preceded by the DSI header):

```
{
   uint16, // service interface major version
   uint16, // service interface minor version
   uint32, // response type ❶
   uint32, // response id   ❷
   int32   // sequence number  ❸
}
```

❶ The following response types are currently defined (`0x00`-`0xFF` are reserved for internal use):

- 0x0200 (RESULT_OK)

  The response method of a request has been called, the `error` method wasn't called. Response parameters follow.

  The message is also used for informations.

- 0x0201 (RESULT_INVALID)

  The `error` method was called with a result method ID as argument. An int32 error code follows.

- 0x0202 (RESULT_DATA_OK)

  A data attribute has been updated, the new value follows.

- 0x0203 (RESULT_DATA_INVALID)

  A data attribute has been marked as invalid, the old value is preserved – only the status changes. An int32 error code follows.

- 0x0204 (RESULT_REQUEST_ERROR)

The `error` method was called with a request method ID as argument. An int32 error code follows.

- 0x0205 (RESULT_REQUEST_BUSY)

  A call to this request has already been received, but the response has not yet been made. The server is busy working on the same request. No data follows.

❷ The response ID is interface specific and identifies an attribute, request or response/information. The allowed kind of ID depends on the response type.
❸ The sequence number is used for session handling and/or to assign a response to a request.

If the response has parameters, they are streamed as described above and will directly follow the generic response data.

If error codes are not provided for the service interface in use, 0x7FFFFFFF (ERROR_CODE_UNDEFINED) is used as error code value.

2.3.3 Service interface ID handling

This section describes how the generator assigns IDs to methods/attributes based on the information found in an `*.hbsi` file. The following ranges have been defined for update IDs:

- `0x00000000u`-`0x7FFFFFFFu`: For request IDs
- `0x80000000u`-`0xBFFFFFFFu`: For response/information IDs
- `0xC0000000u`-`0xFFFFFFFFu`: For attribute IDs

The service interface editor uses increasing IDs for methods and attributes. There may be gaps, but newer methods always have higher IDs so that new elements will never reuse a previously used ID. The service interface generator sorts the methods by their IDs in ascending order and then assigns IDs based on the ranges described above. And here an example:

```
Elements defined in an .hbsi file:

requestA: ID=4
requestB: ID=2
responseA: ID=23
informationA: ID=24
attributeA: ID=12
attributeB: ID=13

Generated IDs:

0x00000000u: requestB
0x00000001u: requestA
0x80000000u: responseA
0x80000001u: informationA
0xC0000000u: attributeA
0xC0000001u: attributeB
```

3 Data serialization

This section describes how the actual data is serialized and deserialized. All alignment values in this section are relative to the beginning of the data section of the buffer. The data section must begin on a 8-byte alignment in memory.

3.1 Primitive data types

All data types are transmitted in little endian byte order. In order to allow primitive data types to be moved into and out of the data stream with instructions specifically designed for these types, all data types must be aligned to their natural boundaries. The following tables show the sizes and alignments of the primitive data types. If necessary, padding bytes must be inserted into the stream so that the next primitive data type hits the correct boundary. The value of the padding byte is not defined.

3.1.1 Integer Data Types

Type	Size [bytes]	Alignment [bytes]
byte/char	1	1
signed/unsigned short	2	2
signed/unsigned int	4	4
signed/unsigned long long	8	8

Table 3.1. Size and alignment requirements for integer data types

3.1.2 Floating Point Types

The format of floating point data types (float and double) follow the IEEE standard formats for floating point numbers.

Type	Size [bytes]	Alignment [bytes]
float	4	4
double	8	8

Table 3.2. Size and alignment requirements for floating point data types

3.1.3 Boolean Type

Boolean data is encoded as a single integer value. For the boolean value `false`, the integer value is set to `0`. For a boolean value `true`, the integer value is set to `1`.

3.1.4 Enumeration Type

An enumeration value is encoded as a single integer value. The enumeration value is simply mapped to the integer value.

3.2 String Type

A string is encoded as an unsigned integer indicating the length of the string in bytes including the NULL termination. An empty string has the length `1` followed by the NULL termination. A NULL string has the length `0`. No NULL termination is written to the stream for a NULL string.

`length`	`byte[0]`	`byte[1]`	...	`byte[length-1]`	`'\0'`

Table 3.3. Streaming schema for type String

3.3 Array Type

An array is encoded as an unsigned integer indicating the number of elements of the array followed by a sequence of elements of arbitrary types. All elements in the sequence must be of the same type.

`length`	0...n padding bytes	`element[0]`	`element[1]`	...	`element[length-1]`

Table 3.4. Streaming schema for array types

3.4 Compound types (struct)

There is no extra serialization information for structures in the DSI protocol. The members of a struct are encoded in the order they appear in the struct declaration as they were a sequence of elements of primitive data types, strings or arrays. If the alignment and data sizes of the underlaying hardware match the DSI protocol specification and a struct only consists of primitive data types, a struct could be written to the output stream using a single memcpy for performance reasons.

3.5 Variant

A variant is encoded by an unsigned integer (type id) indicating the type the variant is holding at the moment followed by the data of the current type. The type id is a number starting with 1 for the first type in the variant. The second type has the type id 2, and so on.

3.6 Argument lists

The arguments are encoded in the order they appear in the request/response/information declaration from left to right. The argument list for e.g. `requestA(int32,bool,Array<String>,int64)` is encoded as:

- encode int32
- encode bool
- encode Array<String>
- encode int64

Each argument is encoded with its specific alignment.