SDDS

Rapid Tool For Binary Communication

© ByteFerry <byteferry@qq.com>

What is SDDS？

- Data Structure
  - SDDS is short word of "stream data dynamic structure".
- DSL
  - SDDS is a DSL (domain-specific language) .
  - The schema of SDDS is human-readable and machine and machine-readable.
- Parse Engine
  - SDDS is a parse engine coded according the SDDS Specification. You can rapid complete the developing of the binary communication program.

Why SDDS

- Simple
  - According the SDDS Specification, the schema of the communication is defined using JSON. So it is very simple.
- Rapid
  - With SDDS JSON schema, it is no longer required that writing the encode and decode program.
  - Ordinary you project finished in months without SDDS. After using SDDS the project will finished in weeks.
- Extensible
  - You need not modify the program code when the require changed, you just only modify the schema which is defined in JSON.
- Open source and free
  - SDDS is a project of open source with MIT license. So you could using it free.

Benefits of SDDS

- Using the mode of core and extensions
  - Support almost primitive data types all。
  - And the custom data type could be added in the extensions.
- Just need call one function
  - Just need call decode function with argument byte_buffer while decoding and you will get the data with key-value format.
  - Just need call encode function with argument key-value format while encoding and you will get the data with byte buffer .
- None Compilation
  - Unlike other DSL of binary stream, the compilation is not required in SDDS. With SDDS the protocol parse is completed via common classes. So you only need to do: write the schema, and add some function in the extension class. That is all.

Simple example

```
"body_length": {
   "id": "body_length",
   "name": "body_length",
   "position": 4,
   "type": "int32",
   "unsigned": true
}
```

- Reading the body_length with data type unsigned int32 from the position byte 4.

Primitive data type supported by SDDS

- Integer

name	Count of byte	Count of bit	Alias
int8	1	8	char, byte
int16	2	16	short, word
int24	3	24	
int32	4	32	int, dword
int64	8	64	long

  - Sign:
    - Both signed and unsigned are supported.
  - Endian
    - Both big-endian and little-endian are supported.
  - Alias
    - All Alias of the data type could be used in the schema of SDDS. The engine of SDDS could know the alias without any problem.

Primitive data type supported by SDDS

- Float:

name	Byte count	Bit count	Alias
float32	4	8	single
float64	8	16	double

  - Alias
    - All of the alias could be used in the SDDS schema, and could be parsed by engine.
  - 64 bit support
    - The 64-bit environment is required when we use the data type Float64 and int64 .

Primitive data type supported by SDDS

- Other data type:

name	comment
bytes	Read or write the count of bytes.
bit	Read or write bit.
bits	Read or write the count of bits.
string	Read or write string
bcd8421	Read or write BCD data with 8421 mode.
hex	Read or write hex string.

  - The manner of operations
    - The operations of read, write, insert, replace are supported by SDDS engine.

Quick tutorial of SDDS

- Summary：
  - This is a quick tutorial for creating a schema step by step with a requirement of device monitor.
- Requirement:
  - Monitor every device with data of temperature, humidity, PH-value, speed.
  - And we must could start and stop the device.

Step 1: Planning the message id

- Sample of structure of message id

From	Type	Action
0x10 : server 0x20: terminal	0x01 command 0x02 status query 0x03 setting 0x04 history data	0x01 request 0x02 response

Step 2: Copy the root of SDDS Schema

```
{
  "meta": {
    "comment":"some comment",
    "version":"your version",
    },
  "options": {
    "endianness":2,
    "min_length":15,
    "debug":true,
    "top_node":"message"
    },
  "sdds": {
    "decode":{
        "message"{
        }
     },
    "encode":{
        "message"{
        }
     }
   }
}
```

- The data of the node meta is only for human reading.
- Endian is setting with the values:
  - 0: system endian
  - 1: little endian
  - 2: big endian
- min_length is used to allocate the memory.

Step 3: exit the message node

```
"message":{
        "name": "message",
        "position":o,
        "byte_fields": {
                "stx":{},
                "head":{},
                "body":{},
                "tail":{}
                "etx":{},
        }
}
```

- In the message node, we must list the structure of the first level using the attribute byte_fields.

Step 4: edit the stx node

```
"stx":{
   "comment": “start flag"
   "type":"int8",
   “unsigned”:true,
   "length":1,
   "position":0,
   "value":125
}
“etx":{
   "comment": “end flag"
   "type":"int8",
   “unsigned”:true,
   "length":1,
   "position":0,
   "value":125
}
```

- The stx is the short of Start flag.
- In the node there is a 6 attributes:
  - comment：comment of the node.
  - type：data type
  - unsigned：if the number is unsigned.
  - length：data length
  - position： position for reading or writing.
  - value: value of data.
- Attribute value is used for store the result of reading or store the value for writing.
- We could complete the etx node because they are same.

Step 5: edit head node

```
"head":{
   "id":"message_header",
   "type":"message_header",
   "position":3,
}
```

- Because there is a structure in the head node, we use the custom node message_header for link to the node named message_header .
- And there is a attribute id in the node. With the id we could found the node by path.

Step 6: edit message_header node

```
"message_header":{
   "name":"header",
   "id":"header",
   "length":11,
   "position":3,
   "byte_fields":{
      "message_id":{},
      "flow_id":{},
      "body_property":{},
      "time":{}
   }
}
```

- There are 4 fields in the message header.
  - time, data type is bcd8421 ,
  - message_id, data type is int24,
  - flow_id, data type is int16,
- The data structure of the nodes is same as the node stx.
  - body_property data type is bits.
- We use the body_property_detail as the cumtom data type.

Step 7: edit body_property_detail node

```
"body_property_detail":{
   "id":"body_property_detail",
   "type":"bytes",
   "length":2,
   "position":3,
   "bit_fields":{
      "body_length":{},
      “message_type":{},
      " crypt ":{},
   }
}
```

- There is a bit_fields structure in the node.
- And in the branch we could define the children.
- Remark: in the node with bit_fields, the attributes length and position are required.

Step 8: complete body_length node

```
"body_length":{
   "id":"body_length",
   "type":"int",
   "length":10,
   "position":0,
   "selector": "id"
}
```

- body_length is a node of bitwise operation. In the node there are the data types:
- bit, bits and int.
- And we also need to define the length of data.
- If the node is in the encode branch, we need define a selector for later access.
- So we choose the id selector.

Step 9: Complete body node(1)

- The message node is the main of the message, so the data of the node would be deference according the message type.
- So we must define a custom type to link a new node. We use message_body.
- And   we use the one_of branch to define the choose flow.
-

Step 9，Complete body node(2)

```
"mssage_body": {
   "id": "mssage_body",
   "one_of": {
      "key": {
         "comment": "",
         "name": "message_key",
         "value": "#message_type",
         "format": "0x%04X",
         "id": "message_key"
      },
      "list": {
        "0x200201":{},
        "0x200402":{}
   }
}
```

- With one_of node, we tell the SDDS parser, with the value of message_key to select from the list.
- For we could read clearly, we use the hex value of message_type.
- So we define a format to formating the message_type.

Passing the Arguments： Selector

- There are 4 selector in SDDS:
  - “#”： id selector
  - "@"： path selector
  - "$"： index selector
  - "*"： function selector
- The selector is defined in the attribute value. For example:
  - "value": "#message_type“
- This means that the value is read from the node with id of message_type.

Selector： "#" ID Selector

- "#" id Selector
  - Retrieve the node via id.
  - To operate a node with id, we need define the attribute Selector in the node. The value of Selector is "id".
  - And give the attribute id value: "id":"body_length"
  - Then we use the string link start with "#". For instance:
  - "value":"#body_length"

  - Note: Id must unique in the SDDS schema. If the id is not unique we must use other selector.

Selector： "@" Path Selector

- "@": path selector
  - path selector is a id strings split by "." . For instance：
  - "value":"@body.message_body.body_length",
  - The program will search up to the node where the branch start and the search according the path to the target. So we only need the relative path to the node where the branch start.
  - We recommend to use short path. If path is to long, the best way is to use id selector.
  - If you could not to know the path, you could to use "trace":"path" to view the path of node.
  - Because the path of the repeat node is not unique, so you must use the index selector.

Selector： "$" Index Selector

- "$" index selector
- index selector is used in the node of repeat to search repeat node via node index.
- To using the index selector, we need to define the attribute selector with value "index", for instance:
- "selector": "index"
- And in the node where we need to access we need to use the link string start with "$", for instance:
- "value":"$item_length"
- This means to read the value of item_length.

Selector：”*” Function Selector

- "*" function selector
  - The SDDS parser will call the function defined in the extension class, If the value of the attribute is a string start with "*".
  - For instance: "length": "*get_body_length"
  - will call the function get_body_length.
  - If the value of attribute start with the characters of selector and not is a selector, we need use escape.
  - For instance:
  - "value":"*test";
  - The function of function selector is write by you in the extension class.

Step 10: Complete the node of history data

```
" 0x200402 ": {
   "id": "history",
   "name": "history",
   "position": 0,
   "length": "#body_length",
   "repeat": true,
   "count": null,
   "until": "",
   "type": "history_data"
}
```

- According our design, the message_type of the history data is 0x200402.
- The node is in the list of the node one_of.
- Because the history data is multiple record, the attribute repeat must be true.
- The condition of repeat is count, until or length or custom function.
- Now we use length via the id selector to body_length.
- And we use custom type t link a node to define the data type.

Before the finish

- Complete the all node.
  - Now we have learned the all type of nodes of SDDS. And we could complete all the node already.
- Test the schema
  - We could using the test tool test the schema. During the testing we need to set the value option debug to true.
  - And we might use trace to trace the node attributes.
- Write the validate and pack and unpack program
  - There is no programs of validating, packing and unpacking in the SDDS parser. So we need to write them.

More features

- There are 26 attributes in SDDS. We learned 16 attributes already. The other 10 attributes are:
- after_change
  - The string split by "," to define the functions called after reading or writing.
- before_change
  - The string split by "," to define the functions called before reading or writing.
- comment
  - Define the comment of the node for better readable.
- default
  - Define the default value of the node. the default value is used in writing.
- format
  - Define a format string to format the value.

More features

- formula
  - Define the formula to transform the value to real value or to the value for transfer. The variable name always is "A". There ia a formula engine in SDDS parser.
- ignore_errors
  - If the value is true, the parser will ignore the errors in the node during the parser operation.
  - For instance: ignore_errors":true
- Name
  - Define the variable name of node which will be the key of the value in the result for return.
  - If the node value is not required to return, you need not to define the name attribute.
- position
  - Define the offset in the stream for reading or writing. If we read or write according the order of stream, the position attribute is not required. But when we insert or replace a value, we must know the position first.
  - Otherwise we could using the position for checking the schema is correct.
- required
  - Define the value for writing is required. The default value is false.

7 semantics

- There are only using 7 semantics structure in SDDS：
  1. Read/write base type data according the length and unsigned value.
  2. Use a custom type to point to a custom type node.
  3. Read/write the number of bytes according the length, unsigned value for bit operation.
  4. Read/write bits according the length and convert to an integer
  5. Defines a node that consists of multiple bytes nodes
  6. Read/write repeated data according repeat condition.
  7. Selector a node using key from the list.

FAQ

- 1. Is there any packet validation feature in the SDDS engine?
  - There is no packet validation feature in the SDDS engine. You must complete the code in the extension according your protocol.
- 2. Failed in reading or writing with no errors.
  - Check the is there a name property in the node.
  - Check the value of property ignore_errors is default value(false).
- 3. How to write multiple_of in schema?
  - Define a repeat node as a parent, and in same time define a one_of node as child.

FAQ

- 4. What the deference of attribute and property?
  - The attribute is used in schema. The property is used in the class of SDDS engine. According the SDDS Specification, the property is start with "_". We must known the usage of the properties because we need access them in the program.
- 5. There is no 'if' condition in SDDS, how to write 'if' in schema?
  - According the SDDS Specification, we hope the usage is more simple.
  - If we need a 'if' condition we could use the one_of node.
- 6. The selector only could search the node loaded, how to access the node loaded after?
  - If you want to access the node which has not been loaded, you just could access the node which has been loaded from the node loading. That is , you must always search the node backward.

More usage

- Not only for communication project, SDDS could also be used for other project, such as:
- Reading and writing the binary file with format, include:
  - Image file
  - Media file(wave, movie)
  - Archive file (zip, rar, …)
  - PDF document

  - And we need your join to discover the more usage.

Related Resource

- SDDS Specification
  - 【链接待填充】
- SDDS Cheat Sheet
  - 【链接待填充】
- Technical Support:
  - QQ group: 769010107

Please give a star to our project

- If you fell the project is helpful, please start it in github.com.
- And if you interesting to contribute the project, please fork it in github.com.

Thank you!

<byteferry@qq.com>