FAQ »

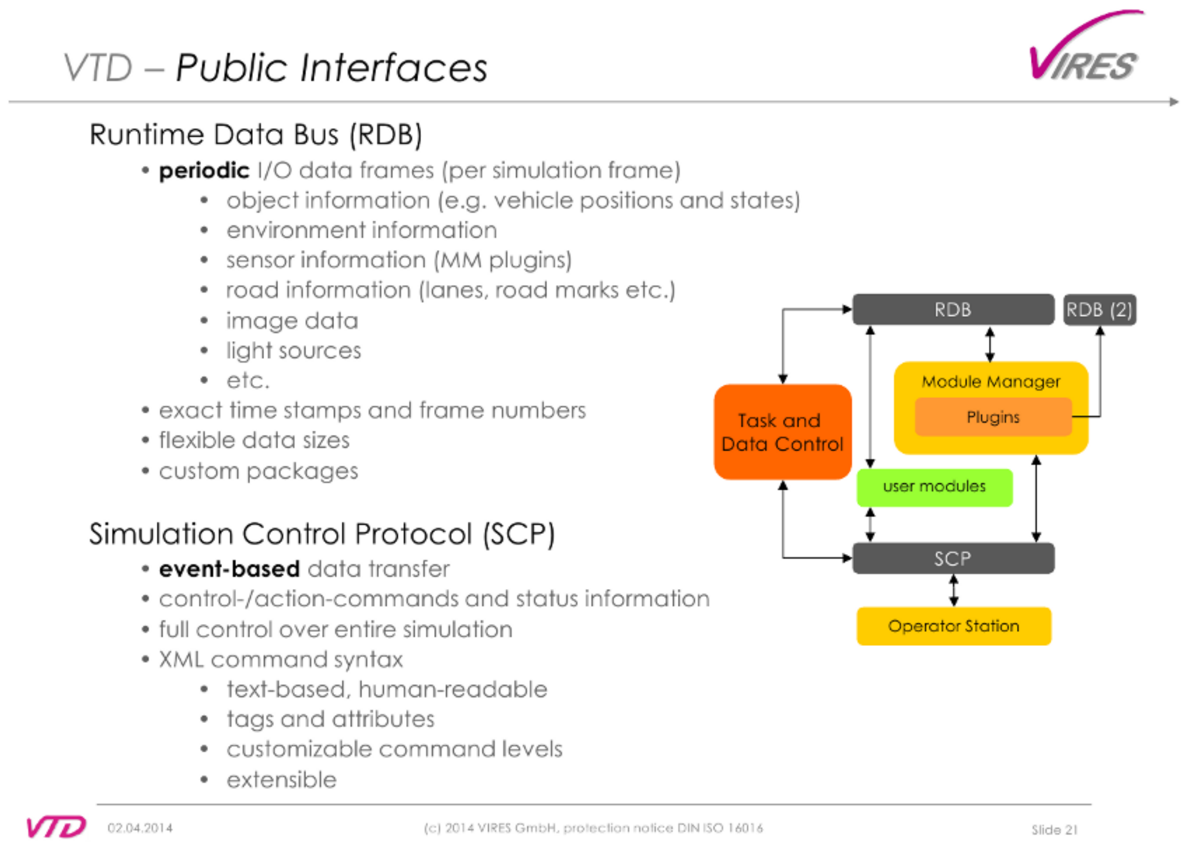
VTD's Simulation Data I/O

Preface

This page provides a short (and most probably incomplete) summary of the data that is made available by the VTD simulation environment. Detailed information about the individual data structures, encoding, units etc. can be found in the respective header files. These are typically located in the directory *Develop/Framework/inc* within the VTD distribution. Additional documents are provided in *Doc/RDB_HTML* and *Doc/SCP_HTML*.

In general, VTD provides two public interfaces which are used for sending and receiving data. These are called *Runtime Data Bus (RDB)* and *Simulation Control Protocol (SCP)*. The former one is used for high frequency data transmission whereas the latter one is used for event-based data only.

The following figure provides an overview:



VTD – Public Interfaces

Runtime Data Bus (RDB)
- **periodic** I/O data frames (per simulation frame)
  - object information (e.g. vehicle positions and states)
  - environment information
  - sensor information (MM plugins)
  - road information (lanes, road marks etc.)
  - image data
  - light sources
  - etc.
- exact time stamps and frame numbers
- flexible data sizes
- custom packages

Simulation Control Protocol (SCP)
- **event-based** data transfer
- control-/action-commands and status information
- full control over entire simulation
- XML command syntax
  - text-based, human-readable
  - tags and attributes
  - customizable command levels
  - extensible

02.04.2014 (c) 2014 VIRES GmbH, protection notice DIN ISO 16016 Slide 21

Runtime Data Bus

Introduction

The Runtime Data Bus (RDB) is the means of first choice to receive run-time data or send recurring information into the system. The primary source / receiver for RDB data is the TaskControl (see figure above). Sensor plug-ins in the ModuleManager may be used to extract data from the primary RDB stream, extend it with additional information gathered within the sensor and send it to the applicable consumers. Plug-ins are only capable of sending RDB data whereas the TaskControl is the only entity within VTD which can send also **receive RDB** data from e.g. user modules.

Formatting

General Structure

An RDB message is basically a vector of lists which contain each an array of entries of identical type. In-between the actual data, some management structures are introduced, so that the actual elements can be parsed easily.

A simple RDB message looks like this (for type definitions, see the RDB header file):

```
RDB_MSG_HDR_t
  RDB_MSG_ENTRY_HDR_t  (pkgId = TypeA)
     entryOfTypeA
     entryOfTypeA
     entryOfTypeA
     :
```

With this message, you may send (or receive) multiple entries of a single type. Each `entry` has to be of a package type which is defined in the header file (e.g. `RDB_OBJECT_STATE_t`, corresponding to `pkdId = RDB_PKG_ID_OBJECT_STATE`)

If you want to pack multiple entry types into a single message, you have to provide an `RDB_MSG_ENTRY_HDR_t` for each array of entries.

Example:

```
RDB_MSG_HDR_t
  RDB_MSG_ENTRY_HDR_t  (pkgId = TypeA)
     entryOfTypeA
     entryOfTypeA
     entryOfTypeA
  RDB_MSG_ENTRY_HDR_t  (pkgId = TypeB)
     entryOfTypeB
  RDB_MSG_ENTRY_HDR_t  (pkgId = TypeC)
     entryOfTypeC
     entryOfTypeC
```

Ideally, a message vector for a complete simulation frame should be enclosed by packages of the types `RDB_PKG_ID_START_OF_FRAME` and `RDB_PKG_ID_END_OF_FRAME`. So, in a perfect world, the previous example looks like this:

```
RDB_MSG_HDR_t
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_START_OF_FRAME)
  RDB_MSG_ENTRY_HDR_t  (pkgId = TypeA)
     entryOfTypeA
     entryOfTypeA
     entryOfTypeA
  RDB_MSG_ENTRY_HDR_t  (pkgId = TypeB)
     entryOfTypeB
  RDB_MSG_ENTRY_HDR_t  (pkgId = TypeC)
     entryOfTypeC
     entryOfTypeC
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_END_OF_FRAME)
```

Now for the various size information fields within the RDB structures. How to calculate them? Actually it's quite easy and shall be illustrated by extending the previous example:

```
RDB_MSG_HDR_t          headerSize  = sizeof( RDB_MSG_HDR_t )
                       dataSize    = sum of all following "headerSize" and "dataSize" information, i.e.
                                     5 * sizeof( RDB_MSG_ENTRY_HDR_t ) +
                                     3 * sizeof( TypeA ) +
                                     1 * sizeof( TypeB ) +
                                     2 * sizeof( TypeC )
  RDB_MSG_ENTRY_HDR_t  pkgId       = RDB_PKG_ID_START_OF_FRAME
                       headerSize  = sizeof( RDB_MSG_ENTRY_HDR_t )
                       dataSize    = 0
                       elementSize = 0
  RDB_MSG_ENTRY_HDR_t  pkgId       = TypeA
                       headerSize  = sizeof( RDB_MSG_ENTRY_HDR_t )
                       dataSize    = 3 * sizeof( TypeA )
                       elementSize = sizeof( TypeA )
     entryOfTypeA
     entryOfTypeA
     entryOfTypeA
  RDB_MSG_ENTRY_HDR_t  pkgId       = TypeB
                       headerSize  = sizeof( RDB_MSG_ENTRY_HDR_t )
                       dataSize    = 1 * sizeof( TypeB )
                       elementSize = sizeof( TypeB )
     entryOfTypeB
  RDB_MSG_ENTRY_HDR_t  pkgId = TypeC
                       headerSize  = sizeof( RDB_MSG_ENTRY_HDR_t )
                       dataSize    = 2 * sizeof( TypeC )
                       elementSize = sizeof( TypeC )
     entryOfTypeC
     entryOfTypeC
  RDB_MSG_ENTRY_HDR_t  pkgId       = RDB_PKG_ID_END_OF_FRAME
                       headerSize  = sizeof( RDB_MSG_ENTRY_HDR_t )
                       dataSize    = 0
                       elementSize = 0
```

Basic and Extended Packages, Trailing Data

Extended Information:

Packages are defined for maximum flexibility. This means that for the complete description of a complex object more data may be required than for a simple object of comparable type.

Therefore, the information about an object may be contained in a basic structure followed immediately - within the same package - by an additional information block (so-called extension). In the respective package's header (of type `RDB_MSG_ENTRY_HDR_t`) the member `flags` will have the flag `RDB_PKG_FLAG_EXTENDED` set in case of an extended package.

Note that if multiple elements of the same package type are packed in a package vector, all of these elements must be either of basic type or of extended type. Mixing these types is not allowed.

For easier casting of basic and extended packages, containers are provided which provide one member of the basic information followed by one member holding the extended information.

Example (for the states of static / dynamic objects):

```
PKG ID = RDB_PKG_ID_OBJECT_STATE)

information about a static object (e.g. obstacle):
  package contains only
      RDB_OBJECT_STATE_BASE_t

information about a dynamic object (e.g. vehicle)
  package contains
      RDB_OBJECT_STATE_BASE_t
  followed by
      RDB_OBJECT_STATE_EXT_t

casting structure provided in the header file:
 {
   RDB_OBJECT_STATE_BASE_t base;
   RDB_OBJECT_STATE_EXT_t  ext;
 } RDB_OBJECT_STATE_t;
```

Packages with Trailing Data

Some packages may be followed by trailing data of flexible size. In this case, a package itself contains information about the number of bytes that will follow the original package. The type and content of the trailing data depend on the package type. The `dataSize` in the package's message entry header must reflect the total of the size of the package type plus the trailing data.

Physical Connection

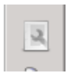
General

RDB data may be sent / received via TCP/IP, UDP or shared memory. See the configuration details of the TaskControl and ModuleManager (in the SCP doc.) for further information.

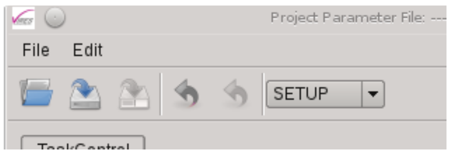
VTD 2.0.0+

Although the parameters that have to be set are equivalent to the ones in previous versions, the way of modifying them is a bit different in VTD 2.0.0 and up. Here's a short description of how to change the parameters. For the parameters and the other options, please refer to the following chapter (VTD 1.4.3 and before).

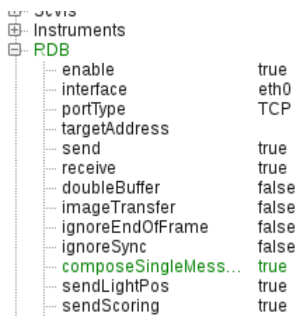
The TaskControl is configured via the GUI using the parameter browser. In the GUI, press the ParamGUI button



The parameter browser will open. Enable the `SETUP` mode (see comboBox on top)



and perform the settings (as described below) under the "RDB" node



Instruments	
RDB	
enable	true
interface	eth0
portType	TCP
targetAddress	
send	true
receive	true
doubleBuffer	false
imageTransfer	false
ignoreEndOfFrame	false
ignoreSync	false
composeSingleMess...	true
sendLightPos	true
sendScoring	true

When you've finished, press the save button, so that the settings are permanently associated with your setup.

VTD 1.4.3 and before

In general, the RDB port is configured in the TaskControl configuration file `Data/Setups/Current/Config/TaskControl/taskControl.xml`. Here (and in the SCP documentation) look for the tag `<RDB>`. The different communication means can be selected by setting the attribute `portType` to the applicable value.

Examples:

RDB via TCP/IP

file: `Data/Setups/Current/Config/TaskControl/taskControl.xml`

```
    <RDB            name="default"
                    portType="TCP"
                    imageTransfer="false"/>
```

file: `Data/Setups/Current/Config/ModuleManager/moduleManager.xml`

```
    <RDB>
        <Port name="RDBraw" number="48190" type="TCP" />
    </RDB>
```

RDB via UDP

file: `Data/Setups/Current/Config/TaskControl/taskControl.xml`

```
    <RDB            name="default"
                    portType="UDP"
                    imageTransfer="false"/>
```

file: `Data/Setups/Current/Config/ModuleManager/moduleManager.xml`

```
    <RDB>
        <Port name="RDBraw" number="48190" type="UDP" />
    </RDB>
```

Note: when changing the default RDB port's physical connection, always make sure that the you adapt all consumers accordingly (here: ModuleManager)

RDB communication via Shared Memory

As of VTD 1.3.1, the taskControl may communicate with other components by means of Shared Memory segments. In order to configure the TC for this sort of communication, edit the config file `Data/Setups/Current/Config/TaskControl/taskControl.xml` as follows:

```
    <RDB            name="default"
                    enable="true"
                    portType="SHM"
                    imageTransfer="false"
                    send="true"/>

    <RDB            name="shmIn"
                    enable="true"
                    portType="SHM"
                    imageTransfer="false"
                    receive="true"/>
```

Note: The first interface must be named *default*, otherwise a parallel communication via network will be established (which is, of course, possible if your application requires this feature).

The only VTD core component currently also capable of Shared Memory communication via RDB is the ModuleManager. In its configuration file `Data/Projects/Current/Config/ModuleManager/moduleManager.xml` the entries for the RDB ports have to be setup as follows:

```
<RDB>
    <Port name="RDBraw" type="SHM" receive="true" />
    <Port name="RDBraw" type="SHM" send="true" />
</RDB>
```

The following principles apply for SharedMemory communication with the taskControl:

- sending and receiving is to be performed in separate shared memory segments
- there may be multiple send and receive SHM-blocks
- unless otherwise configured, the size of each SHM block is `RDB_SHM_SIZE_TC`
- unless otherwise configured, the SHM key for sending data to the TC is `RDB_SHM_ID_TC_IN` and for receiving data from TC is `RDB_SHM_ID_TC_OUT`
- individual shared memory configurations may be achieved by using the attributes *key*, *size* and *doubleBuffer* in the above mentioned SCP tags

More information about the Shared Memory structure in VTD can be found here:

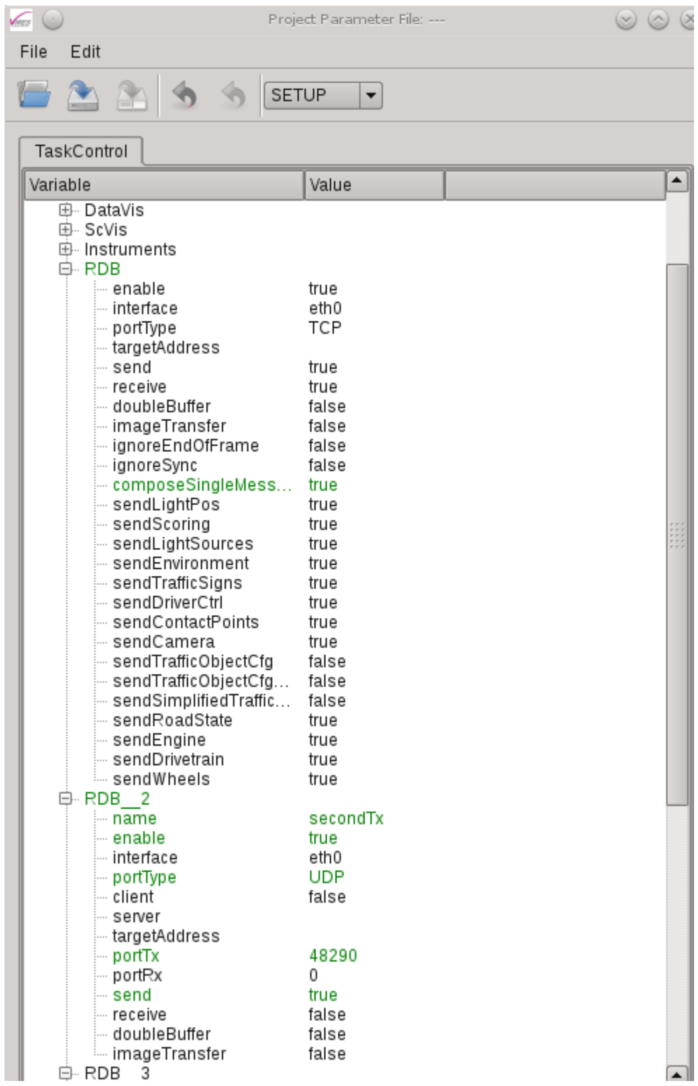
FiguresManualO_SHM.pdf

More than one RDB connection

If you wish to establish more than one RDB connection to/from the TaskControl, you may specify additional interfaces in the TC configuration.

VTD 2.0.0+

For a mixed configuration using TCP/IP and UDP connections, you have to activate two or three dedicated RDB connections. Two connections are needed if you use the TCP/IP connection for tx/rx and the UDP connection for tx only:



Variable	Value
DataVis	
ScVis	
Instruments	
RDB	
enable	true
interface	eth0
portType	TCP
targetAddress	
send	true
receive	true
doubleBuffer	false
imageTransfer	false
ignoreEndOfFrame	false
ignoreSync	false
composeSingleMess...	true
sendLightPos	true
sendScoring	true
sendLightSources	true
sendEnvironment	true
sendTrafficSigns	true
sendDriverCtrl	true
sendContactPoints	true
sendCamera	true
sendTrafficObjectCfg	false
sendTrafficObjectCfg...	false
sendSimplifiedTraffic...	false
sendRoadState	true
sendEngine	true
sendDrivetrain	true
sendWheels	true
RDB__2	
name	secondTx
enable	true
interface	eth0
portType	UDP
client	false
server	
targetAddress	
portTx	48290
portRx	0
send	true
receive	false
doubleBuffer	false
imageTransfer	false
RDB__3	

Note: all of the parameters given in green for connection RDB__2 have to be provided. The connection has to be named, needs a port definition and the property `send` has to be activated. The name of the ethernet device (here: `eth0`) is important!

If you wish to receive on RDB via UDP, either change the above configuration by turning off `send` and activating `receive` or add connection RDB__3 and configure it for `receive` only.

VTD 1.4.3 and older

Edit the file (`Data/Setups/Current/Config/TaskControl/taskControl.xml`).

Example:

Two RDB interfaces are made available:

- default interface via TCP on port 48190
- interface *RDBaddOn* via UDP on eth0, sending over port 48199, receiving on port 48200; receiving is disabled (note: the receiving port must be given despite the fact that receiving is not enabled; it also must be different from the sending port; this is a workaround for VTD versions prior to VTD 2.0)

```
<RDB            enable="true"
                portType="TCP"
                imageTransfer="false" />

<RDB            name="RDBaddOn"
                enable="true"
                interface="eth0"
                portType="UDP"
                send="true"
                receive="false"
                portTx="48199"
                portRx="48200"
                imageTransfer="false" />
```

Contents

Quick Overview

This table gives a quick overview of the main producers and consumers of data packages (**within the standard distribution of VTD**). Where *n/a* is stated for a producer and consumer, a user component will typically be involved

package	producers	consumers
RDB_PKG_ID_START_OF_FRAME	taskControl, perfectSensor	taskControl
RDB_PKG_ID_END_OF_FRAME	taskControl, perfectSensor	taskControl
RDB_PKG_ID_COORD_SYSTEM	part of other messages	part of other messages
RDB_PKG_ID_COORD	part of other messages	part of other messages
RDB_PKG_ID_ROAD_POS	taskControl, perfectSensor	n/a
RDB_PKG_ID_LANE_INFO	taskControl, perfectSensor	n/a
RDB_PKG_ID_ROADMARK	perfectSensor	n/a
RDB_PKG_ID_OBJECT_CFG	taskControl	n/a
RDB_PKG_ID_OBJECT_STATE	taskControl, perfectSensor	taskControl
RDB_PKG_ID_VEHICLE_SYSTEMS	taskControl, perfectSensor	taskControl
RDB_PKG_ID_VEHICLE_SETUP	taskControl	taskControl
RDB_PKG_ID_ENGINE	taskControl, perfectSensor	taskControl
RDB_PKG_ID_DRIVETRAIN	taskControl, perfectSensor	taskControl
RDB_PKG_ID_WHEEL	taskControl, perfectSensor	taskControl
RDB_PKG_ID_PED_ANIMATION	taskControl	taskControl
RDB_PKG_ID_SENSOR_STATE	perfectSensor	taskControl
RDB_PKG_ID_SENSOR_OBJECT	perfectSensor	taskControl
RDB_PKG_ID_CAMERA	taskControl, imageGenerator, cameraSensor	taskControl
RDB_PKG_ID_CONTACT_POINT	taskControl, odrGateway	n/a
RDB_PKG_ID_TRAFFIC_SIGN	taskControl, perfectSensor	n/a
RDB_PKG_ID_ROAD_STATE	taskControl, perfectSensor	taskControl
RDB_PKG_ID_IMAGE	taskControl, imageGenerator	n/a
RDB_PKG_ID_LIGHT_SOURCE	n/a	taskControl
RDB_PKG_ID_ENVIRONMENT	taskControl, perfectSensor	n/a
RDB_PKG_ID_TRIGGER	n/a	taskControl
RDB_PKG_ID_DRIVER_CTRL	taskControl	dynamicsPlugIn
RDB_PKG_ID_TRAFFIC_LIGHT	taskControl, perfectSensor	n/a
RDB_PKG_ID_SYNC	n/a	taskControl
RDB_PKG_ID_DRIVER_PERCEPTION	taskControl	n/a
RDB_PKG_ID_LIGHT_MAP	n/a	imageGenerator
RDB_PKG_ID_TONE_MAPPING	n/a	imageGenerator
RDB_PKG_ID_ROAD_QUERY	odrGateway	n/a
RDB_PKG_ID_SCP	n/a	taskControl
RDB_PKG_ID_TRAJECTORY	taskControl	n/a
RDB_PKG_ID_DYN_2_STEER	n/a	taskControl
RDB_PKG_ID_STEER_2_DYN	n/a	n/a
RDB_PKG_ID_PROXY	sensor plugIn	n/a
RDB_PKG_ID_MOTION_SYSTEM	n/a	taskControl
RDB_PKG_ID_OCCLUSION_MATRIX	perfectSensor	n/a
RDB_PKG_ID_FREESPACE	sensor plugIn	n/a
RDB_PKG_ID_DYN_EL_SWITCH	n/a	taskControl
RDB_PKG_ID_DYN_EL_DOF	n/a	taskControl
RDB_PKG_ID_IG_FRAME	taskControl	n/a
RDB_PKG_ID_RAY	taskControl, multiRaySensor	taskControl, multiRaySensor
RDB_PKG_ID_RT_PERFORMANCE	taskControl	n/a
RDB_PKG_ID_SYMBOL_STATE	n/a	taskControl

Data from VTD

VTD provides an extensive set of binary data for each simulation step. The overall amount of data that is available on RDB is listed in the respective documentation. Some of the more important / frequent contents are given in the following list:

Player Data (Vehicles, Pedestrians)

- position, orientation (heading, pitch, roll) and size (bounding-box)
- category of the player (vehicle, pedestrian)

Details of Vehicles

- size, mass, axle distance
- information about drivetrain (speed, torque, gear...)
- information about driver-vehicle-interface (e.g. pedals, steering wheel)
- detailed information about wheels (steering angle, radius, forces etc.)
- brake pressure, spring compression (per wheel)
- status of vehicle lights

- **Note:** depending on the vehicle dynamics model, not all values may be available or may be interpreted. For the *standard VTD vehicle dynamics* implementations (single-track model / physics-engine model), the following inputs are supported:
  - steering target / acceleration target / gear (D/N)
  - throttle / brake / steering wheel / gear (D/N)

Details of current road sections

- drive lane information (width, id, ...)
- road marks (type, color, ...)
- traffic signs and traffic lights (including current and future states)

Images of the Image Generator

- height, width
- image data (visual range, depth information, infrared, ...)
- information about the camera which is used for generating the image (make sure you configure the TaskControl to the latest IG protocol version for this feature, i.e. "precision10" or higher)

Sensor Data

Perfect sensors may be positioned at arbitrary locations of vehicles. These may be used to simulate real sensors. Sensor output data is provided on dedicated connections of each sensor. The protocol also follows the RDB definition.

- position of the sensor
- distance to detected objects (with global object reference)

Common Data

- time-of-day
- sky state
- weather conditions (fog, rain, snow)
- visibility
- road conditions (dry, wet)

Data into VTD

As stated above you may also send RDB-formatted data into the simulation. In the usual case (i.e. Standard VTD configuration), only the TaskControl will be able to accept these data. The following RDB data types are the ones which the TaskControl understands (excerpt from receiver source code):

```
        case RDB_PKG_ID_LIGHT_SOURCE:
        case RDB_PKG_ID_DRIVER_CTRL:
        case RDB_PKG_ID_SYNC:
        case RDB_PKG_ID_OBJECT_STATE:
        case RDB_PKG_ID_ENGINE:
        case RDB_PKG_ID_DRIVETRAIN:
        case RDB_PKG_ID_WHEEL:
        case RDB_PKG_ID_VEHICLE_SYSTEMS:
        case RDB_PKG_ID_VEHICLE_SETUP:
        case RDB_PKG_ID_TRIGGER:
        case RDB_PKG_ID_TRAFFIC_LIGHT:
        case RDB_PKG_ID_END_OF_FRAME:
        case RDB_PKG_ID_START_OF_FRAME:
```

The following messages are understood but may not be completely implemented. So do **NOT** send them without being requested or authorized to do so:

```
        case RDB_PKG_ID_SENSOR_STATE:
        case RDB_PKG_ID_SENSOR_OBJECT:
        case RDB_PKG_ID_OBJECT_CFG:
        case RDB_PKG_ID_PED_ANIMATION:
        case RDB_PKG_ID_CUSTOM_SCORING:
        case RDB_PKG_ID_CUSTOM_AUDI_FORUM:
        case RDB_PKG_ID_SCP:
```

Customizing Data Streams

Total Number of Dynamic Objects sent by TaskControl

The number of dynamic objects sent by the TaskControl to the primary RDB connection (typically port 48190) may be limited, so that a consumer or the network are not overloaded. This may be done using the following SCP command (also available as TC configuration parameters):

```
<TaskControl>
...
  <RDB ... maxDynObjectsSend="20"
           maxDynObjectsRefPlayer="Ego" .../>
</TaskControl>
```

If you just want to use the only external player as reference player, then you may skip the attribute `maxDynObjectsRefPlayer`.

Important note: trimming the total number of objects sent by the TC will also affect all subsequent modules (e.g. sensors). So, make sure you set the limit to a level sufficient for all connected consumers!

Reading, Writing and Interpreting specific RDB Data

How do I read the roadmark information?

Roadmark information is provided as one or more entries of type `RDB_PKG_ID_ROADMARK`. Each roadmark entry consists of a block of type `RDB_ROADMARK_t` and an optional array of elements of type `RDB_POINT_t`. The number of trailing elements is given by the member `noDataPoints` of the first block.

Example: Roadmark information consisting of meta information only:

```
RDB_MSG_HDR_t
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_ROADMARK)
     RDB_ROADMARK_t
     RDB_ROADMARK_t
     RDB_ROADMARK_t
```

Example: Roadmark information consisting of meta information and 64 tesselation points per roadmark

```
RDB_MSG_HDR_t
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_ROADMARK)
     RDB_ROADMARK_t
        64 x RDB_POINT_t
     RDB_ROADMARK_t
        64 x RDB_POINT_t
     RDB_ROADMARK_t
        64 x RDB_POINT_t
```

The structure `RDB_ROADMARK_t` has (among others) the following important members:

```
    uint32_t    playerId;                /**< id of the player to which roadmark belongs                           @unit _
    int8_t      id;                      /**< id of this road mark                                                 @unit [0.
    int8_t      prevId;                  /**< id of predecessor                                                    @unit [-1
    int8_t      nextId;                  /**< id of successor                                                      @unit [-1
    float       lateralDist;             /**< lateral distance to vehicle ref. point and dir                       @unit m
    float       yawRel;                  /**< yaw angle relative to vehicle dir                                    @unit rad
    double      curvHor;                 /**< horizontal curvature                                                 @unit 1/m
    double      curvHorDot;              /**< change of horizontal curvature                                       @unit 1/m
    float       startDx;                 /**< start of road mark in driving dir                                    @unit m
    float       previewDx;               /**< distance of last valid measurement                                   @unit m
    float       width;                   /**< width of road mark                                                   @unit m
    float       height;                  /**< height of road mark                                                  @unit m
    double      curvVert;                /**< vertical curvature                                                   @unit 1/m
    double      curvVertDot;             /**< change of vertical curvature                                         @unit 1/m
    uint8_t     type;                    /**< type of road mark                                                    @unit @li
    uint8_t     color;                   /**< color of road mark                                                   @unit @li
    uint16_t    noDataPoints;            /**< number of tesselation points following this package                  @unit _
```

The **decoding** of roadmark tesselation points in terms of C-programming is shown in the following excerpt of the `RDBHandler.cc`:

```
void
RDBHandler::print( const RDB_ROADMARK_t & info, unsigned char ident, bool csv, bool csvHeader )
{
    :
    :

    if ( info.noDataPoints )
    {
        RDB_POINT_t* pt = ( RDB_POINT_t* ) ( ( ( unsigned char* ) ( &info ) ) + sizeof( RDB_ROADMARK_t ) );

        for ( int i = 0; i < info.noDataPoints; i++ )
            print( pt[i], ident + 4 );
    }
}
```

General starting point of a roadmark is relative to the reference point of a vehicle (center of rear axle, ground level) as follows:

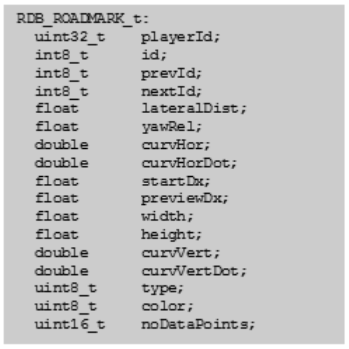
- null distance in vehicle x-direction
- `lateralDist` in vehicle y-direction (positive to the left)
- `yawRel` as relative heading to the vehicle

The valid part of a roadmark starts at a run-length of `startDx` relative to the starting point and reaches up to `previewDx`. These distances are measured along the road.

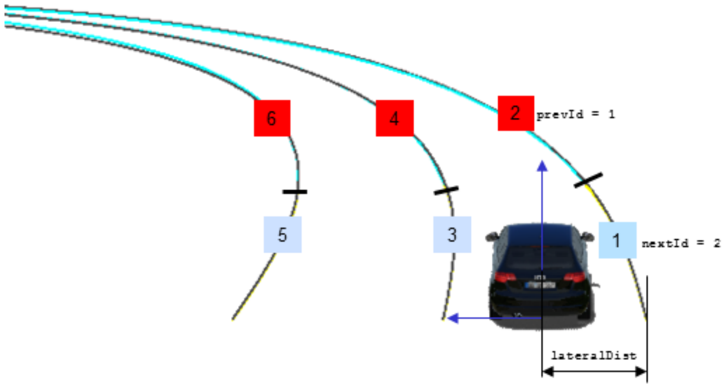
The `lateralDist` of a roadmark is measured to its inner edge (as seen from the sensor carrier / reference vehicle), i.e. the `width` has to be added to reach the outer edge of a roadmark.

All roadmarks are given unique `id` numbers so that a roadmark may also be split into up to two sections and the relation between the sections is given by the members `id`, `prevId`, `nextid`.

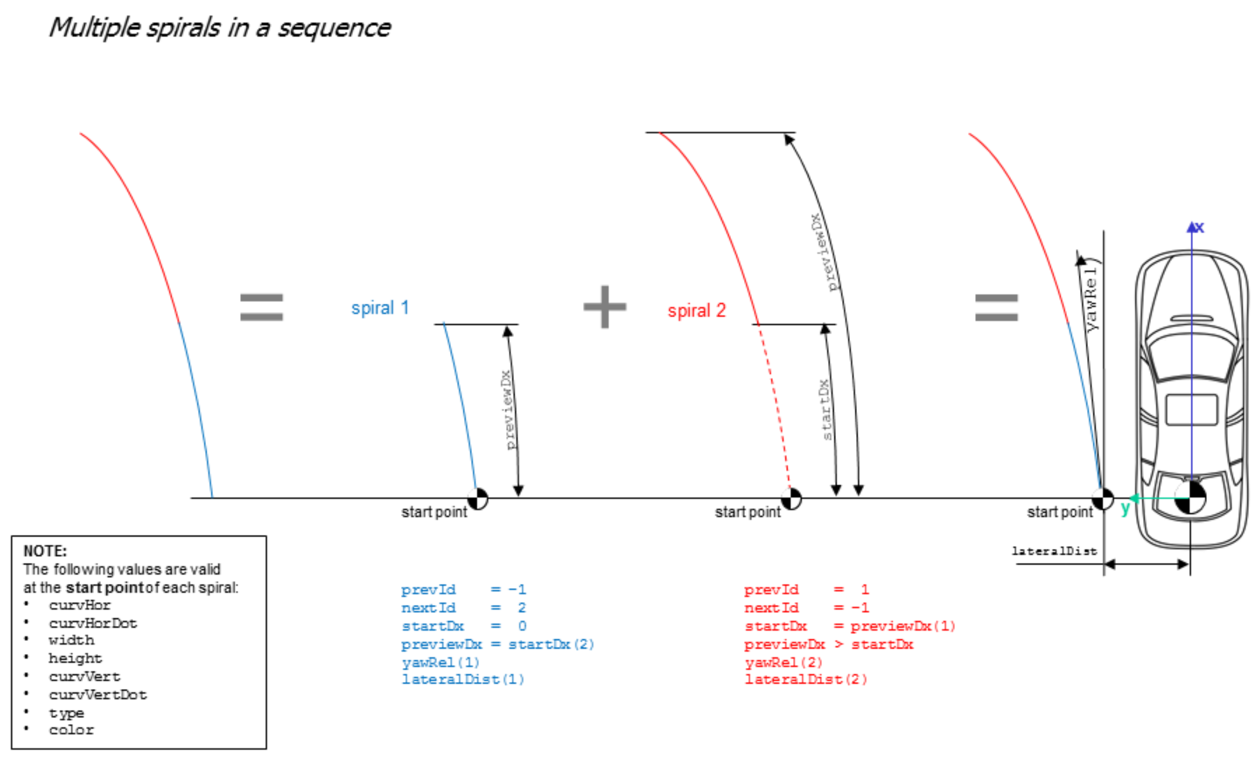
The following figure illustrates the road mark feature in general:



```
RDB_ROADMARK_t:
  uint32_t    playerId;
  int8_t      id;
  int8_t      prevId;
  int8_t      nextId;
  float       lateralDist;
  float       yawRel;
  double      curvHor;
  double      curvHorDot;
  float       startDx;
  float       previewDx;
  float       width;
  float       height;
  double      curvVert;
  double      curvVertDot;
  uint8_t     type;
  uint8_t     color;
  uint16_t    noDataPoints;
```



The following figure illustrates how a sequence of road mark entries describing the same physical road mark shall be interpreted



Important Note: Road marks are generated in a layer above the actual road surface in order to reduce the risk of flickering. The default distance between the road and a road mark polygon is 2cm (0.02m). This distance may be changed but has to be done so in two locations:

a) Road Designer

In the Road Designer's configuration file (`$VTD/Runtime/Tools/ROD/TileLib/SetupFilesPool/VTL/Full/TT_SETUP_base.DAT`), set the value of thee road mark offset with the following parameter (here: set to 3mm)

```
TED_ROADMARK_OFFSET     0.003    # 3mm@
```

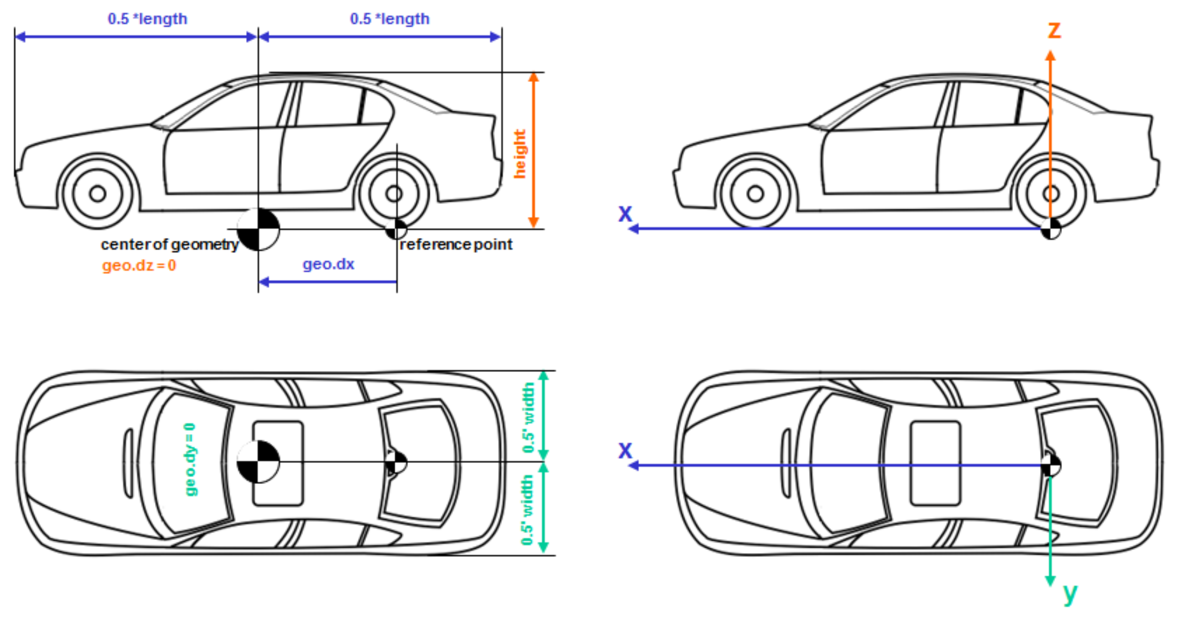
b) Perfect Sensor (or derived class)

In the ModuleManager's configuration file (e.g. `VTD/Data/Setups/Current/Config/ModuleManager/moduleManager.xml`), set the offset parameter to a consistent value:

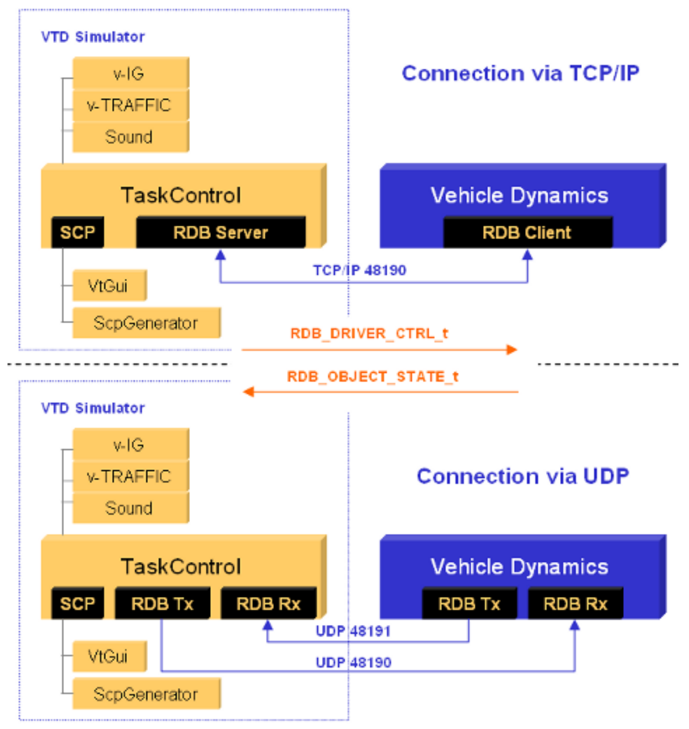
```
<Filter objectType="roadMarks" ... tesselateOffsetZ="0.003" />
```

What does the position and geometry information in the OBJECT_STATE look like?

In packages containing data of type OBJECT_STATE_t (typically positions of vehicles etc.), you will find the position of an object's reference point (in member pos), the object's size and the position of its center of geometry (both in member geo with dimX = length, dimY = width and dimZ = height). For a vehicle, these elements are shown in the following image (note that y-offset and z-offset are both zero):



How do I connect my vehicle dynamics (and other components) to VTD?



Physical connection

Your vehicle dynamics (and other components) may be connected to VTD via the RDB protocol. The physical connection is an Ethernet connection which may be configured to run either via TCP/IP or UDP. The type of connection is specified in the file `Data/Setups/Current/Config/TaskControl/taskControl.xml` in the section `<RDB>`.

Example:

```
<TaskControl>
:
  <RDB enable="true"
       portType="TCP"
       imageTransfer="false" />
:
```

```
</TaskControl>
```

In case of a TCP/IP-connection, the taskControl opens a server port (id: 48190) which will be used for sending and receiving data. Your component should open the corresponding client port. In case of a UDP connection, the taskControl will open one port for sending (id: 48190) and one for receiving data (id: 48191). Your component should open the matching ports.

Data stream from TaskControl to VehicleDynamics

The taskControl will send a considerable amount of data which can be used for a vehicle dynamics simulation as well as for sensors etc. In the remaining parts of this chapter, we will focus on the contents which are relevant for linking a vehicle dynamics package to VTD. Other data contents which are sent via the taskControl may be skipped by the receiving task.

Unless driver model and vehicle dynamics are both managed by your external component, VTD will usually provide the driver input and your component will then compute the response of the vehicle. The taskControl will send you a package or a series of packages which have at least the following content (for the symbolic constants used in the examples, please see `Develop/Framework/inc/viRDBIcd.h` in your VTD distribution):

```
RDB_MSG_HDR_t
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_START_OF_FRAME)
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_DRIVER_CTRL)
     RDB_DRIVER_CTRL_t
     RDB_DRIVER_CTRL_t
     :
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_OBJECT_STATE)
     RDB_OBJECT_STATE_t
     :
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_END_OF_FRAME)
```

The content may also be distributed over various packages, e.g.:

```
package 1:
RDB_MSG_HDR_t
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_START_OF_FRAME)
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_DRIVER_CTRL)
     RDB_DRIVER_CTRL_t
     RDB_DRIVER_CTRL_t
     :

package 2:
RDB_MSG_HDR_t
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_OBJECT_STATE)
     RDB_OBJECT_STATE_t
     :
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_END_OF_FRAME)
```

The important point is that your components monitors the `START_OF_FRAME` and `END_OF_FRAME` in order to make sure that all information of a simulation frame has been received. For details about the formatting of RDB, the coding and decoding, please see also Data from VTD.

Now, all you have to do is to decode above packages and look for the package of type `RDB_PKG_ID_DRIVER_CTRL` which refers to your player (look for the member `playerId` in the structure `RDB_DRIVER_t`). In this package, you will find quite a few (potential) inputs. Which of these inputs are valid is indicated by the member `validityFlags` of the structure `RDB_DRIVER_t`. Assuming, an external mockup is connected to VTD which is driven by a human driver, then the flags (`RDB_DRIVER_INPUT_VALIDITY_STEERING_WHEEL | RDB_DRIVER_INPUT_VALIDITY_THROTTLE | RDB_DRIVER_INPUT_VALIDITY_BRAKE`) will be set. In case of a software driver model running within the VTD traffic simulation, the flags (`RDB_DRIVER_INPUT_VALIDITY_TGT_ACCEL | RDB_DRIVER_INPUT_VALIDITY_TGT_STEERING`) will be set.

The timestamp and frame number of the input will be given in the header of the message (type `RDB_MSG_HDR_t`). Based on the input, your vehicle dynamics should perform the necessary update and compute your own vehicle's position.

Deactivation of built-in VTD dynamics

If you are running your own vehicle dynamics via RDB connection, make sure there is no competing instance of a VTD vehicle dynamics computed by the default ModuleManager plug-in. Please check the following:

1. locate the file that is to be adapted
   1. if you have a file "Data/Projects/Current/Config/ModuleManager/moduleManager.xml" use this file
   2. if you have a file "Data/Setups/Current/Config/ModuleManager/moduleManager.xml" use this file
   3. if you have neither tile, create the directory "Data/Setups/Current/Config/ModuleManager" and copy the file "Data/Distros/Current/Config /ModuleManager/moduleManager.xml" there
2. open the file
3. deactivate all entries of type `<DynamicsPlugin>`

Data stream from VehicleDynamics to TaskControl

After receiving an input from the TaskControl, the VehicleDynamics should return the latest inertial position and orientation of the vehicle to VTD. Typically, vehicle dynamics tools run considerably faster than VTD (e.g. 500Hz vs. 60Hz). Therefore, it is not necessary to send the result of each simulation step back to VTD. Instead, sending the results with approx. 60Hz to the TaskControl is sufficient. The actual send rate to VTD is determined by the synchronization mode (see above) which is foreseen between your vehicle dynamics and VTD. In *frame-synchronous mode*, VTD will provide you with the new time stamp and your dynamics should return the position of the vehicle for exactly this time stamp. In *free mode* both applications (VTD and vehicle dynamics) will run asynchronously. In this case, the real-time covered by both applications shall be identical. In order to minimize jitter in the standard case that VTD is running with 60Hz it is recommended that vehicle dynamics send state messages after 17ms, 17ms and 16ms (and so on). This makes 50ms over 3 frames and corresponds to 3x 16.667ms which are computed by VTD.

Now for the data contents: your package to VTD should look similar to the following:

```
RDB_MSG_HDR_t
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_START_OF_FRAME)
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_OBJECT_STATE)
     RDB_OBJECT_STATE_t
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_WHEEL)
     RDB_WHEEL_t
     RDB_WHEEL_t
     RDB_WHEEL_t
     RDB_WHEEL_t
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_ENGINE)
     RDB_ENGINE_t
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_VEHICLE_SYSTEMS)
     RDB_VEHICLE_SYSTEMS_t
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_DRIVETRAIN)
     RDB_DRIVETRAIN_t
  RDB_MSG_ENTRY_HDR_t  (pkgId = RDB_PKG_ID_END_OF_FRAME)
```

The most important content is the `RDB_OBJECT_STATE_t`. All other parts may be omitted for a simple connection between VTD and an external vehicle dynamics. Make sure that you are sending an extended object state (i.e. including speed and acceleration) so that the extrapolation algorithms within VTD can work properly (again, see Data from VTD for details about extended packages).

That should be it (for a simple connection).

How can I send driver controls directly to VTD (i.e. without ModuleManager plug-in)?

If you want to send driver commands directly to VTD without using the ModuleManager plug-in interface, you may do so by

- connecting to the TC RDB port (typically by providing a TCP client connection on port 48190)
- composing and sending driver control commands according to the following example:

```
void sendDriver( int & sendSocket, const double & simTime, const unsigned int & simFrame )
{
  Framework::RDBHandler myHandler;

  myHandler.initMsg();

  RDB_DRIVER_CTRL_t *myDriver = ( RDB_DRIVER_CTRL_t* ) myHandler.addPackage( simTime, simFrame, RDB_PKG_ID_DRIVER_CTRL );

  if ( !myDriver )
    return;

    myDriver->playerId       = mOwnPlayerId;
    myDriver->steeringWheel  = 9.0 * sin( 0.8 * simTime );
    myDriver->throttlePedal  = 0.5 * ( 1 + sin( 0.8 * simTime ) );
    myDriver->brakePedal     = 0.0;
    myDriver->clutchPedal    = 0.0f;
    myDriver->gear           = RDB_GEAR_BOX_POS_D;
    myDriver->validityFlags  = RDB_DRIVER_INPUT_VALIDITY_STEERING_WHEEL | RDB_DRIVER_INPUT_VALIDITY_THROTTLE | RDB_DRIVER_INPU

  int retVal = send( sendSocket, ( const char* ) ( myHandler.getMsg() ), myHandler.getMsgTotalSize(), 0 );

  if ( !retVal )
    fprintf( stderr, "sendDriver: could not send driver\n" );
}
```

The `RDBHandler` can be found in `VTD/Develop/Framework/RDBHandler`.

What is the coding of the lane types in the RDB messages?

The lane types are reported according to the OpenDRIVE convention. For VTD, the following set of numerical codes applies:

```
    enum EnLaneType {
        ODR_LANE_TYPE_NONE = 0,
        ODR_LANE_TYPE_DRIVING,
        ODR_LANE_TYPE_STOP,
        ODR_LANE_TYPE_SHOULDER,
        ODR_LANE_TYPE_BIKING,
        ODR_LANE_TYPE_SIDEWALK,
        ODR_LANE_TYPE_BORDER,
        ODR_LANE_TYPE_RESTRICTED,
        ODR_LANE_TYPE_PARKING,
        ODR_LANE_TYPE_MWY_ENTRY,
        ODR_LANE_TYPE_MWY_EXIT,
        ODR_LANE_TYPE_SPECIAL1,
        ODR_LANE_TYPE_SPECIAL2,
        ODR_LANE_TYPE_SPECIAL3,
        ODR_LANE_TYPE_SPECIAL4,
        ODR_LANE_TYPE_DRIVING_ROADWORKS,
        ODR_LANE_TYPE_TRAM,
        ODR_LANE_TYPE_RAIL
    };
```

How are traffic signs coded?

Traffic sign type and sub-type comply with German Traffic Rules (Strassenverkehrsordnung). A complete list can be found via German authorities' websites or e.g. here

http://www.findthatpdf.com/search-10547214-hPDF/download-documents-vzkat-handbuch-pdf.htm

Distinction between different co-ordinate systems

When sending an element of type *RDB_COORD_t* as a member of another RDB structure, this element will always indicate the co-ordinate system it is meant to represent (i.e. the way you have to interpret the different members - x, y, z, h, p, r - of the structure). The following co-ordinate systems are available in the RDB data interface:

```
#define RDB_COORD_TYPE_INERTIAL         0  /**< inertial co-ordinate system              @version 0x0101 */
#define RDB_COORD_TYPE_PLAYER           2  /**< player co-ordinate system                @version 0x0100 */
#define RDB_COORD_TYPE_SENSOR           3  /**< sensor-specific co-ordinate system       @version 0x0100 */
#define RDB_COORD_TYPE_USK              4  /**< universal sensor co-ordinate system      @version 0x0100 */
#define RDB_COORD_TYPE_USER             5  /**< relative to a user co-ordinate system    @version 0x0100 */
#define RDB_COORD_TYPE_WINDOW           6  /**< window co-ordinates [pixel]              @version 0x0100 */
#define RDB_COORD_TYPE_TEXTURE          7  /**< texture co-ordinates [normalized]        @version 0x010C */
#define RDB_COORD_TYPE_RELATIVE_START   8  /**< co-ordinate relative to start pos.       @version 0x0110 */
#define RDB_COORD_TYPE_GEO              9  /**< geographic co-ordinate                   @version 0x0118 */
#define RDB_COORD_TYPE_TRACK           10  /**< track co-ordinate (x=s, y=t )            @version 0x0119 */
```

For the definition of the co-ordinate systems, see General_Definitions

The **taskControl** understands the following co-ordinates systems (i.e. you may send *RDB_OBJECT_STATE_BASE* data in these systems):

- *RDB_COORD_TYPE_INERTIAL*: an object will be positions at the given inertial position
- *RDB_COORD_TYPE_RELATIVE_START*: an object will be placed at an inertial position whose delta to the initial position is given in the received co-ordinate (typical use case: a vehicle dynamics simulation starts internally at x/y/z/h = 0/0/0/0 although the object is placed in the scenario at non-zero co-ordinates)
- *RDB_COORD_TYPE_GEO*: the incoming co-ordinate (including heading) is converted to the corresponding inertial co-ordinate using the projection information given in the respective OpenDRIVE map.

The **taskControl** will always send data in *RDB_COORD_TYPE_INERTIAL*

The **PerfectSensor** plug-in of the **moduleManager** may be set to different output systems and will, thus, provide its data in

- *RDB_COORD_TYPE_INERTIAL*
- *RDB_COORD_TYPE_PLAYER*
- *RDB_COORD_TYPE_SENSOR*
- *RDB_COORD_TYPE_USK* (default)
- *RDB_COORD_TYPE_GEO*

For the configuration of the *PerfectSensor*, see Co-ordinate systems of PerfectSensor

Sending Custom Data via RDB

There is a way of using the RDB mechanisms in VTD and still transmitting custom data packages which are not part of the RDB specification. Just use packages of type *RDB_PKG_ID_PROXY* and attach your custom data as trailing data (payload) to the packages.

The definition of the proxy package is

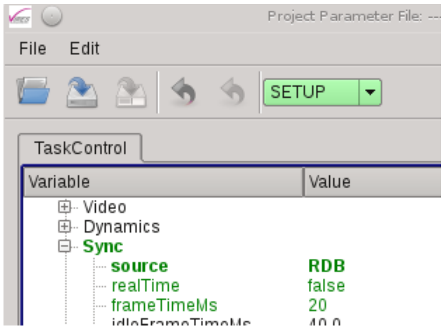
```

/** ------ wrapper for forwarded messages -------
 * @note this package is followed immediately by "dataSize" bytes of data, containing the actual forwarded message */
typedef struct
{
    uint16_t  protocol;   /**< protocol identifier of the wrapped package           @unit _   @version 0x0112 */
    uint16_t  pkgId;      /**< unique pkg id                                        @unit _   @version 0x0112 */
    uint32_t  spare[6];   /**< some spares                                          @unit _   @version 0x0112 */
    uint32_t  dataSize;   /**< number of data bytes following this entry            @unit _   @version 0x0112 */
} RDB_PROXY_t;
```

The values you choos for the members *protocol* and *pkgId* are up to your individual implementation. Just make sure that the member *dataSize* is correct and that the package's part *RDB_PROXY_t* is immediately followed by `dataSize` bytes of custom data.

How to trigger VTD via RDB

In order to control the execution of each time step from an external entity, it is recommended to use the RDB trigger mechanism. This means that the user can control the activation and size (i.e. delta time) of each step. For this, do the following:

- set the property `Sync->source` of the TaskControl to *RDB*



Variable	Value
Video	
Dynamics	
Sync	
source	RDB
realTime	false
frameTimeMs	20
idleFrameTimeMs	40.0

- apply the configuration and start your scenario
- connect to the TaskControl's RDB port
- send the `RDB_PKG_ID_TRIGGER` package whenever you want to execute the next step

The following programming example is an implementation of this functionality:

package: rdbTrigger.20170522.tgz

It sends a time step of 43ms every 500ms (so, far slower than real-time). For compilation, just cd into *RDBTriggerSample* and type *./compile.sh*. **Note:** no network reading is implemented, so the TaskControl will close the connection after a while.

Programming Examples

Helper Class: RDBHandler

In connection with our VTD distribution, we provide a free helper class: `Framework::RDBHandler`. With this class, you will be able to easily compose, analyze and print RDB messages without having to re-invent the wheel all the time. The class is for free, so use it wherever you deem it necessary. For a programmer's description of all routines, please take a look at the header file `RDBHandler.hh`.

package: rdbHandler.20170102.tgz

Include Files

You only have to include `RDBHandler.hh` in your source code. It will automatically also include the interface file `viRDBIcd.h`

Creating Messages

In order to compose a message (e.g. for sending it afterwards), do the following:

- instantiate the `RDBHandler`
- call `initMsg()`
- call `addPackage()` for each package or set of packages that you want to make part of the message

Example:

```
    :
    Framework::RDBHandler myHandler;

    // start a new message
    myHandler.initMsg();

    // add the marker for the start of frame
    myHandler.addPackage( simTime, simFrame, RDB_PKG_ID_START_OF_FRAME );

    // add extended package for a dynamic object's state
    RDB_OBJECT_STATE_t *objState = (RDB_OBJECT_STATE_t* ) myHandler.addPackage( simTime, simFrame, RDB_PKG_ID_OBJECT_STATE, 1,

    if ( !objState )
    {
        fprintf( stderr, "could not create object state\n" );
        return;
    }

    // fill the object state data
    objState->base.id = 1;
    :
    objState->base.pos.x = 1.0;
    :
    objState->ext.pos.x = 0.05;
    :

    // add the marker for the end of frame
    myHandler.addPackage( simTime, simFrame, RDB_PKG_ID_END_OF_FRAME );
    :
```

Now, if you want to send the message via network for example, you may easily retrieve it from the handler using the routines `getMsg()` (provides a pointer to the message) and `getMsgTotalSize()` (providing the total size of the message in `[bytes]`).

Example:

```
  int retVal = send( mClient, ( const char* ) ( myHandler.getMsg() ), myHandler.getMsgTotalSize(), 0 );
```

Analyzing Messages

After having received a message, e.g. from network, you may want to analyze or de-compose it. The easiest way is to print the message. For this, use the static method `printMessage(...)` and provide it with

- the pointer to the message
- with an indication whether to print the details of the message (`true`) or the package types only (`false`).

Example:

```
    Framework::RDBHandler::printMessage( ( RDB_MSG_t* ) pData, true );
```

By deriving your own class from the class `Framework::RDBHandler` and implementing the applicable callbacks of type `parseEntry(...)`, you may handle a message entry by entry. The actual parsing of the message is triggered using the RDBHandler's method `parseMessage(...)`

Managing Shared Memory

The RDBHandler will also help you managing shared memory segments which are supposed to hold RDB messages. Affter having determined the pointer to the start of an SHM segment and its size, just call `shmConfigure(...)` in order to link an instance of the RDBHandler with the SHM segment:

Example:

```
    // allocate a single buffer within the shared memory segment
    mIgCtrlRdbHandler.shmConfigure( mIgCtrlShmPtr, 1, mIgCtrlShmTotalSize );

    // for double-buffering, replace the "1" with "2"
```

A shared memory segment is **initialized** in the following way:

- get access to the buffer's info structure
- reset all flags (only if nobody else has locked the buffer)
- clear the buffer content (make sure nobody else is accessing it at the same time)

Example:

```
    RDB_SHM_BUFFER_INFO_t* info = mIgCtrlRdbHandler.shmBufferGetInfo( 0 );    // 0 = first buffer, 1 = second buffer (in doubl

    if ( !info )
        return 0;

    // force all flags to be zero
    info->flags = 0;

    // clear the buffer before writing to it (otherwise messages will accumulate)
    if ( !mIgCtrlRdbHandler.shmBufferClear( 0, true ) )   // true = clearing will be forced; not recommended!
        return 0;
```

A message is **written** to shared memory using the following sequence:

- get access to the buffer's administration structure
- make sure the buffer is not locked
- lock the buffer
- clear the buffer
- compose your message
- map the message to the shared memory
- set any flags for the consumers of the message
- unlock the buffer

Example (write a sync message into IG's control SHM):

```
    // get access to the administration information of the first RDB buffer in SHM
    RDB_SHM_BUFFER_INFO_t* info = mIgCtrlRdbHandler.shmBufferGetInfo( 0 );

    if ( !info )
        return 0;

    // is the buffer ready for write?
    if ( info->flags )      // is the buffer accessible (flags == 0)?
        return 0;

    // clear the buffer before writing to it (otherwise messages will accumulate)
    if ( !mIgCtrlRdbHandler.shmBufferClear( 0 ) )
        return 0;

    // lock immediately after clearing the buffer
    mIgCtrlRdbHandler.shmBufferLock( 0 );

    // initialize the message, so it does not continually grow
    mIgCtrlRdbHandler.initMsg();
    // create a message containing the sync information
    RDB_SYNC_t* syncData = ( RDB_SYNC_t* ) mIgCtrlRdbHandler.addPackage( mFrameNo * mFrameTime, mFrameNo, RDB_PKG_ID_SYNC );

    if ( !syncData )
    {
        mIgCtrlRdbHandler.shmBufferRelease( 0 );
        fprintf( stderr, "..failed\n" );
        return 0;
    }

    syncData->mask    = 0x0;
    syncData->cmdMask = RDB_SYNC_CMD_RENDER_SINGLE_FRAME;

    // set some information concerning the RDB buffer itself
    info->id    = 1;
    info->flags = RDB_SHM_BUFFER_FLAG_IG;

    // now copy the sync message to the first RDB buffer in SHM
    mIgCtrlRdbHandler.mapMsgToShm( 0 );

    // release the buffer
    mIgCtrlRdbHandler.shmBufferRelease( 0 );
```

In order to **read** data from the SHM, you will have to

- access the corresponding segment
- find the relevant buffer (in double-buffered mode, otherwise use buffer 0)
- make sure it's marked ready-for-read but not locked
- lock the buffer
- copy or analyze the message
- clear your read flags
- release the buffer

Once you have access to the message, you may use the RDBHandler as describe above for the analyzing task.

Sample Client - Tool independent

Network Connection via TCP/IP

The following source code contains a sample client that is *typical* for a connection between VTD and an external vehicle dynamics. It is part of a package which contains also an SCP client in addition to various RDB sample implementations:

sampleClientsRDBandSCP.tgz

For the sample RDB client, you will only need the following files:

```
Communication
├── Common
│   ├── RDBHandler.cc           ...... RDB helper class, implementation
│   ├── RDBHandler.hh           ...... RDB helper class, interface file
│   ├── viRDBIcd.h              ...... RDB interface definition
├── RDBClientSample
│   ├── compileVehDynInteg.sh  ...... compile script
│   └── ExampleVehDynInteg.cpp ...... implementation
```

It allows you to connect to the TaskControl's server port, wait for an incoming DRIVER_CTRL message, compute your vehicle dynamics output and send the result to the TaskControl. This example is *frame synchronous* and, therefore, the vehicle dynamics will not run faster than VTD's main loop. If you wish to compute your vehicle dynamics data asynchronous, then just buffer the incoming DRIVER_CTRL commands and compute your vehicle dynamics in a separate loop.

Here's what you have to do to get the example running:

- unpack the archive
- cd into `RDBClientSample`
- run the compile script `compileVehDynInteg.sh`
- start VTD and make sure you are requesting driver control commands (e.g. by adding the action `<Player name="$owner"><Driver ctrlLatLong="ghostdriver"/></Player>` to your own player in the scenario)
- make sure your own vehicle has the ID no. 1 (or adjust the variable `sMyPlayerId` accordingly)
- stop the ModuleManager if it is running and contains its own vehicle dynamics plug-in
- start `./sampleVehDynRDB`
- your own vehicle will move along the x-axis with 5.0m/s now

The application `ExampleVehDynInteg.cpp` contains the following elements:

Port and network buffer definition:

```
#define DEFAULT_PORT        48190   /* for image port it should be 48192 */
#define DEFAULT_BUFFER      204800
```

Creation of client socket and blocking until connection has been established:

```
    //
    // Create the socket, and attempt to connect to the server
    //
    sClient = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );

    if ( sClient == -1 )
    {
        fprintf( stderr, "socket() failed: %s\n", strerror( errno ) );
        return 1;
    }

    // make this a fast socket
    int opt = 1;  // may have to be of type char in Windows
    setsockopt ( sClient, IPPROTO_TCP, TCP_NODELAY, &opt, sizeof( opt ) );

    server.sin_family      = AF_INET;
    server.sin_port        = htons(iPort);
    server.sin_addr.s_addr = inet_addr(szServer);

    //
    // If the supplied server address wasn't in the form
    // "aaa.bbb.ccc.ddd" it's a hostname, so try to resolve it
    //
    if ( server.sin_addr.s_addr == INADDR_NONE )
    {
        host = gethostbyname(szServer);
        if ( host == NULL )
        {
            fprintf( stderr, "Unable to resolve server: %s\n", szServer );
            return 1;
        }
        memcpy( &server.sin_addr, host->h_addr_list[0], host->h_length );
    }
    // wait for connection
    bool bConnected = false;

    while ( !bConnected )
    {
        if ( connect( sClient, (struct sockaddr *)&server, sizeof( server ) ) == -1 )
        {
            fprintf( stderr, "connect() failed: %s\n", strerror( errno ) );
            sleep( 1 );
        }
        else
            bConnected = true;
    }

    fprintf( stderr, "connected!\n" );
```

Endless loop for reading and parsing RDB data:

```
    unsigned int  bytesInBuffer = 0;
    size_t        bufferSize    = sizeof( RDB_MSG_HDR_t );
    unsigned int  count         = 0;
    unsigned char *pData        = ( unsigned char* ) calloc( 1, bufferSize );

    // Main loop, send and receive data - forever!
    //
    for(;;)
    {
        bool bMsgComplete = false;

        // make sure this is non-bloekcing

        int nrReady = getNoReadyRead( sClient ); // MUST BE ZERO in order to be non-blocking!!!

        if ( nrReady < 0 )
        {
            printf( "recv() failed: %s\n", strerror( errno ) );
            break;
        }

        //fprintf( stderr, "nrReady = %d\n", nrReady );

        if ( nrReady > 0 )
        {
            // read data
            ret = recv( sClient, szBuffer, DEFAULT_BUFFER, 0 );

            if ( ret != 0 )
            {
                // do we have to grow the buffer??
                if ( ( bytesInBuffer + ret ) > bufferSize )
                    pData = ( unsigned char* ) realloc( pData, bytesInBuffer + ret );
                memcpy( pData + bytesInBuffer, szBuffer, ret );
                bytesInBuffer += ret;

                // already complete messagae?
                if ( bytesInBuffer >= sizeof( RDB_MSG_HDR_t ) )
                {
                    RDB_MSG_HDR_t* hdr = ( RDB_MSG_HDR_t* ) pData;

                    // is this message containing the valid magic number?
                    if ( hdr->magicNo != RDB_MAGIC_NO )
                    {
                        printf( "message receiving is out of sync; discarding data" );
                        bytesInBuffer = 0;
                    }

                    // handle all messages in the buffer before proceeding
                    while ( bytesInBuffer >= ( hdr->headerSize + hdr->dataSize ) )
                    {
                        unsigned int msgSize = hdr->headerSize + hdr->dataSize;
                        bool         isImage = false;

                        // print the message
                        if ( sVerbose )
                            Framework::RDBHandler::printMessage( ( RDB_MSG_t* ) pData, true );

                        // now parse the message
                        parseRDBMessage( ( RDB_MSG_t* ) pData, isImage );

                        // remove message from queue
                        memmove( pData, pData + msgSize, bytesInBuffer - msgSize );
                        bytesInBuffer -= msgSize;
                    }
                }
            }
        }
        // do some other stuff before returning to network reading

        // don't use the processor excessively
        usleep( 10 );
    }
```

Type-specific callbacks for various message contents (here, restricted to two package types):

```
void parseRDBMessageEntry( const double & simTime, const unsigned int & simFrame, RDB_MSG_ENTRY_HDR_t* entryHdr )
{
    if ( !entryHdr )
        return;

    int noElements = entryHdr->elementSize ? ( entryHdr->dataSize / entryHdr->elementSize ) : 0;

    unsigned char ident   = 6;
    char*         dataPtr = ( char* ) entryHdr;

    dataPtr += entryHdr->headerSize;

    while ( noElements-- )      // only two types of messages are handled here
    {
        switch ( entryHdr->pkgId )
        {
            case RDB_PKG_ID_OBJECT_STATE:
                handleRDBitem( simTime, simFrame, *( ( RDB_OBJECT_STATE_t* ) dataPtr ), entryHdr->flags & RDB_PKG_FLAG_EXTENDE
                break;

            case RDB_PKG_ID_DRIVER_CTRL:
                handleRDBitem( simTime, simFrame, *( ( RDB_DRIVER_CTRL_t* ) dataPtr ) );
                break;

            default:
                break;
        }
        dataPtr += entryHdr->elementSize;
     }
}
```

Callback for incoming driver control commands and calculation of new vehicle dynamics state:

```
/**
* handle driver control input and compute vehicle dynamics output
*/
void handleRDBitem( const double & simTime, const unsigned int & simFrame, RDB_DRIVER_CTRL_t & item )
{
    static bool         sVerbose     = true;
    static bool         sShowMessage = false;
    static unsigned int sMyPlayerId  = 1;             // this may also be determined from incoming OBJECT_CFG messages
    static double       sLastSimTime = -1.0;

    fprintf( stderr, "handleRDBitem: handling driver control for player %d\n", item.playerId );

    // is this a new message?
    //if ( simTime == sLastSimTime )
    //    return;

    // is this message for me?
    if ( item.playerId != sMyPlayerId )
        return;

    // check for valid inputs (only some may be valid)
    float mdSteeringAngleRequest = ( item.validityFlags & RDB_DRIVER_INPUT_VALIDITY_STEERING_WHEEL ) ? item.steeringWheel / 19
    float mdThrottlePedal        = ( item.validityFlags & RDB_DRIVER_INPUT_VALIDITY_THROTTLE )       ? item.throttlePedal
    float mdBrakePedal           = ( item.validityFlags & RDB_DRIVER_INPUT_VALIDITY_BRAKE )          ? item.brakePedal
    float mInputAccel            = ( item.validityFlags & RDB_DRIVER_INPUT_VALIDITY_TGT_ACCEL )      ? item.accelTgt
    float mInputSteering         = ( item.validityFlags & RDB_DRIVER_INPUT_VALIDITY_TGT_STEERING )   ? item.steeringTgt
    float mdSteeringRequest      = ( item.validityFlags & RDB_DRIVER_INPUT_VALIDITY_TGT_STEERING )   ? item.steeringTgt
    float mdAccRequest           = ( item.validityFlags & RDB_DRIVER_INPUT_VALIDITY_TGT_ACCEL )      ? item.accelTgt
    int   mInputGear             = 0;

    // check the input validity
    unsigned int validFlagsLat  = RDB_DRIVER_INPUT_VALIDITY_TGT_STEERING | RDB_DRIVER_INPUT_VALIDITY_STEERING_WHEEL;
    unsigned int validFlagsLong = RDB_DRIVER_INPUT_VALIDITY_TGT_ACCEL | RDB_DRIVER_INPUT_VALIDITY_THROTTLE | RDB_DRIVER_INPUT_\
    unsigned int checkFlags     = item.validityFlags & 0x00000fff;

    if ( checkFlags )
    {
        if ( ( checkFlags & validFlagsLat ) && ( checkFlags & validFlagsLong ) )
            sShowMessage = false;
        else if ( checkFlags != RDB_DRIVER_INPUT_VALIDITY_GEAR ) // "gear only" is also fine
        {
            if ( !sShowMessage )
                fprintf( stderr, "Invalid driver input for vehicle dynamics" );

            sShowMessage = true;
        }
    }

    // use pedals/wheel or targets?
    bool mUseSteeringTarget = ( ( item.validityFlags & RDB_DRIVER_INPUT_VALIDITY_TGT_STEERING ) != 0 );
    bool mUseAccelTarget    = ( ( item.validityFlags & RDB_DRIVER_INPUT_VALIDITY_TGT_ACCEL ) != 0 );

    if ( item.validityFlags & RDB_DRIVER_INPUT_VALIDITY_GEAR )
    {
        if ( item.gear == RDB_GEAR_BOX_POS_R )
            mInputGear = -1;
        else if ( item.gear == RDB_GEAR_BOX_POS_N )
            mInputGear = 0;
        else if ( item.gear == RDB_GEAR_BOX_POS_D )
            mInputGear = 1;
        else
            mInputGear = 1;
    }

    // now, depending on the inputs, select the control mode and compute outputs
    if ( mUseSteeringTarget && mUseAccelTarget )
    {
        fprintf( stderr, "Compute new vehicle position from acceleration target and steering target.\n" );

        // call your methods here
    }
    else if ( !mUseSteeringTarget && !mUseAccelTarget )
    {
        fprintf( stderr, "Compute new vehicle position from brake pedal, throttle pedal and steering wheel angle.\n" );

        // call your methods here
    }
    else
    {
        fprintf( stderr, "Compute new vehicle position from a mix of targets and pedals / steering wheel angle.\n" );

        // call your methods here
    }

    bool useDummy = true;

    // the following assignments are for dummy purposes only
    // vehicle moves along x-axis with given speed
    // ignore first message
    if ( useDummy && ( sLastSimTime >= 0.0 ) )
    {
        double speedX = 5.0;    // m/s
        double speedY = 0.0;    // m/s
        double speedZ = 0.0;    // m/s
        double dt     = simTime - sLastSimTime;

        sOwnObjectState.base.id       = sMyPlayerId;
        sOwnObjectState.base.category = RDB_OBJECT_CATEGORY_PLAYER;
        sOwnObjectState.base.type     = RDB_OBJECT_TYPE_PLAYER_CAR;
        strcpy( sOwnObjectState.base.name, "Ego" );

        // dimensions of own vehicle
        sOwnObjectState.base.geo.dimX = 4.60;
        sOwnObjectState.base.geo.dimY = 1.86;
        sOwnObjectState.base.geo.dimZ = 1.60;

        // offset between reference point and center of geometry
        sOwnObjectState.base.geo.offX = 0.80;
        sOwnObjectState.base.geo.offY = 0.00;
        sOwnObjectState.base.geo.offZ = 0.30;

        sOwnObjectState.base.pos.x     += dt * speedX;
        sOwnObjectState.base.pos.y     += dt * speedY;
        sOwnObjectState.base.pos.z     += dt * speedZ;
        sOwnObjectState.base.pos.h     = 0.0;
        sOwnObjectState.base.pos.p     = 0.0;
        sOwnObjectState.base.pos.r     = 0.0;
        sOwnObjectState.base.pos.flags = RDB_COORD_FLAG_POINT_VALID | RDB_COORD_FLAG_ANGLES_VALID;

        sOwnObjectState.ext.speed.x     = speedX;
        sOwnObjectState.ext.speed.y     = speedY;
        sOwnObjectState.ext.speed.z     = speedZ;
        sOwnObjectState.ext.speed.h     = 0.0;
        sOwnObjectState.ext.speed.p     = 0.0;
        sOwnObjectState.ext.speed.r     = 0.0;
        sOwnObjectState.ext.speed.flags = RDB_COORD_FLAG_POINT_VALID | RDB_COORD_FLAG_ANGLES_VALID;

        sOwnObjectState.ext.accel.x     = 0.0;
        sOwnObjectState.ext.accel.y     = 0.0;
        sOwnObjectState.ext.accel.z     = 0.0;
        sOwnObjectState.ext.accel.flags = RDB_COORD_FLAG_POINT_VALID;

        sOwnObjectState.base.visMask    =  RDB_OBJECT_VIS_FLAG_TRAFFIC | RDB_OBJECT_VIS_FLAG_RECORDER;
    }

    // ok, I have a new object state, so let's send the data
    sendOwnObjectState( sClient, simTime, simFrame );

    // remember last simulation time
    sLastSimTime = simTime;
}
```

Transmit of new vehicle dynamics state:

```
void sendOwnObjectState( int & sendSocket, const double & simTime, const unsigned int & simFrame )
{
    Framework::RDBHandler myHandler;

    // start a new message
    myHandler.initMsg();

    // begin with an SOF identifier
    myHandler.addPackage( simTime, simFrame, RDB_PKG_ID_START_OF_FRAME );

    // add extended package for the object state
    RDB_OBJECT_STATE_t *objState = ( RDB_OBJECT_STATE_t* ) myHandler.addPackage( simTime, simFrame, RDB_PKG_ID_OBJECT_STATE, 1

    if ( !objState )
    {
        fprintf( stderr, "sendOwnObjectState: could not create object state\n" );
        return;
    }

    // copy contents of internally held object state to output structure
    memcpy( objState, &sOwnObjectState, sizeof( RDB_OBJECT_STATE_t ) );

    fprintf( stderr, "sendOwnObjectState: sending pos x/y/z = %.3lf/%.3lf/%.3lf,\n", objState->base.pos.x, objState->base.pos.

    // terminate with an EOF identifier
    myHandler.addPackage( simTime, simFrame, RDB_PKG_ID_END_OF_FRAME );

    int retVal = send( sendSocket, ( const char* ) ( myHandler.getMsg() ), myHandler.getMsgTotalSize(), 0 );

    if ( !retVal )
        fprintf( stderr, "sendOwnObjectState: could not send object state\n" );
}
```

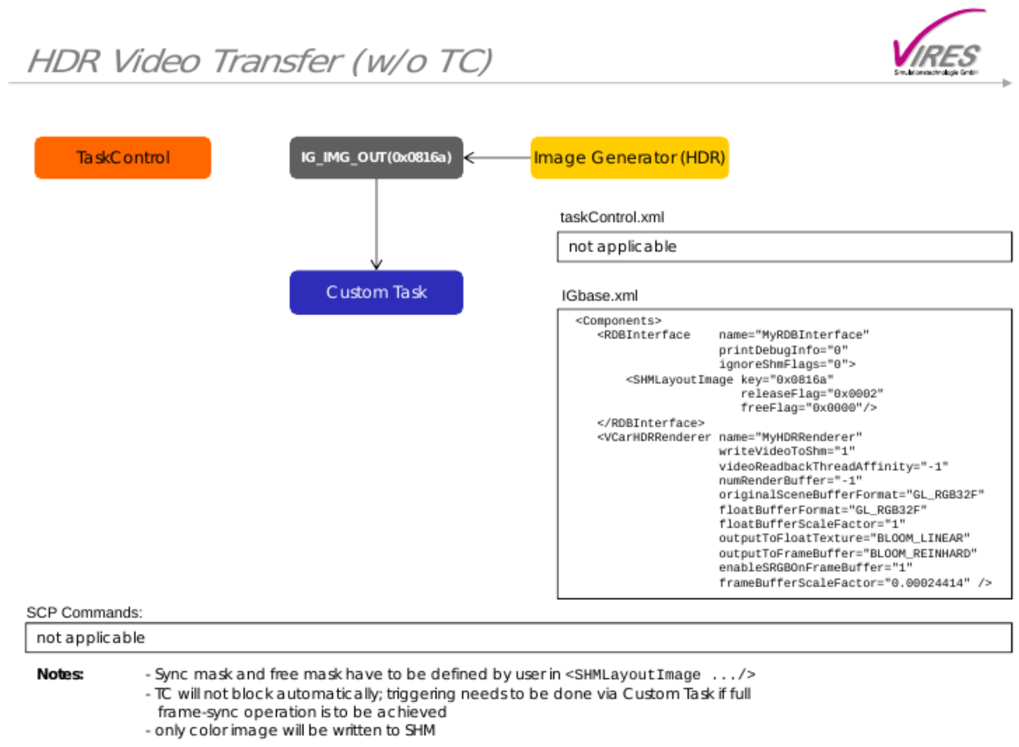
RDB Data Transfer (Receiver Side) via Shared Memory

The following source code may be used for reading RDB data (e.g. image data) from a shared memory interface (e.g. Image Generator's image output SHM segment). For further information about the shared memory and its layout in VTD, please look at

FiguresManualO_SHM.pdf

or follow the documentation of video data transfer in our wiki at Video Image Transfer via SHM

For our example, the connection of the custom task (here: reader routine) and the IG is assumed to be as follows:



```
<Components>
   <RDBInterface    name="MyRDBInterface"
                    printDebugInfo="0"
                    ignoreShmFlags="0">
       <SHMLayoutImage key="0x0816a"
                       releaseFlag="0x0002"
                       freeFlag="0x0000"/>
   </RDBInterface>
   <VCarHDRRenderer name="MyHDRRenderer"
                    writeVideoToShm="1"
                    videoReadbackThreadAffinity="-1"
                    numRenderBuffer="-1"
                    originalSceneBufferFormat="GL_RGB32F"
                    floatBufferFormat="GL_RGB32F"
                    floatBufferScaleFactor="1"
                    outputToFloatTexture="BLOOM_LINEAR"
                    outputToFrameBuffer="BLOOM_REINHARD"
                    enableSRGBOnFrameBuffer="1"
                    frameBufferScaleFactor="0.00024414" />
```

Notes:
- Sync mask and free mask have to be defined by user in `<SHMLayoutImage .../>`
- TC will not block automatically; triggering needs to be done via Custom Task if full frame-sync operation is to be achieved
- only color image will be written to SHM

The reader routine is organized as follows:

- open the shared memory segment identified by the key given as command line argument *-k:key*
- make sure the SHM is double buffered (you may, of course, modify this query if you can work with a single buffer)
- infinitely loop over the following steps
  - access both buffers and make sure at least one of them is ready for read
  - print some debug information about the buffers (if command line argument *-v* has been set)
  - read all messages in the buffer that has been identified as the source buffer; there may be more than one message in a buffer
  - process each message (here: print routine); this is to be implemented by the user

```
// ShmReader.cpp : Sample implementation of a process reading
// from a shared memory segment (double buffered) with RDB layout
// (c) 2016 by VIRES Simulationstechnologie GmbH
// Provided AS IS without any warranty!
//

#include <stdlib.h>
#include <stdio.h>
#include <sys/shm.h>
#include <string.h>
#include <unistd.h>
#include "RDBHandler.hh"

// forward declarations of methods

/**
* method for checking the contents of the SHM
*/
int  checkShm();
void openShm();

/**
* routine for handling an RDB message; to be provided by user;
* here, only a printing of the message is performed
* @param msg    pointer to the message that is to be handled
*/
void handleMessage( RDB_MSG_t* msg );

/**
* some global variables, considered "members" of this example
*/
unsigned int mShmKey       = RDB_SHM_ID_IMG_GENERATOR_OUT;      // key of the SHM segment
unsigned int mCheckMask    = RDB_SHM_BUFFER_FLAG_TC;
void*        mShmPtr       = 0;                                 // pointer to the SHM segment
size_t       mShmTotalSize = 0;                                 // remember the total size of the SHM segment
bool         mVerbose      = false;                             // run in verbose mode?
int          mForceBuffer  = -1;                                // force reading one of the SHM buffers (0=A, 1=B)

/**
* information about usage of the software
* this method will exit the program
*/
void usage()
{
    printf("usage: shmReader [-k:key] [-c:checkMask] [-v] [-f:bufferId]\n\n");
    printf("       -k:key        SHM key that is to be addressed\n");
    printf("       -c:checkMask  mask against which to check before reading an SHM buffer\n");
    printf("       -f:bufferId   force reading of a given buffer (0 or 1) instead of checking for a valid checkMask\n");
    printf("       -v            run in verbose mode\n");
    exit(1);
}

/**
* validate the arguments given in the command line
*/
void ValidateArgs(int argc, char **argv)
{
    for( int i = 1; i < argc; i++)
    {
        if ((argv[i][0] == '-') || (argv[i][0] == '/'))
        {
            switch (tolower(argv[i][1]))
            {
                case 'k':        // shared memory key
                    if ( strlen( argv[i] ) > 3 )
                        mShmKey = atoi( &argv[i][3] );
                    break;

                case 'c':       // check mask
                    if ( strlen( argv[i] ) > 3 )
                        mCheckMask = atoi( &argv[i][3] );
                    break;

                case 'f':       // force reading a given buffer
                    if ( strlen( argv[i] ) > 3 )
                        mForceBuffer = atoi( &argv[i][3] );
                    break;

                case 'v':       // verbose mode
                    mVerbose = true;
                    break;

                default:
                    usage();
                    break;
            }
        }
    }

    fprintf( stderr, "ValidateArgs: key = 0x%x, checkMask = 0x%x, mForceBuffer = %d\n",
                     mShmKey, mCheckMask, mForceBuffer );
}

/**
* main program with high frequency loop for checking the shared memory;
* does nothing else
*/

int main(int argc, char* argv[])
{
    // Parse the command line
    //
    ValidateArgs(argc, argv);

    // first: open the shared memory (try to attach without creating a new segment)

    fprintf( stderr, "attaching to shared memory....\n" );

    while ( !mShmPtr )
    {
        openShm();
        usleep( 100000 );     // do not overload the CPU
    }

    fprintf( stderr, "...attached! Reading now...\n" );

    // now check the SHM for the time being
    while ( 1 )
    {
        checkShm();

        usleep( 1000 );
    }
}

/**
* open the shared memory segment
*/
void openShm()
{
    // do not open twice!
    if ( mShmPtr )
        return;

    int shmid = 0;

    if ( ( shmid = shmget( mShmKey, 0, 0 ) ) < 0 )
        return;

    if ( ( mShmPtr = (char *)shmat( shmid, (char *)0, 0 ) ) == (char *) -1 )
    {
        perror("openShm: shmat()");
        mShmPtr = 0;
    }

    if ( mShmPtr )
    {
        struct shmid_ds sInfo;

        if ( ( shmid = shmctl( shmid, IPC_STAT, &sInfo ) ) < 0 )
            perror( "openShm: shmctl()" );
        else
            mShmTotalSize = sInfo.shm_segsz;
    }
}

int checkShm()
{
    if ( !mShmPtr )
        return 0;

    // get a pointer to the shm info block
    RDB_SHM_HDR_t* shmHdr = ( RDB_SHM_HDR_t* ) ( mShmPtr );

    if ( !shmHdr )
        return 0;

    if ( ( shmHdr->noBuffers != 2 ) )
    {
        fprintf( stderr, "checkShm: no or wrong number of buffers in shared memory. I need two buffers!" );
        return 0;
    }

    // allocate space for the buffer infos
    RDB_SHM_BUFFER_INFO_t** pBufferInfo = ( RDB_SHM_BUFFER_INFO_t** ) ( new char [ shmHdr->noBuffers * sizeof( RDB_SHM_BUFFER_
    RDB_SHM_BUFFER_INFO_t*  pCurrentBufferInfo = 0;

    char* dataPtr = ( char* ) shmHdr;
    dataPtr += shmHdr->headerSize;

    for ( int i = 0; i < shmHdr->noBuffers; i++ )
    {
        pBufferInfo[ i ] = ( RDB_SHM_BUFFER_INFO_t* ) dataPtr;
        dataPtr += pBufferInfo[ i ]->thisSize;
    }

    // get the pointers to message section in each buffer
    RDB_MSG_t* pRdbMsgA = ( RDB_MSG_t* ) ( ( ( char* ) mShmPtr ) + pBufferInfo[0]->offset );
    RDB_MSG_t* pRdbMsgB = ( RDB_MSG_t* ) ( ( ( char* ) mShmPtr ) + pBufferInfo[1]->offset );

    // pointer to the message that will actually be read
    RDB_MSG_t* pRdbMsg  = 0;

    // remember the flags that are set for each buffer
```

```
unsigned int flagsA = pBufferInfo[ 0 ]->flags;
unsigned int flagsB = pBufferInfo[ 1 ]->flags;

// check whether any buffer is ready for reading (checkMask is set (or 0) and buffer is NOT locked)
bool readyForReadA = ( ( flagsA & mCheckMask ) || !mCheckMask ) && !( flagsA & RDB_SHM_BUFFER_FLAG_LOCK );
bool readyForReadB = ( ( flagsB & mCheckMask ) || !mCheckMask ) && !( flagsB & RDB_SHM_BUFFER_FLAG_LOCK );

if ( mVerbose )
{
    fprintf( stderr, "ImageReader::checkShm: before processing SHM\n" );
    fprintf( stderr, "ImageReader::checkShm: Buffer A: frameNo = %06d, flags = 0x%x, locked = <%s>, lock mask set = <%s>,
                     pRdbMsgA->hdr.frameNo,
                     flagsA,
                     ( flagsA & RDB_SHM_BUFFER_FLAG_LOCK ) ? "true" : "false",
                     ( flagsA & mCheckMask ) ? "true" : "false",
                     readyForReadA ?  "true" : "false" );

    fprintf( stderr, "                       Buffer B: frameNo = %06d, flags = 0x%x, locked = <%s>, lock mask set = <%s>,
                     pRdbMsgB->hdr.frameNo,
                     flagsB,
                     ( flagsB & RDB_SHM_BUFFER_FLAG_LOCK ) ? "true" : "false",
                     ( flagsB & mCheckMask ) ? "true" : "false",
                     readyForReadB ?  "true" : "false" );
}

if ( mForceBuffer < 0 )  // auto-select the buffer if none is forced to be read
{
    // check which buffer to read
    if ( ( readyForReadA ) && ( readyForReadB ) )
    {
        if ( pRdbMsgA->hdr.frameNo > pRdbMsgB->hdr.frameNo )        // force using the latest image!!
        {
            pRdbMsg            = pRdbMsgA;
            pCurrentBufferInfo = pBufferInfo[ 0 ];
        }
        else
        {
            pRdbMsg            = pRdbMsgB;
            pCurrentBufferInfo = pBufferInfo[ 1 ];
        }
    }
    else if ( readyForReadA )
    {
        pRdbMsg            = pRdbMsgA;
        pCurrentBufferInfo = pBufferInfo[ 0 ];
    }
    else if ( readyForReadB )
    {
        pRdbMsg            = pRdbMsgB;
        pCurrentBufferInfo = pBufferInfo[ 1 ];
    }
}
else if ( ( mForceBuffer == 0 ) && readyForReadA )   // force reading buffer A
{
    pRdbMsg            = pRdbMsgA;
    pCurrentBufferInfo = pBufferInfo[ 0 ];
}
else if ( ( mForceBuffer == 1 ) && readyForReadB ) // force reading buffer B
{
    pRdbMsg            = pRdbMsgB;
    pCurrentBufferInfo = pBufferInfo[ 1 ];
}

// lock the buffer that will be processed now (by this, no other process will alter the contents)
if ( pCurrentBufferInfo )
    pCurrentBufferInfo->flags |= RDB_SHM_BUFFER_FLAG_LOCK;

// no data available?
if ( !pRdbMsg || !pCurrentBufferInfo )
{
    delete pBufferInfo;
    pBufferInfo = 0;

    // return with valid result if simulation is not yet running
    if ( ( pRdbMsgA->hdr.frameNo == 0 ) && ( pRdbMsgB->hdr.frameNo == 0 ) )
        return 1;

    // otherwise return a failure
    return 0;
}

// handle all messages in the buffer
if ( !pRdbMsg->hdr.dataSize )
{
    fprintf( stderr, "checkShm: zero message data size, error.\n" );
```

```
        return 0;
    }

    unsigned int maxReadSize = pCurrentBufferInfo->bufferSize;

    while ( 1 )
    {
        // handle the message that is contained in the buffer; this method should be provided by the user (i.e. YOU!)
        handleMessage( pRdbMsg );

        // do not read more bytes than there are in the buffer (avoid reading a following buffer accidentally)
        maxReadSize -= pRdbMsg->hdr.dataSize + pRdbMsg->hdr.headerSize;

        if ( maxReadSize < ( pRdbMsg->hdr.headerSize + pRdbMsg->entryHdr.headerSize ) )
            break;

        // go to the next message (if available); there may be more than one message in an SHM buffer!
        pRdbMsg = ( RDB_MSG_t* ) ( ( ( char* ) pRdbMsg ) + pRdbMsg->hdr.dataSize + pRdbMsg->hdr.headerSize );

        if ( !pRdbMsg )
            break;

        if ( pRdbMsg->hdr.magicNo != RDB_MAGIC_NO )
            break;
    }

    // release after reading
    pCurrentBufferInfo->flags &= ~mCheckMask;                   // remove the check mask
    pCurrentBufferInfo->flags &= ~RDB_SHM_BUFFER_FLAG_LOCK;     // remove the lock mask

    if ( mVerbose )
    {
        unsigned int flagsA = pBufferInfo[ 0 ]->flags;
        unsigned int flagsB = pBufferInfo[ 1 ]->flags;

        fprintf( stderr, "ImageReader::checkShm: after processing SHM\n" );
        fprintf( stderr, "ImageReader::checkShm: Buffer A: frameNo = %06d, flags = 0x%x, locked = <%s>, lock mask set = <%s>\n
                         pRdbMsgA->hdr.frameNo,
                         flagsA,
                         ( flagsA & RDB_SHM_BUFFER_FLAG_LOCK ) ? "true" : "false",
                         ( flagsA & mCheckMask ) ? "true" : "false" );
        fprintf( stderr, "                       Buffer B: frameNo = %06d, flags = 0x%x, locked = <%s>, lock mask set = <%s>.\
                         pRdbMsgB->hdr.frameNo,
                         flagsB,
                         ( flagsB & RDB_SHM_BUFFER_FLAG_LOCK ) ? "true" : "false",
                         ( flagsB & mCheckMask ) ? "true" : "false" );
    }

    return 1;
}

void handleMessage( RDB_MSG_t* msg )
{
    // just print the message
    Framework::RDBHandler::printMessage( msg );
}
```

The source code is also available at the following link:

sampleShmReaderWriter.tgz

The programming sample may be used as follows:

- download the archive (see above)
- unpack the archive
- go to the sub-dir `RDBShmSample/`
- execute `./compile.sh`
- start the application that provides the data in its shared memory segment (e.g. VTD's image generator).
- execute the compiled programming sample as follows:
  - `./shmReader -c:0` (setting the check mask to 0 will enable reading of any buffer)

The typical output when connecting to the SHM of the ImageGenerator (key 0x0816a) will be similar to:

```
RDBHandler::printMessage: ---- short info ----- BEGIN
  message: version = 0x0117, simTime = 12.066, simFrame = 723, headerSize = 24, dataSize = 3537838
    entry: pkgId = 22 (RDB_PKG_ID_IMAGE), headersize = 16, dataSize = 1516118, elementSize = 1516118, noElements = 1, flags =
    entry: pkgId = 18 (RDB_PKG_ID_CAMERA), headersize = 16, dataSize = 88, elementSize = 88, noElements = 1, flags = 0x0
    entry: pkgId = 22 (RDB_PKG_ID_IMAGE), headersize = 16, dataSize = 2021480, elementSize = 2021480, noElements = 1, flags =
    entry: pkgId = 18 (RDB_PKG_ID_CAMERA), headersize = 16, dataSize = 88, elementSize = 88, noElements = 1, flags = 0x0
RDBHandler::printMessage: ---- short info ----- END
```

Parsing a VTD Data Recording

VTD's data recording format has been changed with VTD 2.x. It is now a mere dump of RDB and SCP packages. Reading this dump is shown in the programming example provided here.

version: VTD.2.x

package: vtd.2.1.0.addOns.fileReader.20180510.tgz

operation:

- unpack the package
- cd into "VTD.2.1/Develop/Communication/RDBFileReader"
- execute "./compile.sh"
- execute "./fileReader testRec_000.dat"
- a dump of selected data elements of the recording will be displayed in the terminal
- adjust as necessary

Sample Client - Interface Routines Only

With the code in the following package, you are able to connect your own network receivers / senders (e.g. UDP socket, TCP/IP socket) and code / decode RDB messages. The main application for these routines is in a case where

- you are taking care of network reading / writing
- you are within a real-time platform
- your compiler environment is pure C (i.e. no C++)

The following files are included in the package:

```
├── Common .................................collection of include files used by the actual routines
│   ├── rdbMapper.c
│   ├── viRDBIcd.h
│   └── viRDBTypes.h
├── GenericPlatform
│   ├── CustomBlockRDB_FromVTD.Contact
│   │   ├── vtdGatewayRxCp.c ................routines for de-coding contact point messages from VTD
│   │   └── vtdGatewayRxCp.h
│   ├── CustomBlockRDB_FromVTD.Dyn
│   │   ├── vtdGatewayRxDyn.c ...............routines for de-coding object state and driver control messages from VTD
│   │   └── vtdGatewayRxDyn.h
│   ├── CustomBlockRDB_ToVTD.Contact
│   │   ├── vtdGatewayTxCp.c ................routines for creating road queries (for contact points) to VTD
│   │   └── vtdGatewayTxCp.h
│   └── CustomBlockRDB_ToVTD.Dyn
│       ├── vtdGatewayTxDyn.c ...............routines for sending the own vehicle's state to VTD
│       └── vtdGatewayTxDyn.h
```

Example for receiving contact point data (pseudo code, *CustomBlockRDB_FromVTD.Contact*):

```
vtdGatewayRxCpInitialize();

main loop
{
  dataReceived= readNetwork(); // to be done by user

  if ( data available )
  {
    vtdGatewayCalcOutputs( receiveFrameNo++, noBytesReceived, &dataReceived );

    // get general information about received data
    RX_CP_GENERAL_t* genInfo = vtdGatewayRxCpGetGeneralInfo();
    simTime  = genInfo->simTime;
    simFrame = genInfo->simFrame;

    // get contact point data
    RX_CP_CONTACT_POINT_t* cpInfo = vtdGatewayRxCpGetContactPoints();

    for ( unsigned int i = 0; i < cpInfo->noValidElements; i++ )
    {
        // do something with the contact point information
        zAvg += cpInfo->data[i].z;
    }
  }
}
```

Example for sending contact queries (pseudo code, *CustomBlockRDB_ToVTD.Contact*):

```
vtdGatewayTxCpInitialize();

main loop
{
  vtdGatewayTxCpSetMsgHdr( sendFrameNo++, mySimTime );

  // set the actual contact point locations
  vtdGatewayTxCpSetCpData( 0, 53, vehicleX,  0.9 );
  vtdGatewayTxCpSetCpData( 1, 27, vehicleX, -0.9 );
  vtdGatewayTxCpSetCpData( 2, 12, vehicleX+wheelBase,  0.9 );
  vtdGatewayTxCpSetCpData( 3, 15, vehicleX+wheelBase, -0.9 );

  // get the resulting message and its size
  unsigned int noBytes = 0;

  void* msgData = vtdGatewayTxCpGetMessage( &noBytes );

  if ( msgData && noBytes )
    sendDataToNetwork();    // by user's routines
}
```

Here's the archive containing all of the routines:

sampleGenericPlatformRDB.tgz

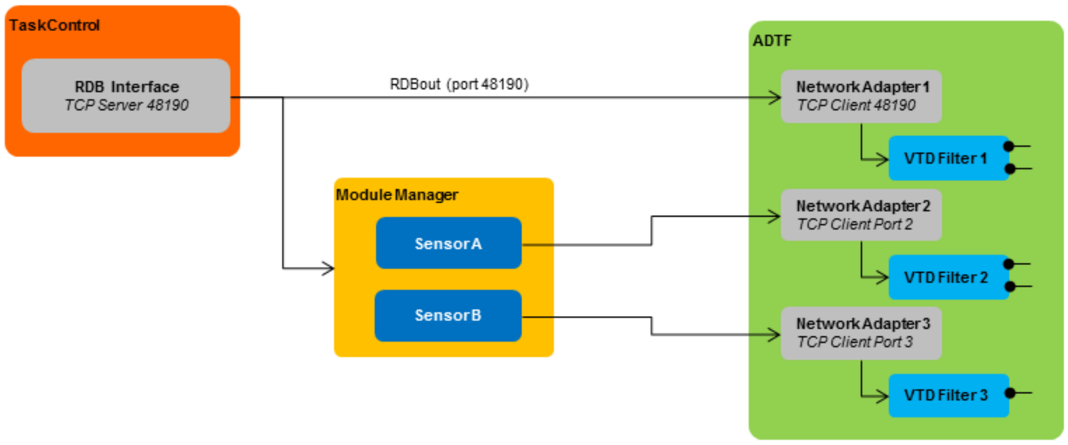
Sample Client - ADTF

For connecting RDB to ADTF, the VTD distribution contains a sample filter which may be adapted by the user. It is located in *Develop/VTD2ADTF*.

Note: this sample filter is not part of each distribution since some users of VTD already have their own, proprietary solutions. In no case do we want to interfere with these solutions.

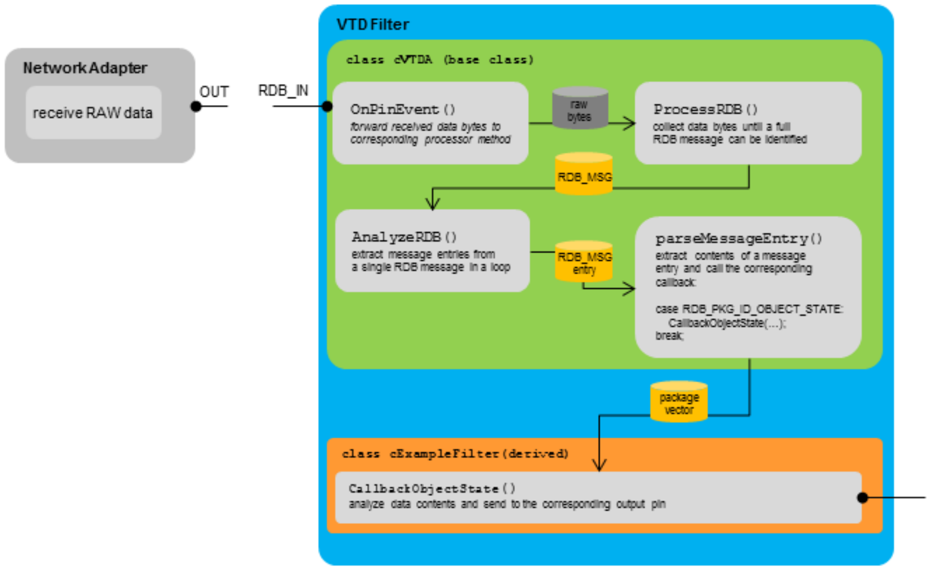
Overview

The following figure gives an overview of the data flow:



The filter may collect data from the TaskControl's main output (typically port 48190) or from any sensor output (ports according to user configuration). The filter itself does not provide a network interface, so it must be connected to the output of an ADTF network filter. For each network connection, a separate network interface and filter shall be instantiated.

Within the filter, data packages are collected, tested for RDB content, and forwarded to callback routines for the respective RDB package ids. The source code is split between a library and a filter prototype. The library contains the basic functionality (receiving and interpreting packages in class cVTDA) and should usually not be modified. The filter prototype (class cExampleFilter) is an example of an implementation of an RDB filter within ADTF. Here, the callbacks for the different RDB package types must be implemented individually by the user (see next figure).


Network Adapter
receive RAW data
OUT
RDB_IN
VTD Filter
class cVTDA (base class)
OnPinEvent ()
forward received data bytes to corresponding processor method
raw bytes
ProcessRDB ()
collect data bytes until a full RDB message can be identified
RDB_MSG
AnalyzeRDB ()
extract message entries from a single RDB message in a loop
RDB_MSG entry
parseMessageEntry()
extract contents of a message entry and call the corresponding callback:
case RDB_PKG_ID_OBJECT_STATE:
CallbackObjectState(...);
break;
package vector
class cExampleFilter (derived)
CallbackObjectState ()
analyze data contents and send to the corresponding output pin

The code may be compiled using CMAKE or using the VisualStudio project that is provided with the example.

Files

The VTD2ADTF filter comes with the following files (incomplete list):

```
├── CMakeLists.txt
├── doc
│   ├── readme.txt  - general instructions
│   └── vtdCom.pdf  - PDF of this wiki page
├── src
│   ├── library
│   │   ├── CMakeLists.txt
│   │   ├── VTDA.cpp    - base class implementation
│   │   └── VTDA.h
│   └── prototype
│       ├── CMakeLists.txt
│       ├── filter.cpp  - user class, implement callbacks here
│       ├── filter.h
└── VisualStudio
    ├── Example.vcproj
    ├── Library.vcproj
    └── VTDAnbindung.sln - solution for VisualStudio
```

The callbacks that may be implemented by the user are listed in the file `VTDA.h` (see methods following the pattern `virtual tResult Callback...`).

Note that each callback is called individually for each element that is contained in an RDB entry (since RDB entries may contain arrays of data, not only single data items).

Compilation

1. Open a shell and go to the directory VTD2ADTF
2. Enter the command `ccmake .`
3. A console will open. Press `c`
4. Press `e` upon appearance of the error message
5. Specify the `ADTF_DIR` variable
    1. Select the line containing the variable `ADTF_DIR`
    2. Press `Enter`
    3. Adjust the variable `ADTF_DIR` to point to your `adtf` directory
    4. Press `Enter` again
6. Specify the `CMAKE_BUILD_TYPE` variable
    1. Select the line containing the variable `CMAKE_BUILD_TYPE`
    2. Press `Enter`
    3. Adjust the variable `CMAKE_BUILD_TYPE` (set it to `debug` or `release`)
    4. Press `Enter` again
7. Press `c` for re-configuration
8. Press `g`
9. A `Makefile` will have been generated; now you can call it by entering `make`

Sample Client - Simulink

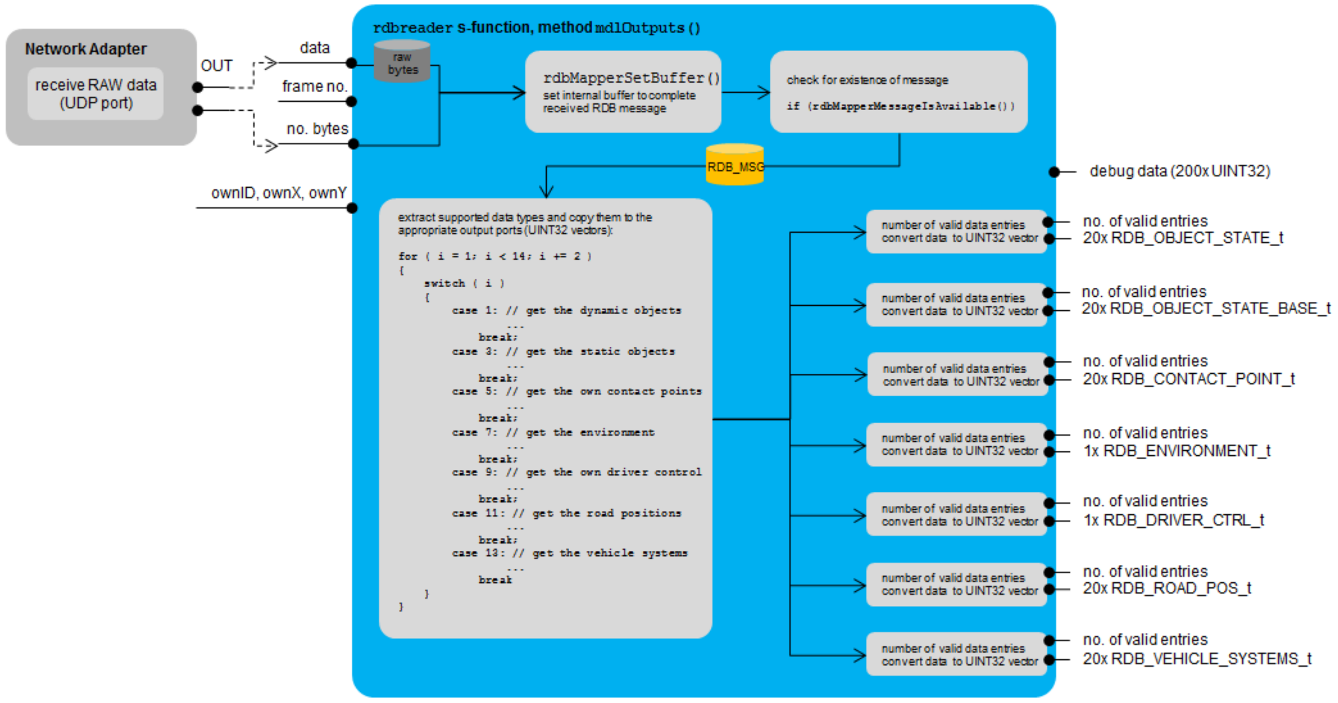
For connecting RDB to Matlab/Simulink, we provide template implementations of two s-functions, one for reading RDB data, one for writing the data. Both can be found under `Develop/Communication/Simulink`.

Note: these s-functions are not part of each distribution since some users of VTD already have their own, proprietary solutions. In no case do we want to interfere with these solutions.

Overview

The following figure gives an overview of the data flows:

Incoming Data


Network Adapter
receive RAW data (UDP port)
OUT
data
frame no.
no. bytes
ownID, ownX, ownY
rdbreader s-function, method mdlOutputs()
raw bytes
rdbMapperSetBuffer()
set internal buffer to complete received RDB message
check for existence of message
if (rdbMapperMessageIsAvailable())
RDB_MSG
debug data (200x UINT32)
number of valid data entries convert data to UINT32 vector
no. of valid entries
20x RDB_OBJECT_STATE_t
20x RDB_OBJECT_STATE_BASE_t
20x RDB_CONTACT_POINT_t
1x RDB_ENVIRONMENT_t
1x RDB_DRIVER_CTRL_t
20x RDB_ROAD_POS_t
20x RDB_VEHICLE_SYSTEMS_t

extract supported data types and copy them to the appropriate output ports (UINT32 vectors):

```
for ( i = 1; i < 14; i += 2 )
{
    switch ( i )
    {
        case 1: // get the dynamic objects
                ...
            break;
        case 3: // get the static objects
                ...
            break;
        case 5: // get the own contact points
                ...
            break;
        case 7: // get the environment
                ...
            break;
        case 9: // get the own driver control
                ...
            break;
        case 11: // get the road positions
                ...
            break;
        case 13: // get the vehicle systems
                ...
            break
    }
}
```

The s-function has to be connected to a network block from Simulink's or a supplier's library. Typically, UDP communication is used for the data exchange between VTD and Matlab/Simulink. The user has to provide the following inputs to the s-function:

- the data that has been read from network
- the Simulink simulation frame (internal variable)
- the number of bytes received from network (package size)
- numerical ID of own player, x- and y-position

It is recommended to set the network block to

- non-blocking
- variable message size
- data buffer: 30,000 bytes
- port: 48190
- sender address: any

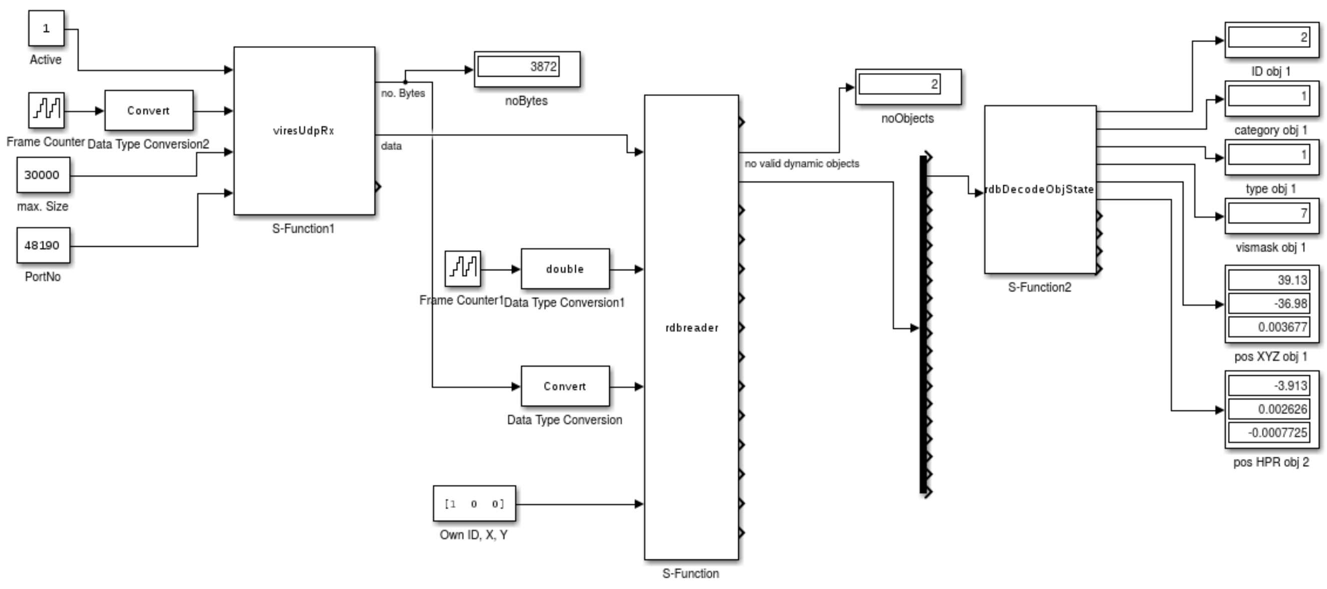
The TaskControl should be configured as follows (`Data/Setups/Current/Config/TaskControl/taskControl.xml`):

```
<TaskControl>
    :
    <RDB ... portType="RDB"
             composeSingleMessage="true" />
    :
</TaskControl>
```

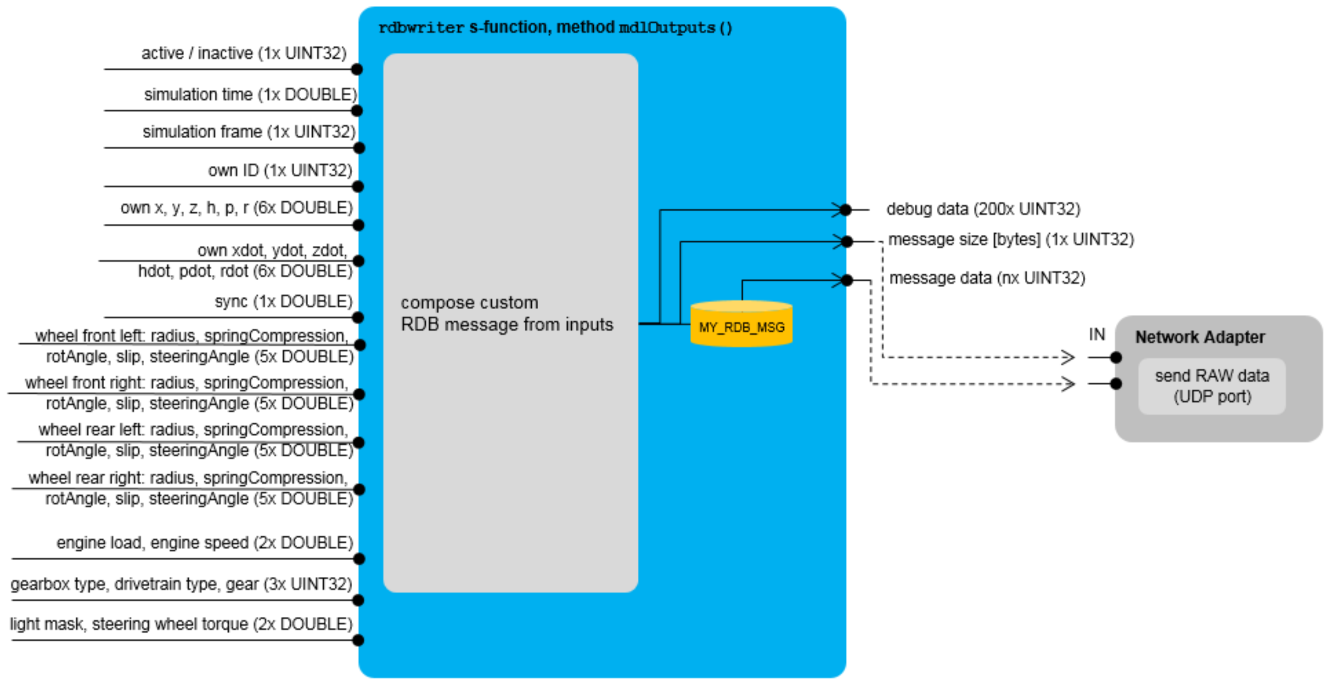
The s-function will return vectors of RDB data blocks. The valid size of each vector will be given together with the vectorized data itself. For decomposing the vectorized data, please use the RDB interface specification *viRDBIcd.h* or adapt the s-function as necessary.

The s-function should be called in each simulation frame but data should only be used if the respective number of valid entries is set to a positive value.

The following image shows a prototype implementation of the *rdbreader* s-function in combination with a UDP receiver block (here, VIRES' custom block is used but may be exchanged for the user's, of course).



Outgoing Data


rdbwriter s-function, method mdlOutputs()
active / inactive (1x UINT32)
simulation time (1x DOUBLE)
simulation frame (1x UINT32)
own ID (1x UINT32)
own x, y, z, h, p, r (6x DOUBLE)
own xdot, ydot, zdot, hdot, pdot, rdot (6x DOUBLE)
sync (1x DOUBLE)
wheel front left: radius, springCompression, rotAngle, slip, steeringAngle (5x DOUBLE)
wheel front right: radius, springCompression, rotAngle, slip, steeringAngle (5x DOUBLE)
wheel rear left: radius, springCompression, rotAngle, slip, steeringAngle (5x DOUBLE)
wheel rear right: radius, springCompression, rotAngle, slip, steeringAngle (5x DOUBLE)
engine load, engine speed (2x DOUBLE)
gearbox type, drivetrain type, gear (3x UINT32)
light mask, steering wheel torque (2x DOUBLE)
compose custom RDB message from inputs
MY_RDB_MSG
debug data (200x UINT32)
message size [bytes] (1x UINT32)
message data (nx UINT32)
IN
Network Adapter
send RAW data (UDP port)

The s-function for outgoing data assumes that the vehicle dynamics of the own vehicle is being computed within the Matlab/Simulink environment. The inputs have to be provided as labeled in the above image. The s-function will compose these inputs and create a network message from them. This message together with the appropriate size information has to be forwarded to a network block which is either instantiated from Simulink's or a supplier's library.

It is recommended to set the network block to

- immediate sending
- variable message size
- port: 48191
- receiver address: known host address or broadcast

Files

The s-functions come with the following files (complete list):

```
── Simulink
│   ├── CustomBlockRDB_Decoders
│   │   ├── rdbDecodeObjState.c
│   │   └── rdbDecodeRoadPos.c
│   ├── CustomBlockRDB_FromVTD
│   │   ├── rdbMapper.c
│   │   └── rdbreader.c
│   ├── CustomBlockRDB_SHMRx
│   │   └── viresShmRx.c
│   ├── CustomBlockRDB_ToVTD
│   │   └── rdbwriter.c
│   ├── CustomBlockRDB_UDPRx
│   │   └── viresUdpRx.c
│   ├── CustomBlockRDB_UDPTx
│   │   └── viresUdpTx.c
│   ├── inc
│   │   ├── slDummyDefs.h
│   │   ├── viRDBIcd.h
│   │   └── viRDBTypes.h
```

The main functions are *CustomBlockRDB_FromVTD* and *CustomBlockRDB_ToVTD*. They will include definitions from the directory *inc*. All other files/directories provide additional functionality in case you don't have the right Simulink blocks available (e.g. for reading/writing UDP) or you want to decode VTD data further (*CustomBlockRDB_Decoder*).

The code of the routines is contained in the following archive:

sampleClientSimulink.tgz

Compiling

Each block may be compiled and used independently. You will need Matlab/Simulink, of course. For the s-function **reading** VTD data, you will have to MEX and compile the file *rdbreader.c* only. All others will be referenced from within this file. For the s-function **writing** VTD data, you will have to MEX and compile only *rdbwriter.c*.

Due to the use of some C++ coding etc. we recommend that you use a command similar to the following:

> *mex -v GCC='/usr/bin/g++-4.8' CFLAGS='$FLAGS -fpermissive -fPIC' rdbreader.c*

and, respectively,

> *mex -v GCC='/usr/bin/g++-4.8' CFLAGS='$FLAGS -fpermissive -fPIC' rdbwriter.c*

Please set the compiler version in the above commands according to your installation.

Note: on some compilers you might have to use *COMPFLAGS* instead of *CFLAGS*. Please see the description of your compiler package.

Another Note: on MS Windows systems, a command like the following might also work

> *mex -LC:\\MATLAB\R2016b\sys\lcc64\lcc64\lib64 -lws2_32 viresUdpRx.c*

Debugging

RDBSniffer

VTD contains a tool which helps you trace an RDB data stream. The tool is called *RDBSnifffer*.

Location: `Runtime/Tools/RDBSniffer`

You may invoke this tool from the command line with the following options:

```
Usage:
rdbSniffer [-i interface] [-t frametime] [-h] [-s server] [-c portType] [-pkg id] [-play]
           [-p portNo] [-shm key] [-d] [-b] [-f filename] [-r filename] [-m messageCount] [-a] [-saveImages] [-v]
           [-realTime] [-csv filename] [-id number]
   -h :                        show this help information
   -i interface:               use the indicated interface for communication (e.g. "eth1")
   -t frametime:               run with the given frametime instead of the std. 0.001s (1000Hz)
                               in replay mode this is the pause between two frames
   -s server:                  server name / address
   -c [udp | tcp | loopback]:  connection type
   -p port:                    port number (for ethernet device communication)
   -shm key:                   read from SHM with the given key
   -d :                        show details
   -b :                        show binary dump
   -f filename:                analyze the indicated file
   -r filename:                set the record file
   -m messageCount:            length of recording [messages], forever per default
   -pkg id:                    numeric ID of the package that is to be shown in the output (i.e. pkg filter)
   -play:                      replay the data contained in a given file
   -a:                         analyze custom data contents
   -saveImages:                save images in dedicated files
   -v:                         enable verbose mode
   -realTime:                  show debug output of real-time vs. simulation time of packages
   -csv filename:              convert output to CSV and save in file <filename>
   -id number:                 show only packages matching a certain player or object ID (not applicable to all packages)
```

Examples:

1) display all package headers from standard taskControl RDB port which is configured for TCP

`./rdbSniffer -c tcp`

2) display all package headers and contents from standard taskControl RDB port which is configured for TCP

```
./rdbSniffer -c tcp -d
```

3) display only OBJECT_STATE headers and contents from standard taskControl RDB port which is configured for TCP

```
./rdbSniffer -c tcp -d -pkg 9
```

4) display only OBJECT_STATE headers and contents for objects no. 1 and no. 2 from standard taskControl RDB port which is configured for TCP

```
./rdbSniffer -c tcp -d -pkg 9 -id 1 -id 2
```

Note: for a different port number, use option `-p portNo`, for a UDP connection use option `-c udp`

Data Stream Example

The RDBSniffer (see above) may be used to create a data stream for debugging purposes, e.g. if you wish to check whether you are correctly reading/writing RDB data. The attached package

rdbSniffer.tgz

contains the RDBSniffer and a data recording that may be replayed. After unpacking, open a terminal, cd into the directory `RDBSniffer` and execute the command for your intended connection type; the recorded data will be played back to the respective port:

- UDP connection (port 48270):
  - ./rdbSniffer -c udp -p 48270 -play -f rdbRecord.dat
- TCP server (port 48190):
  - ./rdbSniffer -c tcpServer -play -f rdbRecord.dat
- TCP client (port 48190):
  - ./rdbSniffer -c tcp -play -f rdbRecord.dat

For verifying that the replay works, you may use the sniffer (in a separate terminal) and read the played-back data:

- UDP connection (port 48270):
  - ./rdbSniffer -c udp -p 48270
- TCP client (port 48190) for TCP server player:
  - ./rdbSniffer -c tcp
- TCP server (port 48190) for TCP client player:
  - ./rdbSniffer -c tcpServer

RDB Data Recording

RDB streams may be recorded for debugging or logging. Currently, there are three means to record RDB data:

A) Recording of the RDB stream into the TaskControl

The RDBin stream of the TaskControl (i.e. all command sent by exterior components to the TC) is available for recording. You may activate this feature using the following command sequence:

```
<Record stream="RDBin"><File path="/tmp" name="rdbinrec" addDate="true"/><Start/></Record>
```

The TaskControl will create a file according to the following pattern:

```
/tmp/rdbinrec20140529_105535.dat
```

B) Recording of an arbitrary RDB stream using the *RDBSniffer*

You may use the *RDBSniffer* to record an RDB stream by attaching it to the corresponding interface (see above) and specifying an output file. Note: you can only attach the *RDBSniffer* to the producer of data, not to the consumer.

Example (record RDBout of TC):

```
./rdbSniffer -c tcp -p 48190 -r myFile.dat
```

This will create the record file `myFile.dat`

C) RDBRecorder Plugin

This plugin runs in the ModuleManager. Its advantage is that you can configure it to start with each simulation automatically. It will record all data sent via the RDBout port of the TaskControl.

Configuration

Example configuration (excerpt of MM config file):

```
<Sensor name="rdbRec" type="video">
    <Load     lib="libModuleRDBRecorder.so" path="" persistent="true" />
    <Cull     enable="false" />
    <Player   default="true" />
    <File     name="myRDBRecording" path="/tmp" addDate="false" addCounter="true" autoRecord="true" />
    <Debug    enable="false"/>
</Sensor>
```

The main configuration parameters of the RDB recorder are given in the tag `<File>`:

- `name: root file name of the recording`
- `path: storage path of the recording`

- `addDate: if "true" the start date and time of the recording will be added to the filename automatically`
- `addCounter: if "true" a counter will be added to the filename automatically`
- `autoRecord: if "true" (or missing), the recording will start automatically at the beginning of a simulation; otherwise it has to be started explicitly (see below)`

In addition, the user may also specify the types of data packages that shall be recorded. In this case, one entry of type `<Filter>` has to be provided for each package type and the **numeric** ID of the corresponding RDB package has to be provided (see also RDB interface documentation). If no filter is defined, all packages will be recorded.

Example:

```
<Sensor name="rdbRec" type="video">
    <Load     lib="libModuleRDBRecorder.so" path="" persistent="true" />
    <Cull     enable="false" />
    <Player   default="true" />
    <File     name="myRDBRecording" path="/tmp" addDate="false" addCounter="true" autoRecord="true" />
    <Filter   pkgId="9" />  <!-- RDB_PKG_ID_OBJECT_STATE -->
    <Filter   pkgId="26" /> <!-- RDB_PKG_ID_DRIVER_CTRL -->
    <Debug    enable="false" />
</Sensor>
```

The configuration parameters may also be sent via SCP in the init phase of the simulation.

Control at run-time

The recorder plug-in may be influenced at run-time. Recordings may be started and stopped, filenames may be modified etc.

NOTE: if you want to modify the settings of the RDB recorder, make sure that you are always addressing it with its correct name (*rdbRec* in the example above). Otherwise the commands will not be executed!

Start the recording:
`<Sensor name="rdbRec"><File><Start/></File></Sensor>`

Stop the recording:
`<Sensor name="rdbRec"><File><Stop/></File></Sensor>`

Data Analysis

A recorded file may be analyzed using, again, the tool *RDBSniffer*. The command sequence is as follows:

```
./rdbSniffer -f /tmp/rdbinrec20140529_105535.dat
```

For further information, just invoke

```
./rdbSniffer -h
```

Copy RDB messages - rdbCopy

package: vtd.2.0.3.addOns.RDBCopy.20180123.tgz
package: vtd.2.1.0.addOns.RDBCopy.20180123.tgz

RDB messages can be copied from network/shared memory to network/shared memory with the tool rdbCopy:
The tool can be found in `VTD.2.0/Runtime/Tools/RDBCopy`.

Copy from local shared memory segment 0x0816a to the local server port 50555:

```
./rdbCopy 0x0816a localhost:50555
```

Copy from host 192.168.101.133 with port 50555 to the local shared memory segment 0x0716a:

```
./rdbCopy 192.168.101.133:50555 0x0716a
```

For further information, just invoke

```
./rdbCopy -h
```

Simulation Control Protocol

Control of the simulation and event-driven feedback is performed via the XML-based SCP protocol (see figure at the top of this page). It is accessible via a dedicated network (Ethernet) connection.

Contents

SCP provides **the** interface to all non-periodic communication within VTD. Among these are:

- Common Information About the Simulation
  - filename and path of the current scenario and OpenDRIVE file
  - simulation state (start, stop, ...)
- Information About Actions Within the Simulation
  - execution of triggers from within the scenario
  - state changes of traffic lights
- many more...

The full extent of the SCP command interface can be retrieved from the documentation in `VTD/Doc/SCP_HTML/index.html`.

Format

Each message is formatted as follows:

```
SCP_MSG_HDR_t
command text
```

This means that the header of an SCP message is followed immediately by the actual command' text. The length of the command text has to be given as information within the header block.

The header is structured as follows:

```
typedef struct
{
    uint16_t  magicNo;                    /**< must be SCP_MAGIC_NO                        @unit @link GENERAL_DEFINITIONS @e
    uint16_t  version;                    /**< upper byte = major, lower byte = minor       @unit _
    char      sender[SCP_NAME_LENGTH];    /**< name of the sender as text                  @unit _
    char      receiver[SCP_NAME_LENGTH];  /**< name of the receiver as text                @unit _
    uint32_t  dataSize;                   /**< number of data bytes following the header    @unit byte
} SCP_MSG_HDR_t;
```

The member *dataSize* represents the length of the command. If your command string includes a terminating `\0` character, then the *dataSize* must also count this character. It is allowed not to include the terminating `\0` character in an SCP message.

Send Structure

Mirroring of Commands

SCP commands are - typically - sent to the TaskControl using port 48179. The TaskControl will interpret these commmands, execute its own actions (if applicable) and then forward the commands to everyone connected to the SCP port. Therefore, the original sender will also receive again the command sent. Some commands, however, are excepted from this broadcast mechanism.

Confirmation of Commands

SCP is based on communication via TCP/IP connections. Therefore, it is assumed that commands sent will arrive at the receiver. Also most commands will be mirrored (see above). There is a sub-set of commands that will create explicit replies upon successful execution. These are:

- `<SimCtrl><Init/>...</SimCtrl>`
  - answer: `<SimCtrl><InitDone/></SimCtrl>`
- `<SimCtrl><Start/>...</SimCtrl>`
  - answer: `<SimCtrl><Run/></SimCtrl>`
- `<Query>...</Query>`
  - answer: `<Reply>...</Reply>`

Special case: `<Query>` Commands

In case of `<Query>` commands, the user may ask for a specific tagging of the corresponding `<Reply>` command with a unique *label* that has been provided by the user.

Example:

```
<Query label="a58s7" entity="player" id="1"/>"
<Reply label="a58s7" entity="player" id="1" name="Ego"/>
```

All `<Query>` commands handled by The taskControl are covered by this feature, i.e. all but

- `<Query entity="traffic..../>`
- `<Query entity="imagegenerator"><VisualDatabase/></Query>`
- `<Query entity="imagegenerator"><Camera/></Query>`

General Receipt

If you want to make sure that the connection to the TaskControl is working and that a command has arrived, you may issue an independent `<Query>` for a receipt **after** having sent your actual command. Note: this is not an explicit confirmation of arrival / processing of the previous but quite good as a means to adjust timing of your components according to the arrival of a command at the TaskControl. You may use this functionality as follows:

1. send your original command
2. issue `<Query entity="taskControl"><Receipt id="<your unique string>"/></Query>`
3. you will receive: `<Reply entity="taskControl"><Receipt id="<your unique string>"/></Reply>`

Time-stamping of Commands

availability: VTD 2.0.2+

Per default and in its header, SCP does not provide any information about the simulation time when a command has been issued. You may change this by setting the following parameter in the TC configuration:

```
<TaskControl><SCP addSimTime="true"/></TaskControl>
```

Afterwards, each command sent by the TaskControl will be preceded by a timing information.

Example:

```
<SCP simTime="1.920000"/>
  <Symbol name="trLight02">
```

```
    <Orientation type="inertial" dhDeg="45"/>
    <Overlay id="2" state="2"/>
    <PosInertial x="3.3" y="11.2" z="2.8"/>
    <RectSize width="0.2" height="0.2"/>
  </Symbol>
```

Control by External Components

Almost all actions can be controlled / triggered by an external component if it's connected to VTD via SCP. In addition, settings (e.g. driver parameters) may be adjusted online.

Implementation, sample client

The following archive contains sample clients for SCP and RDB communications. These examples are provided "AS IS" and are not optimized for a specific platform.

sampleClientsRDBandSCP.tgz

Scenarios and Databases

Scenario

The scenario is stored in an XML format which is proprietary to VIRES. It contains references to the visual database and the logical database. Actions of all players are also contained in the scenario file. The file may be edited using the graphical ScenarioEditor.

Remote ScenarioEditor

Description:

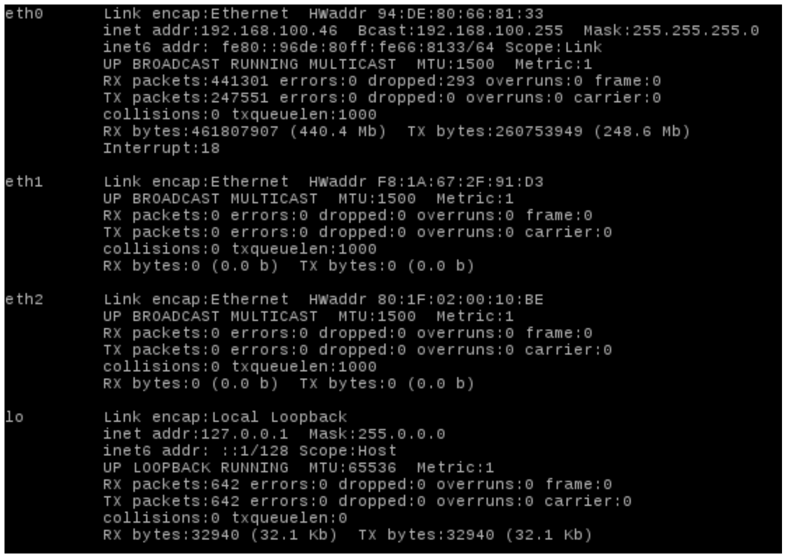
1.) Install/Copy ScenarioEditor to a remote computer

a) Copy current ScenarioEditor (not the link) from directory #ProjectPath#/VTD/Runtime/Core/ScenarioEditor/ to the remote computer directory (e.g. #ProjectPath#/ScenarioEditorRemote/)
b) Copy ScenarioEditor configuration scenarioEditor.xml.ini (not the link) from the directory #ProjectPath#/VTD/Data/Setups/Standard/Config/ScenarioEditor/ into the created directory (point a) of the remote computer
c) Copy scenario from directory #ProjectPath#/VTD/Data/Projects/Current/Scenarios/ (e.g. HighwayPulk.xml) into the created directory (point a) of the remote computer

2.) Configure VTD for UDP transmission

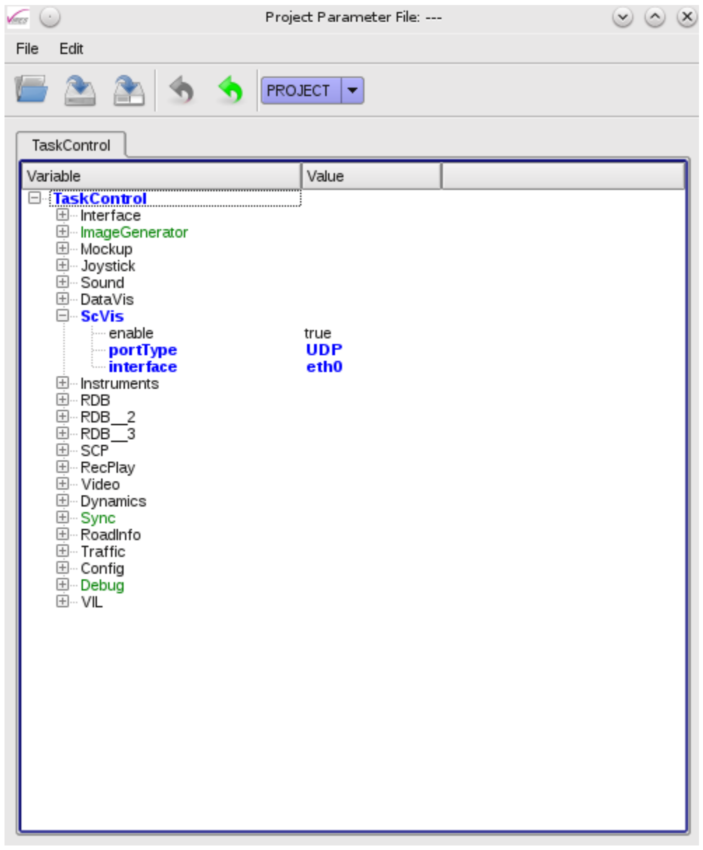
a) Open a console on the computer where VTD is running and type
b) ifconfig

Example:



```
eth0      Link encap:Ethernet  HWaddr 94:DE:80:66:81:33
          inet addr:192.168.100.46  Bcast:192.168.100.255  Mask:255.255.255.0
          inet6 addr: fe80::96de:80ff:fe66:8133/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:441301 errors:0 dropped:293 overruns:0 frame:0
          TX packets:247551 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:461807907 (440.4 Mb)  TX bytes:260753949 (248.6 Mb)
          Interrupt:18

eth1      Link encap:Ethernet  HWaddr F8:1A:67:2F:91:D3
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

eth2      Link encap:Ethernet  HWaddr 80:1F:02:00:10:BE
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:642 errors:0 dropped:0 overruns:0 frame:0
          TX packets:642 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:32940 (32.1 Kb)  TX bytes:32940 (32.1 Kb)
```

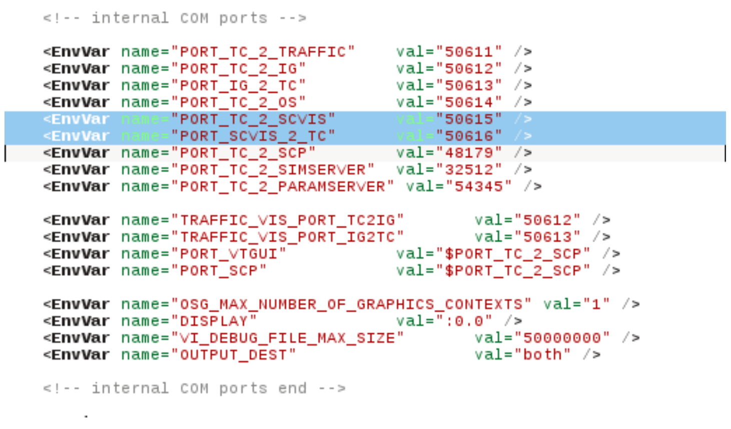
c) Start VTD.
d) Menu: View->Show Parameterbrowser
e) Edit ScVis.
f) Set Variable: portType to Value: UDP
g) Set Variable: interface to Value: eth0 (use interface name from command "ifconfig" of point b, because this is used/connected)



```
Variable                  Value
⊟ TaskControl
   ⊞ Interface
   ⊞ ImageGenerator
   ⊞ Mockup
   ⊞ Joystick
   ⊞ Sound
   ⊞ DataVis
   ⊟ ScVis
        enable            true
        portType          UDP
        interface         eth0
   ⊞ Instruments
   ⊞ RDB
   ⊞ RDB__2
   ⊞ RDB__3
   ⊞ SCP
   ⊞ RecPlay
   ⊞ Video
   ⊞ Dynamics
   ⊞ Sync
   ⊞ RoadInfo
   ⊞ Traffic
   ⊞ Config
   ⊞ Debug
   ⊞ VIL
```

3.) Find UDP ports from VTD

a) Open console.
b) cd #ProjectPath#/VTD/Data/Setups/Standard/Config/SimServer/
c) Open simServer.xml with a editor (e.g. mit kwrite simServer.xml)

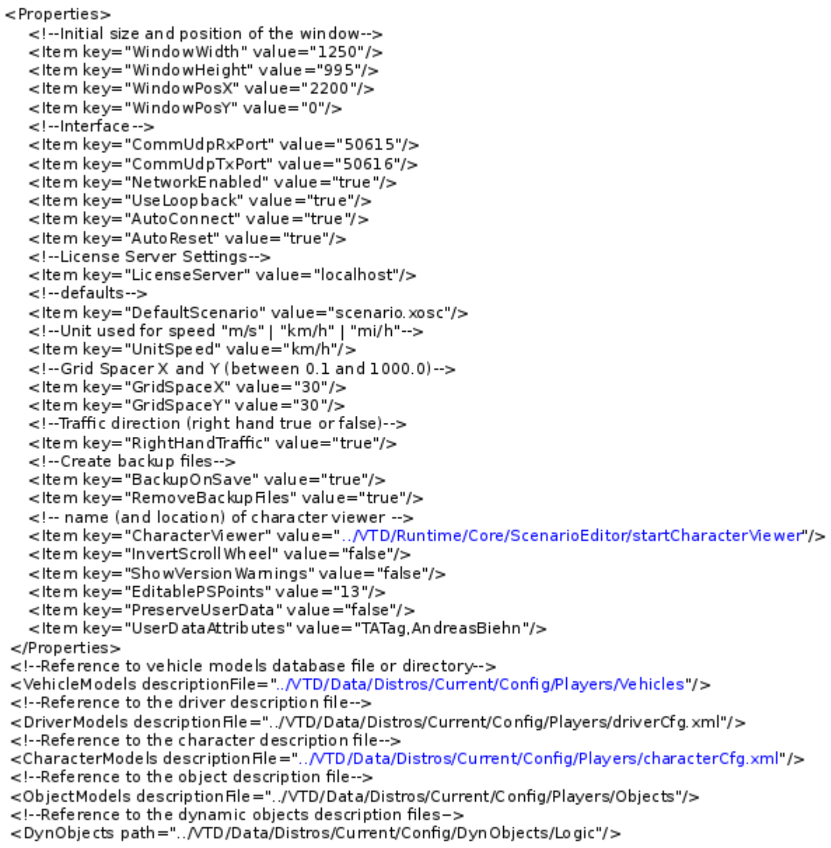


```
<!-- internal COM ports -->

<EnvVar name="PORT_TC_2_TRAFFIC"     val="50611" />
<EnvVar name="PORT_TC_2_IG"          val="50612" />
<EnvVar name="PORT_IG_2_TC"          val="50613" />
<EnvVar name="PORT_TC_2_OS"          val="50614" />
<EnvVar name="PORT_TC_2_SCVIS"       val="50615" />
<EnvVar name="PORT_SCVIS_2_TC"       val="50616" />
<EnvVar name="PORT_TC_2_SCP"         val="48179" />
<EnvVar name="PORT_TC_2_SIMSERVER"   val="32512" />
<EnvVar name="PORT_TC_2_PARAMSERVER" val="54345" />

<EnvVar name="TRAFFIC_VIS_PORT_TC2IG"        val="50612" />
<EnvVar name="TRAFFIC_VIS_PORT_IG2TC"        val="50613" />
<EnvVar name="PORT_VTGUI"            val="$PORT_TC_2_SCP" />
<EnvVar name="PORT_SCP"              val="$PORT_TC_2_SCP" />

<EnvVar name="OSG_MAX_NUMBER_OF_GRAPHICS_CONTEXTS" val="1" />
<EnvVar name="DISPLAY"               val=":0.0" />
<EnvVar name="VI_DEBUG_FILE_MAX_SIZE"        val="50000000" />
<EnvVar name="OUTPUT_DEST"                   val="both" />

<!-- internal COM ports end -->
```

d) Notice the two values of PORT_TC_2_SCVIS and PORT_SCVIS_2_TC

4) Configure SenarioEditor

4.1) Configure scenarioEditor.xm.ini
a) Open console.
b) cd #ProjectPath#/ScenarioEditorRemote
c) Open scenarioEditor.xml.ini with kwrite



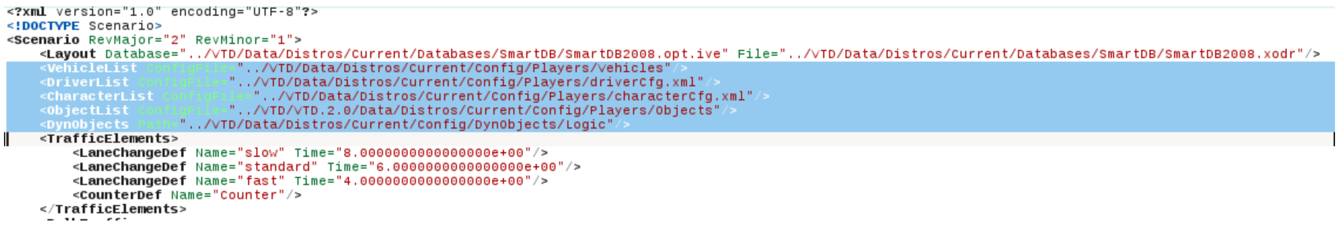
```
<Properties>
    <!--Initial size and position of the window-->
    <Item key="WindowWidth" value="1250"/>
    <Item key="WindowHeight" value="995"/>
    <Item key="WindowPosX" value="2200"/>
    <Item key="WindowPosY" value="0"/>
    <!--Interface-->
    <Item key="CommUdpRxPort" value="50615"/>
    <Item key="CommUdpTxPort" value="50616"/>
    <Item key="NetworkEnabled" value="true"/>
    <Item key="UseLoopback" value="true"/>
    <Item key="AutoConnect" value="true"/>
    <Item key="AutoReset" value="true"/>
    <!--License Server Settings-->
    <Item key="LicenseServer" value="localhost"/>
    <!--defaults-->
    <Item key="DefaultScenario" value="scenario.xosc"/>
    <!--Unit used for speed "m/s" | "km/h" | "mi/h"-->
    <Item key="UnitSpeed" value="km/h"/>
    <!--Grid Spacer X and Y (between 0.1 and 1000.0)-->
    <Item key="GridSpaceX" value="30"/>
    <Item key="GridSpaceY" value="30"/>
    <!--Traffic direction (right hand true or false)-->
    <Item key="RightHandTraffic" value="true"/>
    <!--Create backup files-->
    <Item key="BackupOnSave" value="true"/>
    <Item key="RemoveBackupFiles" value="true"/>
    <!-- name (and location) of character viewer -->
    <Item key="CharacterViewer" value="../VTD/Runtime/Core/ScenarioEditor/startCharacterViewer"/>
    <Item key="InvertScrollWheel" value="false"/>
    <Item key="ShowVersionWarnings" value="false"/>
    <Item key="EditablePSPoints" value="13"/>
    <Item key="PreserveUserData" value="false"/>
    <Item key="UserDataAttributes" value="TATag,AndreasBiehn"/>
</Properties>
<!--Reference to vehicle models database file or directory-->
<VehicleModels descriptionFile="../VTD/Data/Distros/Current/Config/Players/Vehicles"/>
<!--Reference to the driver description file-->
<DriverModels descriptionFile="../VTD/Data/Distros/Current/Config/Players/driverCfg.xml"/>
<!--Reference to the character description file-->
<CharacterModels descriptionFile="../VTD/Data/Distros/Current/Config/Players/characterCfg.xml"/>
<!--Reference to the object description file-->
<ObjectModels descriptionFile="../VTD/Data/Distros/Current/Config/Players/Objects"/>
<!--Reference to the dynamic objects description files-->
<DynObjects path="../VTD/Data/Distros/Current/Config/DynObjects/Logic"/>
```

d) Customize the blue maked paths.

> *The paths must be available on the remote machine (physical or mounted)*
> *Customize the blue marked paths from the path #ProjectPath#/ScenarioEditorRemote because there is the executealbe ScenarioEditor*

4.2) Configure Scenario (e.g. HighwayPulk.xml)

a) Open console on remote computer
b) cd #ProjectPath#/ScenarioEditorRemote
c) Open HighwayPulk.xml with a editor



```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE Scenario>
<Scenario RevMajor="2" RevMinor="1">
    <Layout Database="../VTD/Data/Distros/Current/Databases/SmartDB/SmartDB2008.opt.ive" File="../VTD/Data/Distros/Current/Databases/SmartDB/SmartDB2008.xodr"/>
    <VehicleList ConfigFile="../VTD/Data/Distros/Current/Config/Players/vehicles"/>
    <DriverList ConfigFile="../VTD/Data/Distros/Current/Config/Players/driverCfg.xml"/>
    <CharacterList ConfigFile="../VTD/Data/Distros/Current/Config/Players/characterCfg.xml"/>
    <ObjectList ConfigFile="../VTD/VTD.2.0/Data/Distros/Current/Config/Players/Objects"/>
    <DynObjects Path="../VTD/Data/Distros/Current/Config/DynObjects/Logic"/>
    <TrafficElements>
        <LaneChangeDef Name="slow" Time="8.0000000000000000e+00"/>
        <LaneChangeDef Name="standard" Time="6.0000000000000000e+00"/>
        <LaneChangeDef Name="fast" Time="4.0000000000000000e+00"/>
        <CounterDef Name="Counter"/>
    </TrafficElements>
```

d) Customize the marked paths

> *The paths must be available on the remote machine (physical or mounted)*
> *Customize the blue marked paths from the path #ProjectPath#/ScenarioEditorRemote because there is the executealbe ScenarioEditor*

4.3) Open ScenarioEditor and customize other settings

a) Open console on the remote computer
b) cd #ProjectPath#/ScenarioEditorRemote
c) Open ScenarioEditor on console ./scenarioEditor -f scenarioEditor.xml.ini HighwayPulk.xml
d) Press Properties button
e) Customize paths of Scene

> *Layout File: ../VTDData/Distros/Current/Databases/SmartDB/SmartDB2008.xodr (Beispiel)*
> *Visual Database: ../VTD/Data/Distros/Current/Databases/SmartDB/SmartDB2008.opt.ive (Beispiel)*

f) Press OK
g) Open Menu Edit->Editor Settings
h) Customize Interface

> *UDP Receiver Port: 50615 (use value PORT_TC_2_SCVIS from simServer.xml)*
> *UDP Transmit Port: 50616 (use value PORT_SCVIS_2_TC from simServer.xml)*

> *Network Enabled:true*
> *Use Loopback: true*
> *Auto Connect: true*
> *Auto Reset: true*

i) Press OK
j) Press Save button
k) Close ScenarioEditor and reopen it with command like in point c

5) Possible ScenarioEditor configurations

a) It is possible to run different ScenarioEditor on different remote computers wich are connected to the same Simluation (VTD).
b) It is possible to run multiple ScenarioEditor on a remote machine. Start every ScenarioEditor from a different directory (e.g. #ProjectPath#/ScenarioEditorRemote2) with corresponding configuration files.
VTD and ScenarioEditor are connected with Ports. You have to customize the different VTD's. So configure in the corresponding simServer.xml the values PORT_TC_2_SCVIS and PORT_SCVIS_2_TC that the ports are unique. Customize in the different running ScenarioEditor in Settings->Interface the ports corresponding to the configuration of the VTD.

Database

The database contains the graphical elements of the virtual world. It is available in binary .ive-format (OpenSceneGraph). The corresponding logical database of the road network is stored in the OpenDRIVE format, an XML-based description. For the design and modification of both database, the editor RoadDesigner (ROD) may be purchased optionally.

ROS (Robot Operating System)

Please find here an example for linking ROS with VTD:

vtd_ros_gateway.tgz

These steps were performed on an Ubuntu 16.04.1 LTS computer:

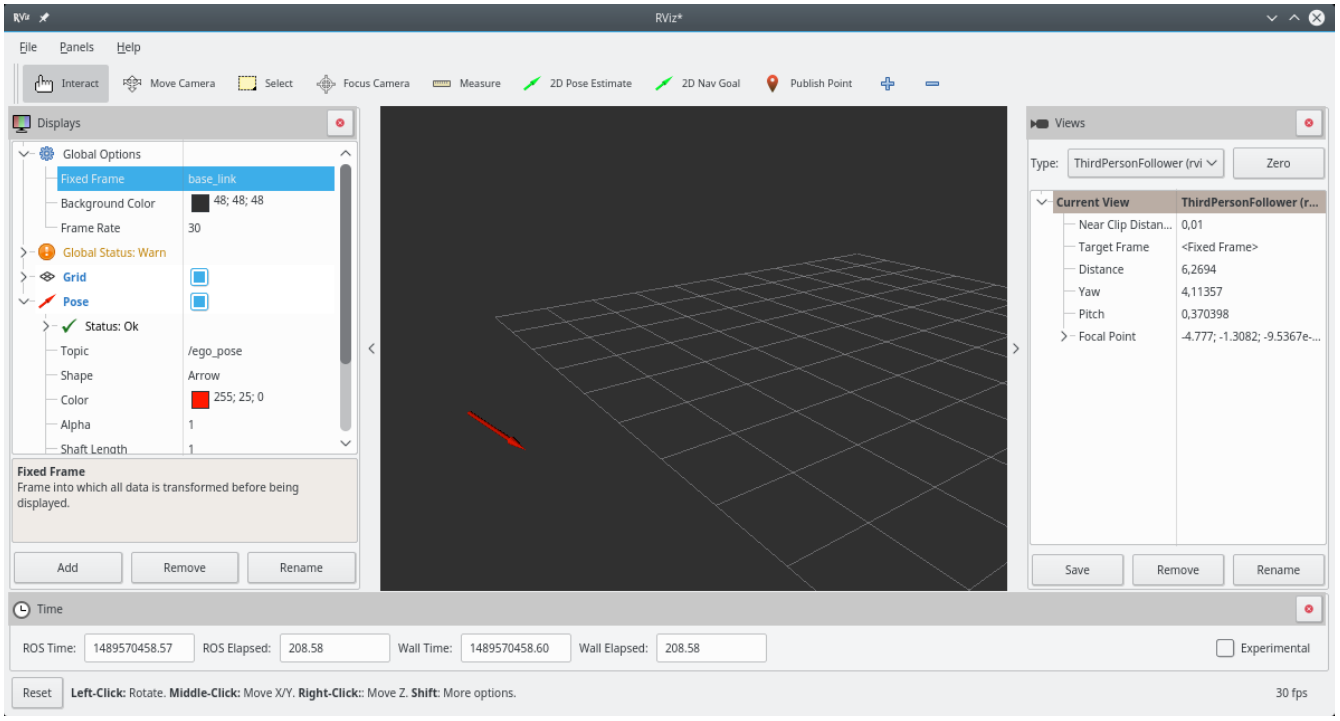
- installed ROS Kinetic environment according to ROS Kinetic installation instructions
- created catkin workspace as described in the beginners tutorials
- followed the ROS tutorials up to "Writing a Simple Publisher and Subscriber (C++)"
- created the vtd_gateway prototype based on the "talker" example from the ROS tutorials and based on the "RDBClientSample" delivered with VTD

The attached "vtd_ros_gateway.tgz" was packed from the catkin workspace source folder (e.g. ~/catkin_ws/src).

The VTD gateway does the following:

- it reads the RDB_OBJECT_STATE messages from the RDB stream of a VTD instance running on the same machine
- based on the RDB_OBJECT_STATE, it computes a message of the type "geometry_msgs::PoseStamped" for the vehicle with id==1
- this message is published in the ROS environment

For testing "rviz" was installed on the computer. It allows graphically displaying PoseStamped messages:



Global Options	
Fixed Frame	base_link
Background Color	48; 48; 48
Frame Rate	30
Global Status: Warn	
Grid	✓
Pose	✓
Status: Ok	
Topic	/ego_pose
Shape	Arrow
Color	🟥 255; 25; 0
Alpha	1
Shaft Length	1

Fixed Frame
Frame into which all data is transformed before being displayed.

Type: ThirdPersonFollower (rvi

Current View	ThirdPersonFollower (r...
Near Clip Distan...	0,01
Target Frame	<Fixed Frame>
Distance	6,2694
Yaw	4,11357
Pitch	0,370398
Focal Point	-4.777; -1.3082; -9.5367e-...

ROS Time: 1489570458.57 ROS Elapsed: 208.58 Wall Time: 1489570458.60 Wall Elapsed: 208.58

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click::** Move Z. **Shift:** More options. 30 fps

Open Simulation Interface

The current version of the OSI Plug-In.

vtd.2.1.0.addOns.OSI.20180903.tgz

Note: The Plug-In writes OSI3 data of type SensorData (not SensorView)!

Installation

The package contains following files:

- VTD.2.1/Data/Distros/Distro/Plugins/ModuleManager/libModuleOsi3Fmu.so -> libModuleOsi3Fmu.so.0.10.2
- VTD.2.1/Data/Distros/Distro/Plugins/ModuleManager/libModuleOsi3Fmu.so.0 -> libModuleOsi3Fmu.so.0.10.2
- VTD.2.1/Data/Distros/Distro/Plugins/ModuleManager/libModuleOsi3Fmu.so.0.10 -> libModuleOsi3Fmu.so.0.10.2
- VTD.2.1/Data/Distros/Distro/Plugins/ModuleManager/libModuleOsi3Fmu.so.0.10.2
- VTD.2.1/Data/Setups/Current/Bin/libopen_simulation_interface.so
- VTD.2.1/Data/Distros/Distro/Config/ModuleManager/moduleManager.xml

The moduleManager.xml can be adapted at Project, Setup or Distro level. A modified moduleManager.xml is included in the package and will overwrite the existing one in the Distro! If you want to modify an existing moduleManager.xml, following changes are necessary:

Plug-In with FMU (see following chapters):

```
<Sensor name="OsiFmu">
    <Load     lib="libModuleOsi3Fmu.so" path="" persistent="true" />
    <Port     name="RDBout" number="48567" type="TCP" />
    <File     fmuFilePath="PATH/TO/FMU/FMU.so"/>
    <Frustum  near="5.0" far="500.0" left="10.0" right="10.0" bottom="3.0" top="3.0" />
    <Cull     maxObjects="100" enable="true" />
    <Player   default="true" />
    <Origin   dx="0.0" dy="0.0" dz="0.0" dhDeg="0.0" dpDeg="0.0" drDeg="0.0" type="sensor"/>
    <Position dx="3.5" dy="0.0" dz="0.5" dhDeg="0.0" dpDeg="0.0" drDeg="0.0"/>
    <Database resolveRepeatedObjects="true" continuousObjectTesselation="2.0" />
    <Filter   objectType="pedestrian"/>
    <Filter   objectType="vehicle"/>
    <Filter   objectType="trafficSign"/>
    <Filter   objectType="roadInfo"/>
    <Filter   objectType="laneInfo"/>
    <Filter   objectType="laneInfoComplete"/>
    <Filter   objectType="roadMarks"/>
</Sensor>
```

Plug-In without FMU (see following chapters):

```
<Sensor name="OsiFmu">
    <Load     lib="libModuleOsi3Fmu.so" path="" persistent="true" />
    <Port     name="RDBout" number="48567" type="TCP" />
    <Frustum  near="5.0" far="500.0" left="10.0" right="10.0" bottom="3.0" top="3.0" />
    <Cull     maxObjects="100" enable="true" />
    <Player   default="true" />
    <Origin   dx="0.0" dy="0.0" dz="0.0" dhDeg="0.0" dpDeg="0.0" drDeg="0.0" type="sensor"/>
    <Position dx="3.5" dy="0.0" dz="0.5" dhDeg="0.0" dpDeg="0.0" drDeg="0.0"/>
    <Database resolveRepeatedObjects="true" continuousObjectTesselation="2.0" />
    <Filter   objectType="pedestrian"/>
    <Filter   objectType="vehicle"/>
    <Filter   objectType="trafficSign"/>
    <Filter   objectType="roadInfo"/>
    <Filter   objectType="laneInfo"/>
    <Filter   objectType="laneInfoComplete"/>
    <Filter   objectType="roadMarks"/>
</Sensor>
```

For installation, untar the package at the level of your VTD 2.1 installation (folder VTD.2.1 is included in the package).

> *tar xvzf vtd.2.1.0.addOns.OSI.[DATE].tgz*

Point Cloud Configuration

Optionally the plugin supports reading of LiDAR point cloud data from shared memory. To configure the plugin to do so, add the following to the sensor configuration. Fill in shmKey and bufferFlag with values from your setup:

```
<Sensor name="OsiFmu">

    ...

    <PointCloud shmKey="0x0816a" bufferFlag="0x2"/>
</Sensor>
```

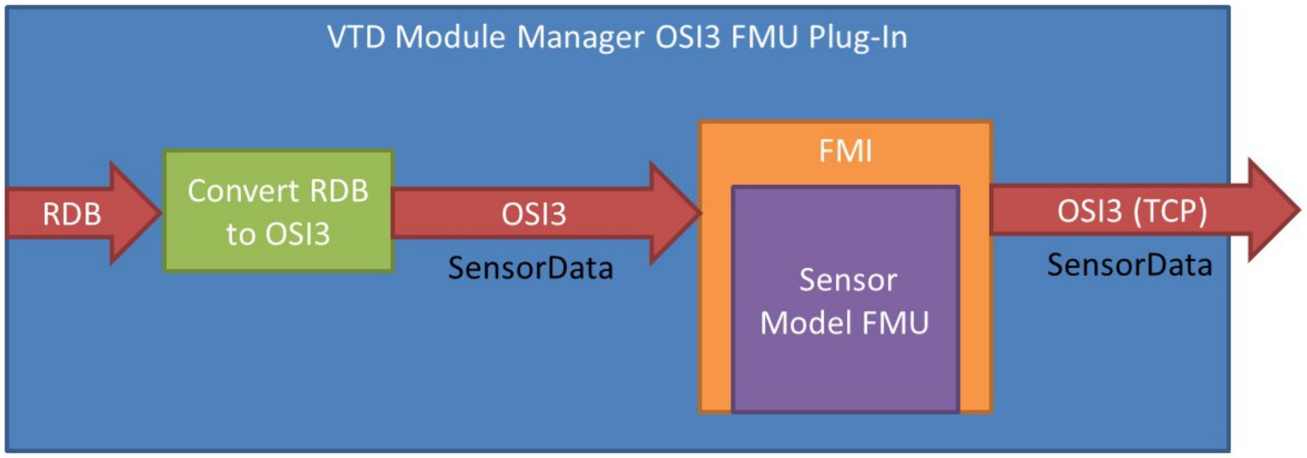
Data must be written into shared memory in the format RGBA32F where the RGB channels correspond to spherical coordinates of the hit point, the most significant 2 bytes of A represent the signal intensity as a uint16 and the least significant 2 bytes of A represent the detection classification as a uint16. This is in keeping with the current version of the OptiXLidar plugin. Multiple RGBA32A buffers are treated as subsequent reflections. For example the first, second and third buffers correspond to the results of the first, second and third reflection respectively.

Example camera Cuda kernel to write data into buffer:

```
__device__ void writePayloadToBuffer(const float3& position, float intensity, int objectType, optix::buffer<float4, 2>& buffer
{
    uint signalIntensityUInt = (uint) (intensity * 65535.0f);
    uint codedInfo = (signalIntensityUInt << 16) | (objectType & 0x00ff);

    buffer[launch_index].x = position.x;
    buffer[launch_index].y = position.y;
    buffer[launch_index].z = position.z;
    buffer[launch_index].w = __uint_as_float(codedInfo);
}
```

Using the Plug-In with a OSI3 Sensor Model FMU

A Sensor Model FMU can directly be loaded into the Plug-In using the FMI standard. OSI Sensor Model FMUs are usually build according to the OSI Sensor Model Packaging (OSMP) rules (https://github.com/OpenSimulationInterface/osi-sensor-model-packaging). In difference to OSMP, the Plug-In requires a Sensor Model FMU that reads and writes OSI3 data of the type osi3::SensorData. The reason for this deviation from OSMP is the usage of osi3::FeatureData in particular osi3::LidarDetectionData which is not contained in osi3::SensorView. A modified OSMP DummySensor can be found in https://redmine.vires.com/projects/vtd/wiki/Daten_aus_VTD#Differences-to-OSMP.

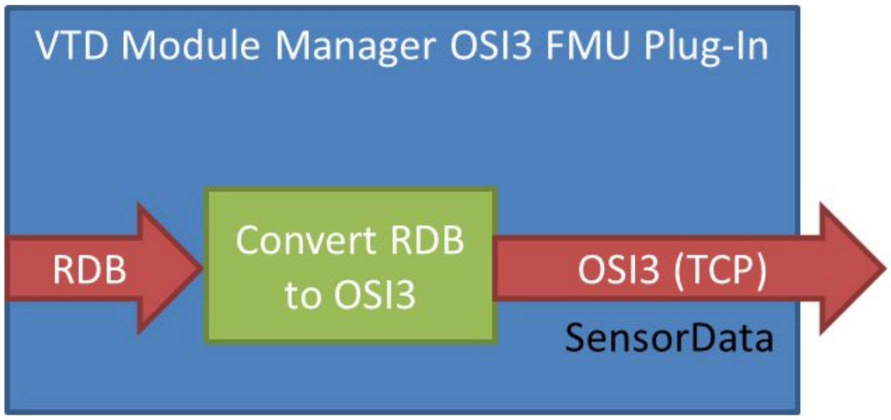


The Plug-In will load, initialize and cyclically call the doStep function of the FMU, providing input and reading output using the pointer method described in OSMP. As stated above input and output are of the same OSI3 message type osi3::SensorData. A list of valid input provided by VTD can be found in the next chapter.
The output of the FMU (osi3::SensorData) will be transferred via network using standard VTD socket connection mechanism ("sendMessage()" in the following example). The size of the message is encoded at the beginning of the transmitted data block. The following code is used inside the Plug-In (mSensorDataOut is of type osi3::SensorData):

```
        const size_t bufLength = mSensorDataOut.ByteSize() + sizeof(google::protobuf::uint32);
        char* buffer = new char[bufLength];

        google::protobuf::io::ArrayOutputStream arrayOutput(buffer, static_cast<int>(bufLength));
        google::protobuf::io::CodedOutputStream codedOutput(&arrayOutput);

        codedOutput.WriteLittleEndian32(mSensorDataOut.ByteSize());
        mSensorDataOut.SerializeToCodedStream(&codedOutput);
        sendMessage(mPortRDBout, buffer, bufLength, 0.0 /*delay*/);
```

Using the Plug-In without FMU

The Plug-In can additionally be used without OSI3 Sensor Model FMU.



In this case the socket connection is used in exactly the same way as described above but instead of transferring the output of the FMU via network, the input which is also of type osi3::SensorData is sent. A typical use case for this configuration would be an external simulation component scheduling the Sensor Model or just some component using OSI3 data without OSI Sensor Model Packaging at all.

Supported content

Currently following elements of osi3::SensorData are filled by the Plug-In:

- sensor_view
  - global_ground_truth
    - moving_objects
    - occupants
    - stationary_object
    - traffic_lights
    - environmental_conditions
    - traffic_signs
    - lanes (partly)
- feature_data
  - lidar_sensor

Differences to OSMP

As stated above the sensor model interface deviates from OSMP by reading osi3::SensorData instead of osi3::SensorView. The differences to the original OSMP 1.0.0 DummySensor are contained in the following two files:

- OSMPDummySensor.cpp
- OSMPDummySensor.h

Note: For loading the OSMP DummySensor as FMU into VTD it needs to link against a shared version of the OSI library. Therefore line 6 in CMakeLists.txt has to be modified as follows:

```
set(LINK_WITH_SHARED_OSI ON CACHE BOOL "Link FMU with shared OSI library instead of statically linking")
```

Activating private logging might also be useful, line 8 in CMakeLists.txt:

```
set(PRIVATE_LOGGING ON CACHE BOOL "Enable private logging to file")
```

vtd160.png	(121 KB)	Marius Dupuis, 14.07.2014 14:09
vtd2adtf01.png	(22.9 KB)	Marius Dupuis, 27.07.2014 08:12
vtd2adtf02.png	(32.1 KB)	Marius Dupuis, 27.07.2014 08:12
FiguresManualO4.png	(143 KB)	Marius Dupuis, 12.08.2014 12:31
FiguresManualO5.png	(57.9 KB)	Marius Dupuis, 12.08.2014 12:31
FiguresManualO_SHM.pdf	(248 KB)	Marius Dupuis, 02.01.2015 13:08
dynLinkRDB.png	(61.3 KB)	Marius Dupuis, 01.07.2015 10:29
vtd233.png	(65.7 KB)	Marius Dupuis, 20.07.2015 15:29
slin.png	(54.2 KB)	Marius Dupuis, 24.08.2015 16:13
vtd250.png	(45.4 KB)	Marius Dupuis, 24.08.2015 17:52
slout.png	(52.1 KB)	Marius Dupuis, 06.04.2016 21:14
sampleGenericPlatformRDB.tgz	(37.3 KB)	Marius Dupuis, 06.04.2016 21:57
sampleClientsRDBandSCP.tgz	(77.3 KB)	Marius Dupuis, 06.04.2016 22:00
sampleClientSimulink.tgz	(70.8 KB)	Marius Dupuis, 06.04.2016 22:14
rdbHandler.20170102.tgz	(49.7 KB)	Marius Dupuis, 02.01.2017 20:18
vtd468.png	(921 Bytes)	Marius Dupuis, 22.02.2017 20:45

vtd469.png (8.02 KB) Marius Dupuis, 22.02.2017 20:47
vtd470.png (9.73 KB) Marius Dupuis, 22.02.2017 20:48
vtd497.png (15 KB) Marius Dupuis, 22.05.2017 20:04
rdbTrigger.20170522.tgz (51.8 KB) Marius Dupuis, 22.05.2017 20:10
vtd.2.0.3.addOns.RDBCopy.20170802.tgz (110 KB) Daniel Wiesenhuetter, 02.08.2017 13:56
vtd.2.0.3.addOns.RDBCopy.20180123.tgz (109 KB) Daniel Wiesenhuetter, 23.01.2018 11:13
vtd.2.1.0.addOns.RDBCopy.20180123.tgz (109 KB) Daniel Wiesenhuetter, 23.01.2018 11:13
rdbSniffer.tgz (683 KB) Marius Dupuis, 02.03.2018 14:38
ScenarioEditor01.png (36.5 KB) Robert Wierl, 25.04.2018 11:10
ScenarioEditor02.png (30 KB) Robert Wierl, 25.04.2018 11:11
ScenarioEditor03.png (29.1 KB) Robert Wierl, 25.04.2018 11:11
ScenarioEditor04.png (59.7 KB) Robert Wierl, 25.04.2018 11:11
ScenarioEditor05.png (42.4 KB) Robert Wierl, 25.04.2018 11:11
objectCoord01.png (86.7 KB) Marius Dupuis, 27.04.2018 20:17
vtd.2.1.0.addOns.fileReader.20180510.tgz (584 KB) Marius Dupuis, 10.05.2018 11:47
sampleShmReaderWriter.tgz (55.4 KB) Marius Dupuis, 20.06.2018 05:47
vtd_ros_gateway.tgz (56.1 KB) Andreas Biehn, 28.06.2018 13:12
rviz.png (87 KB) Andreas Biehn, 28.06.2018 13:14
osi3_mm2.jpg (27.2 KB) Andreas Biehn, 18.09.2018 11:45
vtd.2.1.0.addOns.OSI.20180903.tgz (473 KB) Andreas Biehn, 18.09.2018 11:45
osi3_mm1.jpg (49.7 KB) Andreas Biehn, 18.09.2018 11:45
OSMPDummySensor.cpp - modified osi-sensor-model-packaging-1.0.0/examples/OSMPDummySensor/OSMPDummySensor.cpp, reading osi3::SensorData (31.5 KB) Andreas Biehn, 05.10.2018 09:29
OSMPDummySensor.h - modified osi-sensor-model-packaging-1.0.0/examples/OSMPDummySensor/OSMPDummySensor.h, reading osi3::SensorData (9.01 KB) Andreas Biehn, 05.10.2018 09:29
vtd562.png (44.9 KB) Marius Dupuis, 04.11.2018 19:21