Virtual Test Drive 2023.2

User Manual

HEXAGON

Table of Contents

Introduction

Virtual Test Drive (VTD) is a tool-chain and modular framework for the provision of virtual environments in engineering simulations for the automotive and railroad industry. This document provides an introduction to installing and operating VTD.

Since paper (even in electronic format) tends to be up-to-date only on the day of printing, we have established an extensive Wiki on our support website, which addresses the most recent questions and instructions and a lot of configuration info. The URL is: http://tracking.vires.com/

It is recommended that you become a registered user of this valuable resource. For access related details refer to Troubleshooting [7].

The FAQs are available on online Wiki and the hard copies of the Wiki provided with each release of VTD (see directory `VTD/Doc/Wiki`). For the online Wiki, please register at http://tracking.vires.com/.

The following abbreviations are used in this document:

CL	Closed-Loop
DIL	Driver-in-the-Loop
Ego	Own vehicle
GUI	Graphical User Interface
GSI	Generic Simulation Interface
IG	Image Generator
IOS	Instructor Operator Station
MM	ModuleManager
ROD	Road Designer
RDB	Runtime Data Bus
SCP	Simulation Control Protocol
SDK	Software Development Kit
SIL	Software-in-the-Loop
TC	TaskControl
VIL	Vehicle-in-the-Loop
VILSI	Vehicle-in-the-Loop Simulator
VTD	Virtual Test Drive

References

1. not for public
2. not for public
3. "Scenario Editor, User Manual", VI2008.027, VIRES GmbH
4. "ROD, Tutorial", VIRES GmbH
5. obsolete
6. obsolete
7. RDB_HTML, documentation of the RDB, created with doxygen, latest version, VIRES GmbH
8. SCP_HTML, documentation of the SCP syntax, Issue AL, latest version, VIRES GmbH
9. VTD Wiki, http://tracking.vires.com, latest version, VIRES GmbH

Troubleshooting

If you experience any kind of trouble or have any access related queries or questions, please use the following means in the indicated order:

- ***Wiki (VTD Online Documentation)*** **:** https://redmine.vires.com

> **NOTE**
> If you are not yet registered, do the following:
>
> 1. Go to the indicated website (https://redmine.vires.com).
>   You will be forwarded to the bug/feature reporting tool.
> 2. Click **Register** in the top right corner.
> 3. Create your own login and password.
>   When providing your e-mail address during the registration process, ensure it is a company e-mail address. We do not accept *private* registrations via general accounts like Gmail, etc.).
> 4. Wait for approval of your account.
>   We reserve the right to refuse approval without further explanation. Usually, we will send you an e-mail and ask for further references within your company so that we can also provide you with access to relevant subprojects.

The online *Wiki* provides FAQs, installation instructions, samples, components, and interface descriptions.

SimCompanion (Ticket System) **:** https://simcompanion.hexagon.com/customers/s/article/VTD-Support-Home-Page

> **NOTE**
> To get further information on how to enable the account or how to create tickets through SimCompanion, please have a look at the following page:
>
> https://simcompanion.hexagon.com/customers/s/article/How-to-log-a-case-ticket-using-the-knowledge-base-portal-Simcompanion
>
> If you face any issue or have additional question regarding the process, please have a look at our FAQ page:
>
> https://simcompanion.hexagon.com/customers/s/article/Logging-case-tickets-for-VTD-Support

SimCompanion provides ticket creation for **support assistance, bugs, and improvements requests.**

For technical issues, first consult the Wiki. To report a bug, to propose an improvement or a new feature, select N**ew Issue** and fill the fields you are familiar with. You may leave the rest of the fields blank. They will be filled in by our staff.

- ***Email :*** vtd.support@mscsoftware.com
  Please contact the VTD support team via email, only if you can't find the solution on *Wiki* or you cannot log a ticket on *SimCompanion* portal.
- ***Telephone (International):*** https://mscsoftware.my.site.com/customers/s/article/support-contact-information-kb8019304#Phone
  Use the Hexagon support number for your country in the list. Please reach us via telephone if you are not able to log a case ticket via *SimCompanion* or via *email*

- ***Telephone (Germany) :*** +49.8061.939093-0
  Please use this telephone number only for **urgent** issues and you are not able to log a case ticket via *SimCompanion* or via *email.*

Resources

The components of Virtual Test Drive are provided for authorized/licensed users in the following way:

- Download: `https://secure.vires.com/workgroups/vtd/<subDir>`
  (or a specific address you have been given by our staff)
- Login and password for a specific download area are available from us or your company's coordinator for VTD.
- The files on the server might be encrypted. A key for decryption will also be available from your coordinator, if applicable.

Repository Structure

The repository's structure is as follows:

`vtd/`

`ReleaseNotes.txt`	Notes concerning the current release
`vtd.x.y.z.[date].tgz`	Package with complete current distribution (x= major revision, y.z = minor revision)

`vtd/Adhoc/`

It contains ad-hoc fixes, test versions, preliminary versions, etc. This will usually become part of the next release. Only use the files in this directory after you have been instructed to do so by our staff.

`vtd/AddOns/`

It contains additional components that are not part of the common VTD distribution. These may include, for example, sound simulation, the OpenDRIVE Gateway, SDKs, and many other tools.

System Requirements

The latest recommendations for the system configuration can be found in our online Wiki at the address indicated above.

Your system should fulfill the following minimum requirements:

- 64bit Linux operating system (reference: SuSE 15.1+, Ubuntu 18.04 +)
- Nvidia graphics card
- RAM: 16+GB

VTD may also run on virtual machines (VMs). However, the image generator may neither reach full performance nor provide sufficient image quality since it relies on the native support of the Nvidia driver for rendering. This support cannot be guaranteed within a VM.

Installation

The software may be installed on systems consisting of 1 to n computers. In the following description, unless otherwise stated, a *single-computer* system is assumed to be present. More than one computer is usually found when high graphics performance is required or when multiple graphics output channels are to be generated.

For a complete description of various types of installation, please see the online documentation:

`http://redmine.vires.com/projects/vtd/wiki/VTD_Configuration_Issues`

New Installation

By default, VTD will be installed in the user space, and hence no root permissions are required.

Do the following for a new installation:

1. Within your user, create a dedicated directory for VTD
2. Download the complete distribution
3. De-crypt the distribution, if applicable

Single/Multi User Environment

For multi-user systems, we provide a script where common parts of VTD are installed under

```
/opt/VIRES/VTD.x.y
/var/VIRES/VTD.x.y
```

All files relevant to the individual user are installed in userspace. The script is provided separately

```
vtdInstallMultiUser.sh
```

Please run it with the following option to see its parameterization

```
./vtdInstallMultiUser.sh –h
```

Installing Standard Distribution Files (Single-User)

One of the new features of VTD 2020 and later, is an easier/more user-friendly installation of Virtual Test Drive to speed up the installation and to improve the usability of our product. To realize this feature, we developed a standalone installer with the InstallAnywhere-Assembly by Flexera software.

The pre-installation process before using the VTD installer is as following:

1. Get the right hardware.
   `https://redmine.vires.com/projects/vtd/wiki/VTD_Configuration_Issues#Hardware`
2. Install the operating system.
   `https://redmine.vires.com/projects/vtd/wiki/VTD_Configuration_Issues#Operating-System`
3. Install additional required packages.
   `https://redmine.vires.com/projects/vtd/wiki/VTD_Configuration_Issues#Additional-required-packages-for-VTD-20191-and-later`
4. Verify or install the accelerated graphics driver (NVIDIA).
5. Download the needed installer.
6. Execute the file.

Additional Packages can be installed as follow:

Run.Control → install Dev.Control
Run.Perception → Dev.Control and/or Dev.Perception
Service Pack

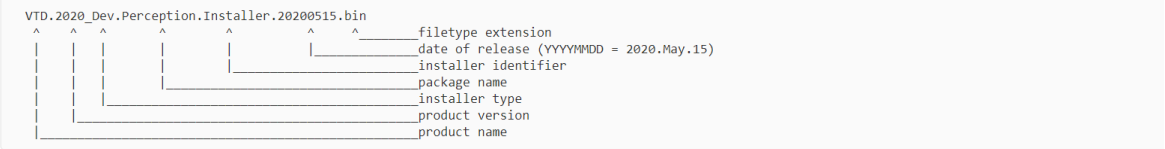
The installer will get distributed over the regular repositories as our regular software archives in the past.

Installer File

The installer gets delivered as a standalone binary file. Depending on the installer type-specific size of this standalone file, we recommend a good broadband internet connection to download it. It's named after the following identifiers:

`PRODUCT-NAME.PRODUCT-VERSION_INSTALLER-TYPE.PACKAGE NAME.Installer.RELEASE-DATE.bin`

Example



Installer Types/Packages

Four different versions of the VTD installer are available as of now. They can be differentiated further in *Run-Installations* (main software installation) and *AddOns*.

- Run.Control
- Run.Perception
- Dev.Control
- Dev.Perception
- Service Packs

The installer types were derived from the existing packaging convention. You can explore their content following this link:

`https://redmine.vires.com/projects/vtd/wiki/VTD_Migration_2_2_0_to_2019_1#Packaging`

Installing the License

With the 2019.1 version of VTD, it introduced a new licensing implementation that is based on FlexLM. Locally installed license files in general work like in the old licensing implementation. For server-based license files, there is a different license manager that has to be installed.

License Type

There are three different license types:

- **cnodelock** indicates a counted, node locked, network license
- **unodelock** indicates an uncounted, node locked, local/network license
- **seats** are a seat-based network license.

Using a Local License

Local license files can be stored as `[VTD]/bin/license.dat`, and VTD automatically uses that license file.

Anyway, if there is a necessity to use the license file with another name and/or with another path, there is the possibility to configure this manually.

Inside the `[VTD]/bin/vtdStart.sh` file there is a licensing section:

```
# Licensing
# By default VTD will check for '[VTD-PATH]/bin/license.dat'. In every other case
  the location of the
#  license file has to be specified:
#  i.e. use 'export VIRES_LICENSE_FILE="/opt/licenses/VTD-license.dat"'
   for using a local license
#  i.e. use 'export VIRES_LICENSE_FILE="27500@LICENSE-SERVER"'
   for using a license server license
#export VIRES_LICENSE_FILE="27500@LICENSE-SERVER"
```

As in the section itself described, add a line below this section which looks like this:

```
export VIRES_LICENSE_FILE="/[PATH_TO_LICENSE_FILE]/[LICENSE_FILE_NAME]"
```

Please replace `"[PATH_TO_LICENSE_FILE]"` and `"[LICENSE_FILE_NAME]"` with the correct values.

Using a Server License

Server licenses cannot be used locally; it is necessary to install a license server. For custom and server license files, do the following:

1. Activate the variable VIRES_LICENSE_FILE inside the "[VTD]/bin/vtdStart.sh" file in the licensing section.

   ```
   # Licensing
   # By default VTD will check for '[VTD-PATH]/bin/license.dat'.
     In every other case the location of the
   #  license file has to be specified:
   #  i.e. use 'export VIRES_LICENSE_FILE="/opt/licenses/VTD-license.dat"
      for using a local license
   #  i.e. use 'export VIRES_LICENSE_FILE="27500@LICENSE-SERVER"
      for using a license server license
   #export VIRES_LICENSE_FILE="27500@LICENSE-SERVER"
   ```

2. Add a line below this section which is as follows:
   `export VIRES_LICENSE_FILE="[PORT]@[LICENSE-SERVER]"`
3. Replace `"[PORT]"` and `"[LICENSE-SERVER]"` with the correct values.
   `"[LICENSE-SERVER]"` can either be an IP address or a hostname.

License Server Installation

The License Manager is used as the license server. It can be installed either on Windows or Linux operating systems.

1. Download the License Manager from the Software Download Center (https://mscsoftware.subscribenet.com/control/mnsc/download?element=11042537).
2. Register for an account if necessary.
3. Access the license server and its user guide through the download page.

Troubleshooting

If the installation does not use `vtdStart.sh` to start VTD, it is necessary to provide the variable `"VIRES_LICENSE_FILE"` with the correct value in the system environment.

AddOns

Your distribution may contain additional components (e.g., 3rd party vehicle dynamics, OdrGateway, sound, etc.). You will be notified by us which packages you have to download in addition to the base package.

To install the additional components, perform the following for each of them:

1. Download your add-on archives
2. De-crypt, the archives, if applicable
3. Unpack each archive in the directory, which also contains the directory „VTD.x.y“.
4. Follow specific rules for individual packages, if applicable

Ad-hoc Updates

If an ad-hoc update is available, you have to read the release notes and decide whether to start with a complete distribution or only to update some components.

In both cases, the following mode of operation is recommended:

1. Backup all data that has been modified by the user (that deviates from the standard distribution).
   We recommended that you first make a backup of the entire active distribution in its current state (just in case you forgot to add some files to the partial backup).
2. Download the update file.
3. De-crypt the update file.
4. Unpack the file in the directory, which also contains the directory „VTD.x.y“.
5. Only in exceptional cases will the installation/unpacking have to take place in a directory other than the one noted in 4. In these cases, you will be given individual instructions.

Configuration

The overall configuration and data organization is distributed over three different levels:

1. Distributions (`Data/Distros`)
2. Setups (`Data/Setups`)
3. Projects (`Data/Projects`)

Configuration and simulation data for run-time components may be found in all three levels. A so-called *file finder* is used to locate data, searching for it from the specific (project) level to the general level (distribution).

- **Distributions**: A distribution contains the basic configuration and simulation data (e.g., visual databases), which applies to all kinds of setups and projects (see below). We organize the data upon packing of the release. **No user-data** shall be placed within the `Distros` directory. The applicable distribution is symbolically linked to `Current` in the directory `Data/Distros`. This link may be changed by invoking the script `selectDistro.sh` in the same directory
- **Projects**: The actual user and test data are contained in the sub-tree Projects. Here, the user may create additional subdirectories for each project. The GUI will guide the user during the process of creating new or activating existing projects.
- **Setups**: They represent different environments in which VTD is supposed to run. With the introduction of the setups, a stronger concept of separation of platforms from project data has been realized.

The following setups are part of the default distribution:

`Standard`	desktop installation, single machine
`Standard.noIG`	desktop installation, single machine, without image generator
`Joystick`	for joystick / game-wheel operation
`DualHost`	dual-channel operation using two separate hosts
`DualIGDualPlayer`	two concurrent IGs on a single machine simulating two players' eyepoints, driven by two vehicle dynamics instances
`Stereo`	stereo setup of IG on a single host

Other distributions may include additional setups like:

`Standard.HDR` : HDR configuration of an image generator

Upon delivery, the `Standard` setup is configured as the current setup.

Switching between setups may be performed optionally upon starting VTD:

1. Go to the `VTD.x.y/bin` directory
2. Execute `./vtdStart.sh -select`.

New setups may be created by recursively copying the setup template (or another one that comes close to the new setup's target configuration) to a new name. When copying recursively, make sure that symbolic links are preserved. The following command will do just that in a Linux shell:

```
cp –R Template MyNewSetup
```

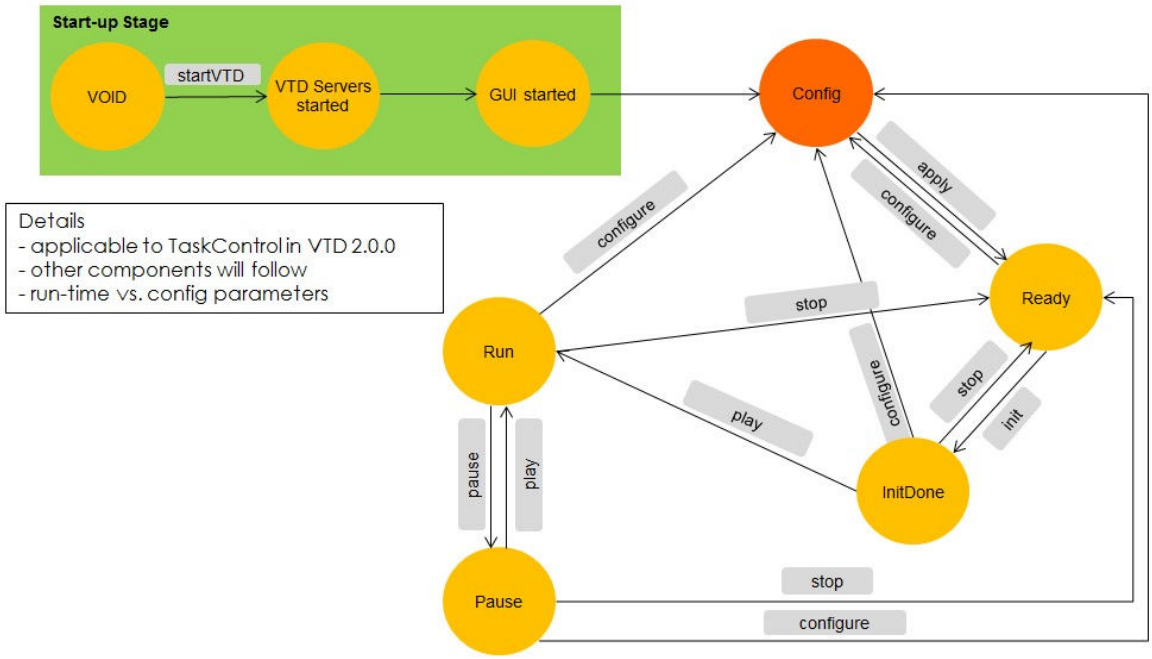
Getting Started

You may start the system either with or without the GUI (e.g., from within your test automation system).

Operation Stages

Starting with VTD 2.0, a new stage **Config** has been introduced, which will provide means to configure tasks interactively employing a so-called **Parameter GUI**. The first component for which this stage is relevant is the TaskControl. Based on experience with this component, we will migrate all other components to the new mechanism.

The following figure gives an overview of the simulation stages:

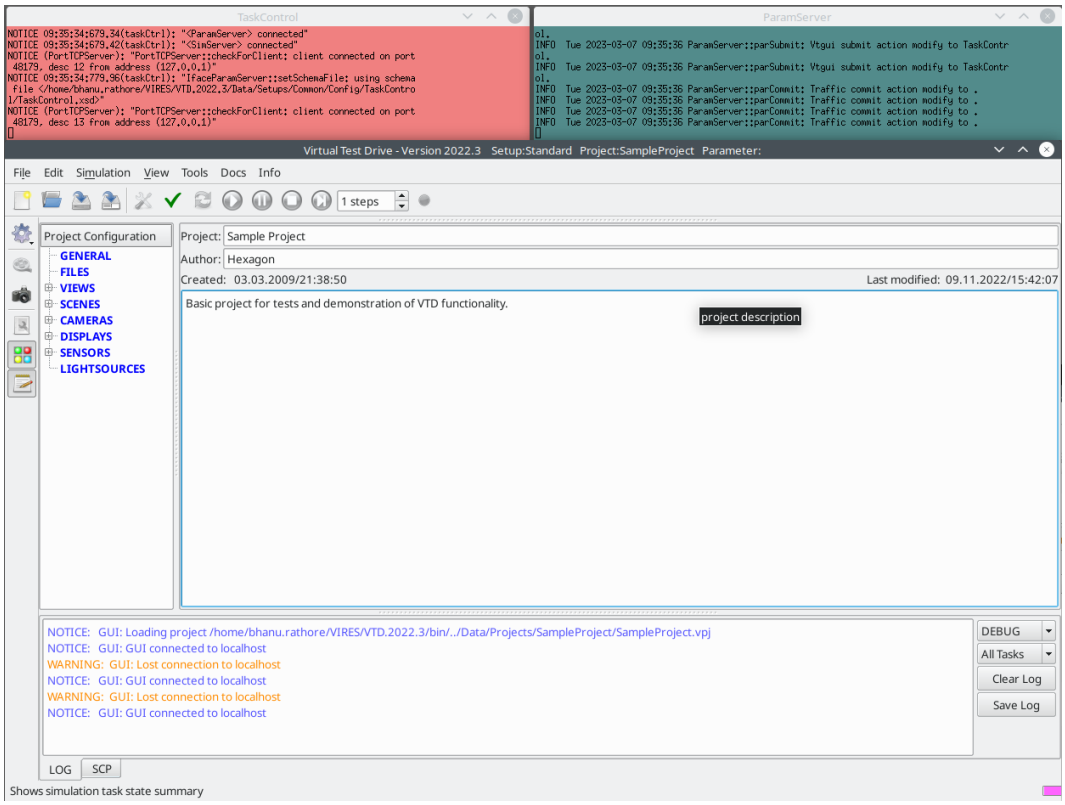


Quick Start Version

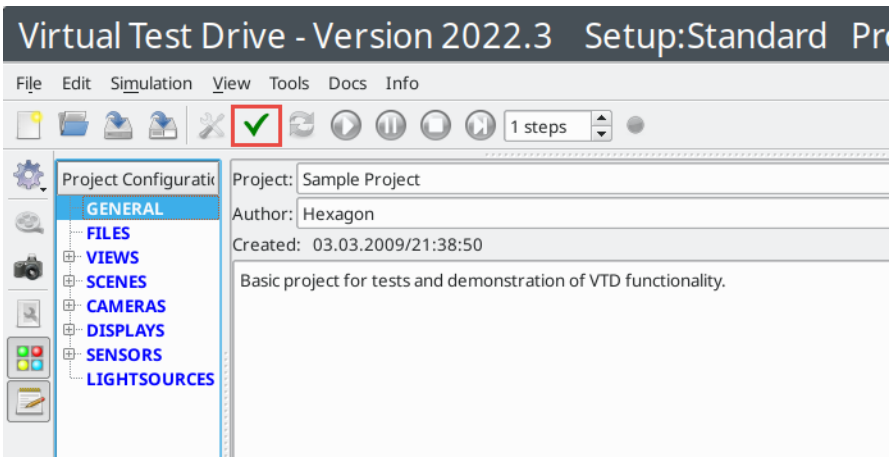
Please perform the following steps:

1. Go to the `VTD.x.y` directory.
2. Execute one of the following:
   a. `bin/vtdStart.sh`
   b. `bin/vtdStart.sh -select` (for selecting the applicable setup)

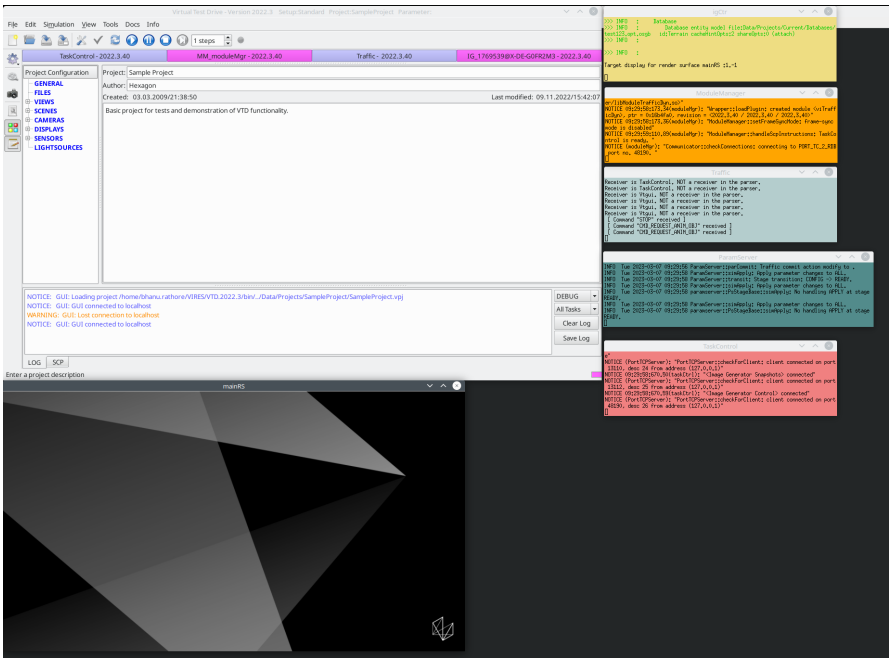
   This will start VTD in the Config mode. **TaskControl** (red window) and **Parameter Server** (blue window) will be displayed:



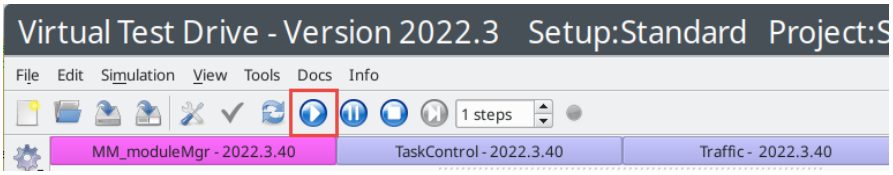
3. If you do not want to change any of the default settings of the current project and instead want to run VTD immediately, do the following:
   a. Press **APPLY**.



The TaskControl (TC) will be configured and all remaining components will be loaded.



b. Press the **PLAY** button (the IG may be restarted automatically).



Now, you can enjoy your ride.

Slow Starting Version

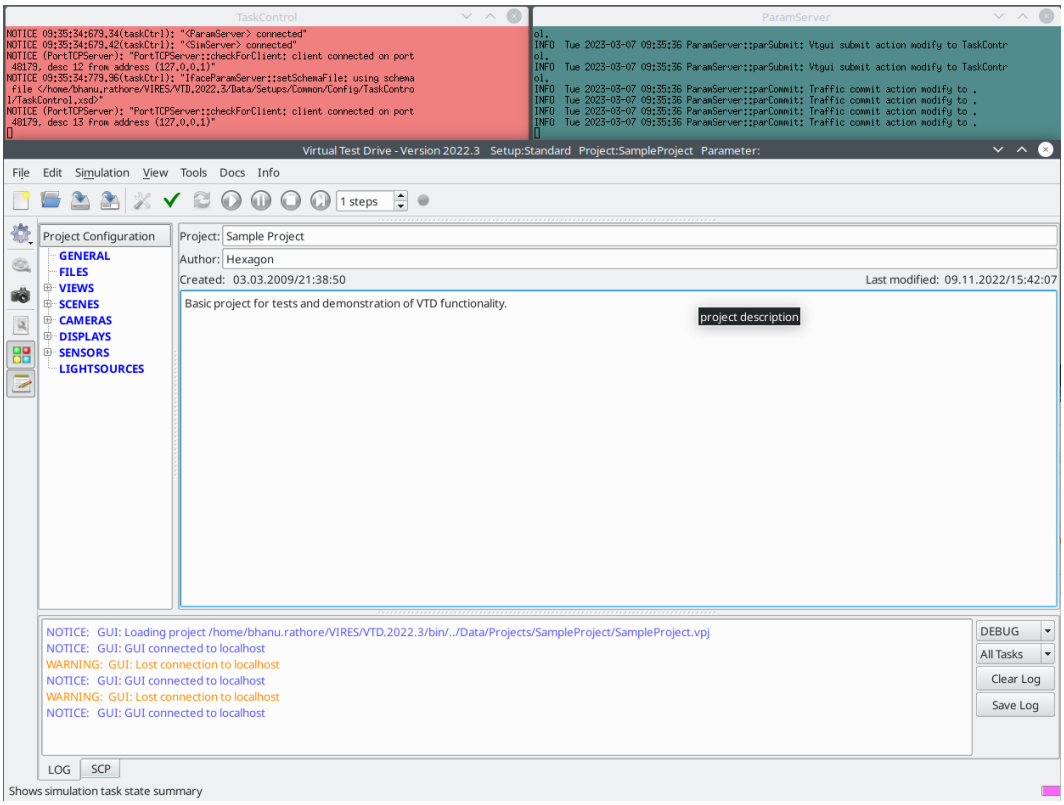
Configure > Start > Reconfigure...

With VTD 2.0, we started migrating the configuration of VTD's components from dedicated configuration files to better serviceable user configuration files. These will be easier to migrate between different setups and projects. Since VTD 2.0.0, the TaskControl may be configured with the new mechanism.

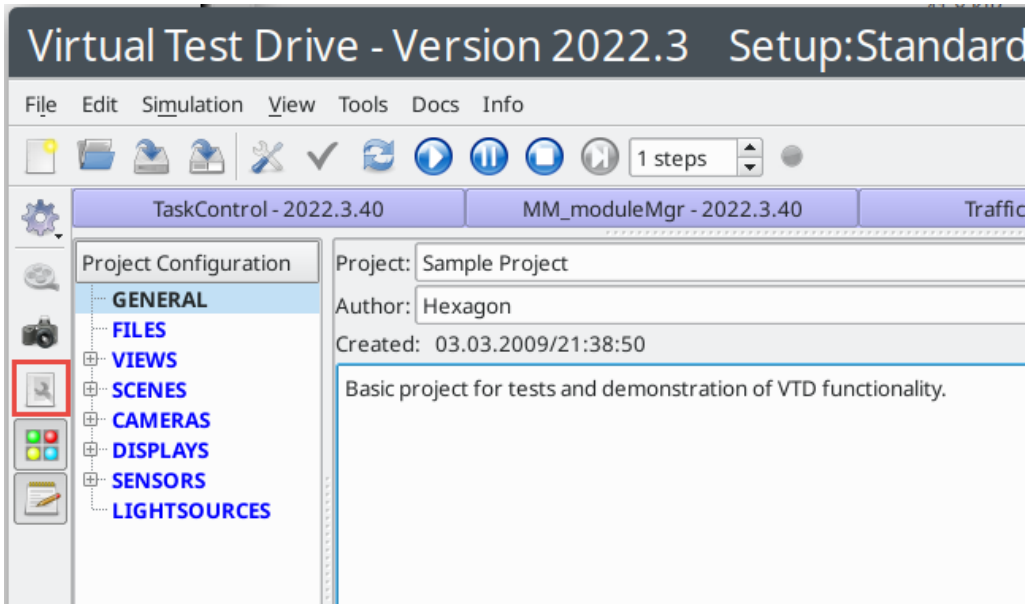
Perform the following steps:

1. Go to the `VTD.x.y` directory.
2. Execute one of the following:
   - `bin/vtdStart.sh`
   - `bin/vtdStart.sh -select` (for selecting the current setup)

   VTD starts in **Config** mode. TaskControl (red window) and Parameter Server (blue window) will be present.

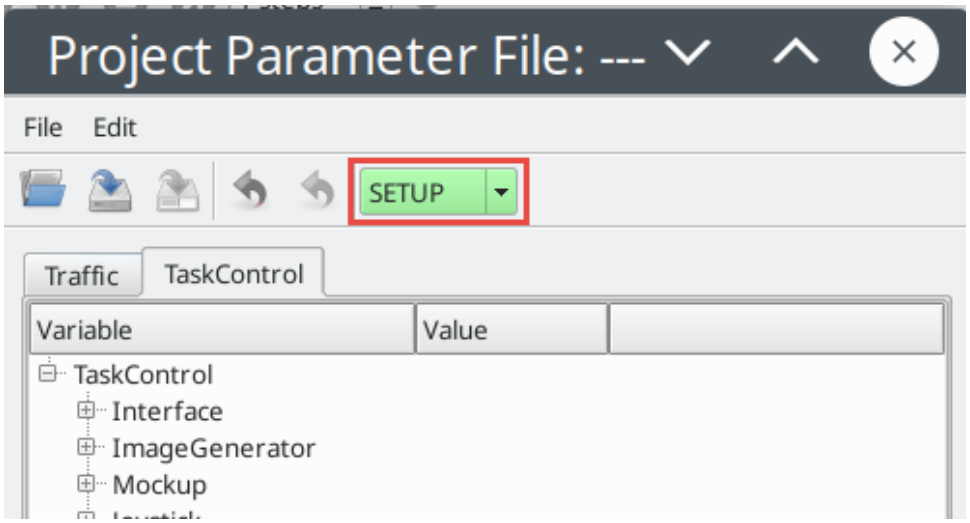


3. Click **Show Parameter Browser**.

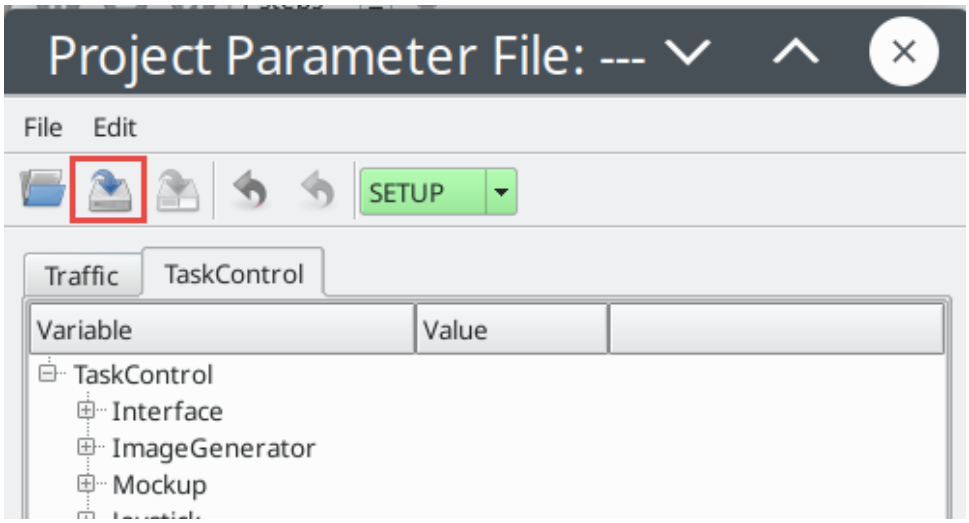


The **Component Parameter** Browser opens. In the top section you will see a toggle button for **Project** and **Setup**.

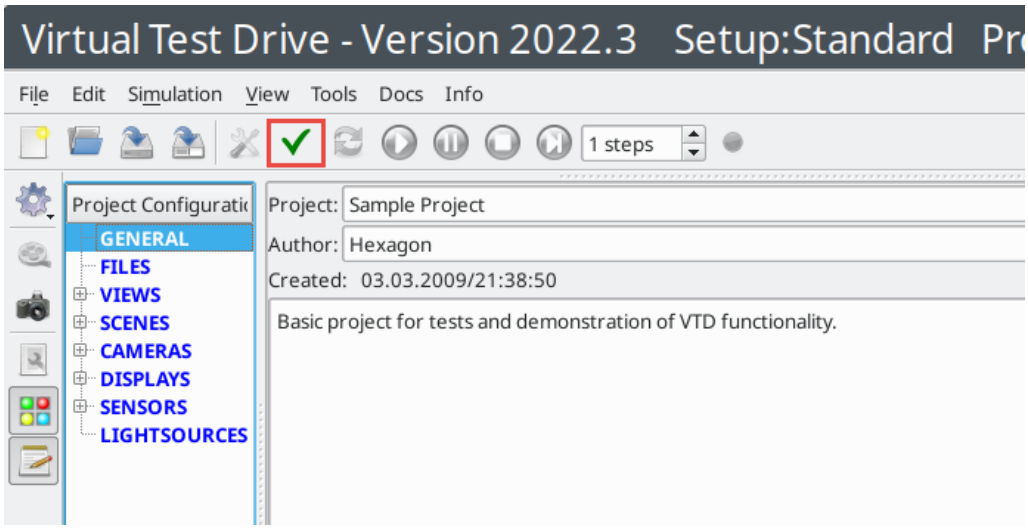
4. Select the **SETUP** mode. Expand the tree view as required.



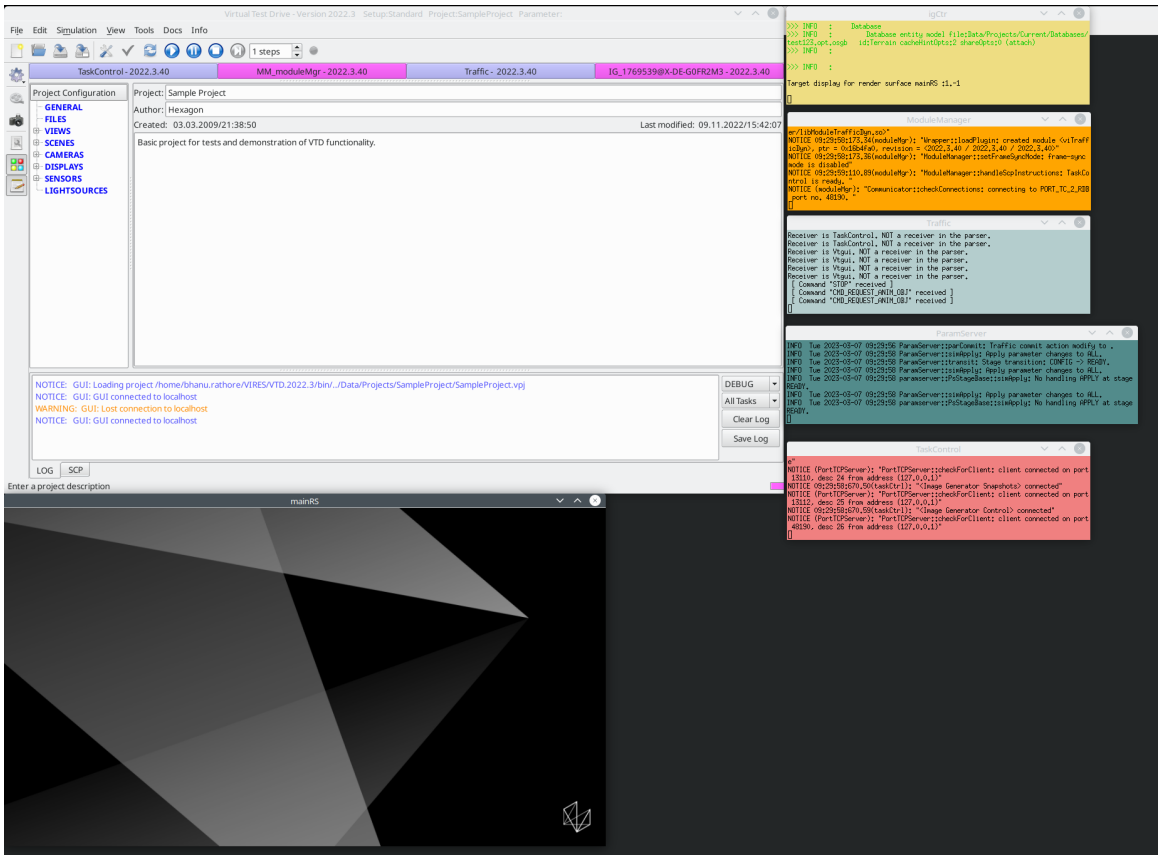
5. Adjust the parameters as required for your configuration.
   - New parameter settings will be displayed in red color until confirmed by the corresponding component (here: TaskControl).
   - Parameters deviating from the default values will be shown in green (SETUP) or blue (PROJECT) color.
   - The font will be set to **bold** unless a parameter set has been saved to file (use the SAVE button in the top menu bar of the parameter dialog).



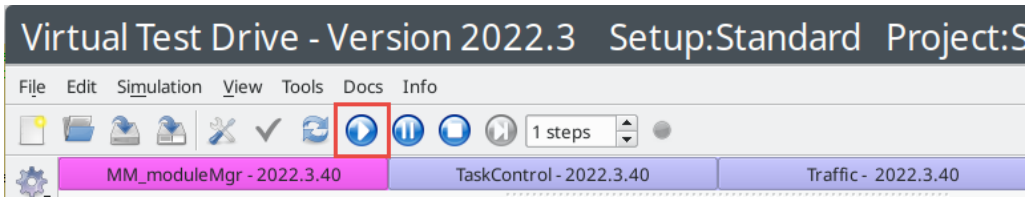
6. Close the **Parameter** Browser window after performing all settings.
7. Click **APPLY**.



The TC will be configured and all remaining components will be loaded. You are now at the same stage that you reached after executing load components in VTD.



8. Press **PLAY** (the IG may be restarted automatically)

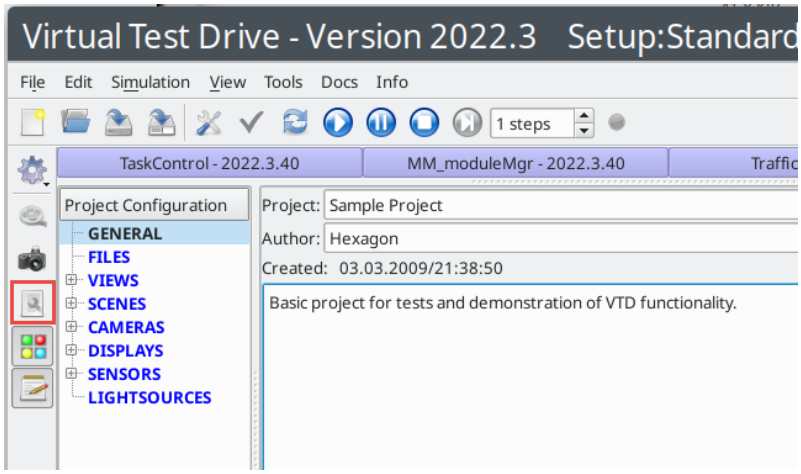


9. Either terminate the simulation or go back to the configuration stage.

Modifying Parameter at Run-time

Selected parameters of the components (here: TaskControl) may be modified at run-time. These parameters may be accessed by opening the Parameter Browser and changing the values accordingly.

1. Open the Parameter Browser window by pressing the button on the left side of the GUI



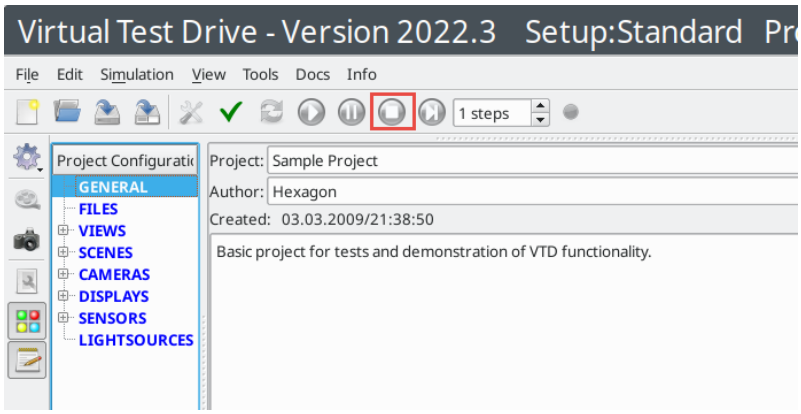
2. Change the value of the individual parameters. They will automatically be transferred to TC and will be confirmed by TC if the settings are valid.

Re-configuring the Simulation

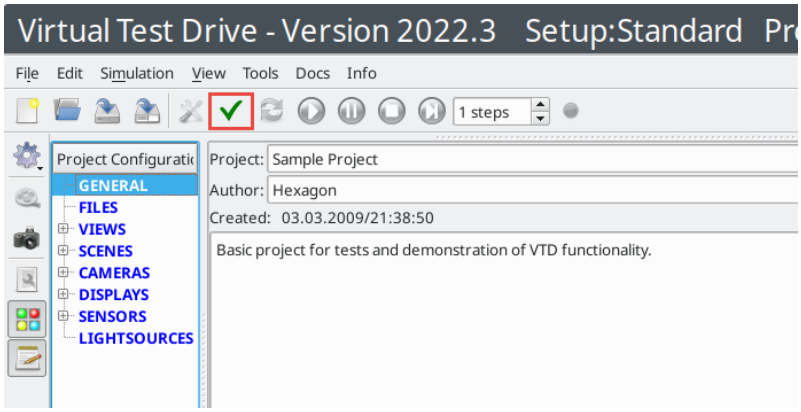
Several parameters of a component (here, TaskControl) will only be accessible during the configuration stage.

To transfer a running simulation back to the configuration stage, do the following:

1. Click the **STOP** button.

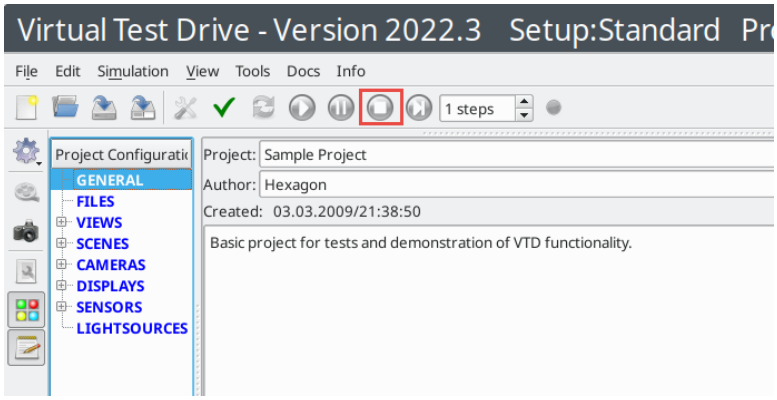


2. Click the **CONFIGURE** button.
   All processes but TaskControl, ParamServer, and GUI will be terminated. The **Parameter Browser** window will open.
3. Adjust the parameters • when done, Click **Apply**.

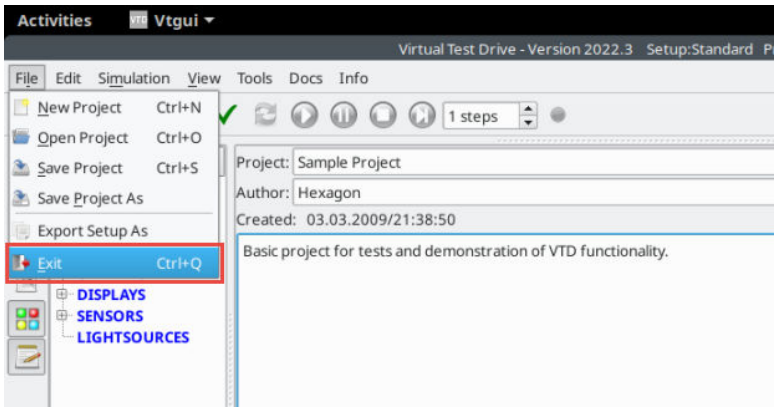


Terminating the Simulation

1. Click the **Stop** button in the GUI.



2. To shut down the simulation completely, select **File->Exit** from the GUI and confirm in the subsequent dialog.



3. Do the following if you want to kill the complete simulation:
   a. Open a shell.
   b. Go to the VTD root directory.

c. Stop VTD via script: `bin/vtdStop.sh`.

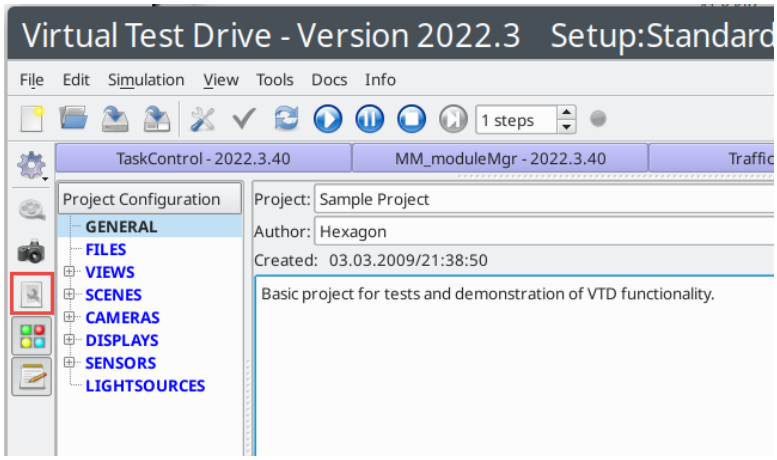
> **NOTE**
> This procedure may result in a loss of user data!

Saving/Loading Configuration Data

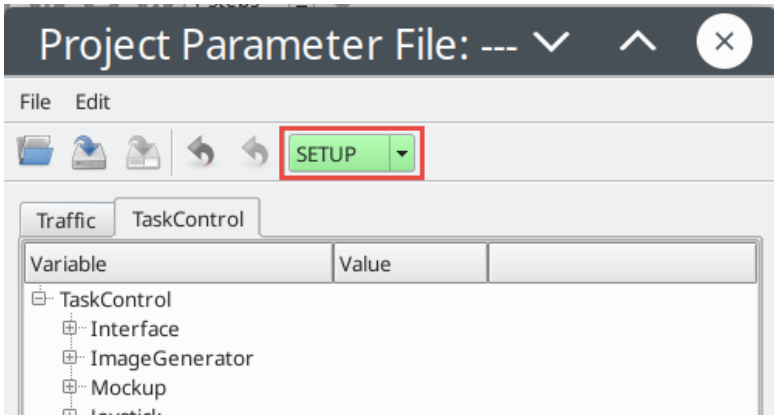
Configuration data (here: for TaskControl) may be saved in a setup- or project-related file at the user's discretion.

To **save** configuration data, do the following:

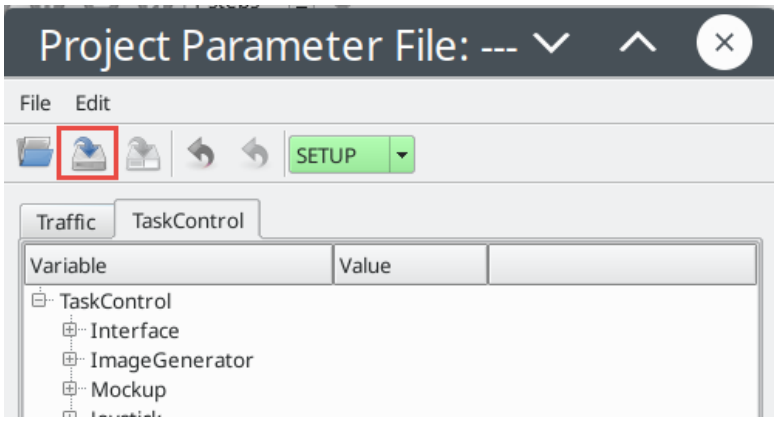
1. Open the Parameter Browser window by pressing the corresponding button on the left side of the GUI.



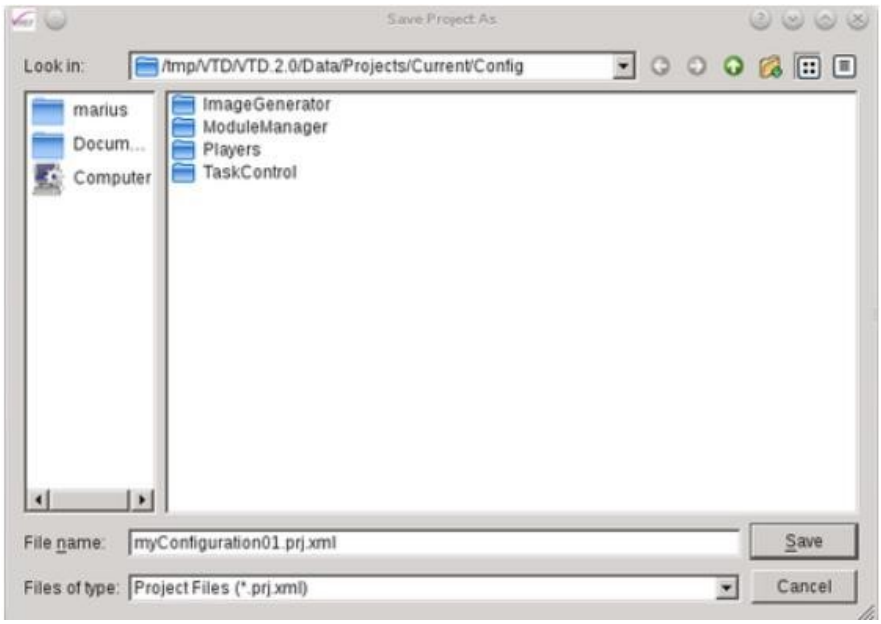
2. Select either **SETUP** or **PROJECT** mode.



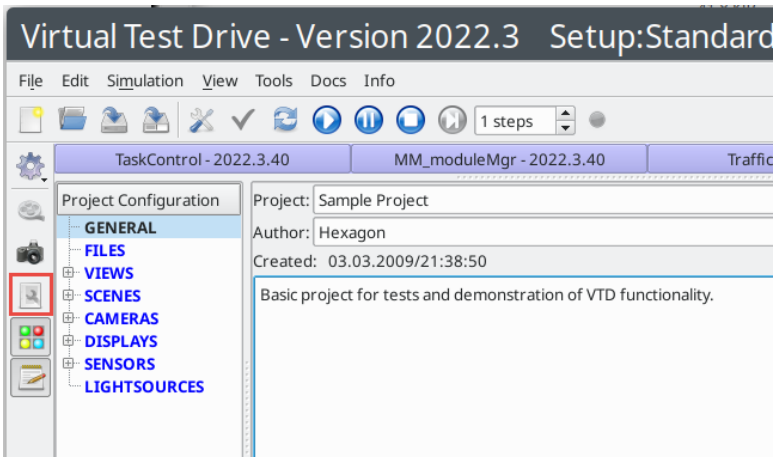
3. Modify parameters as appropriate and press the save button.



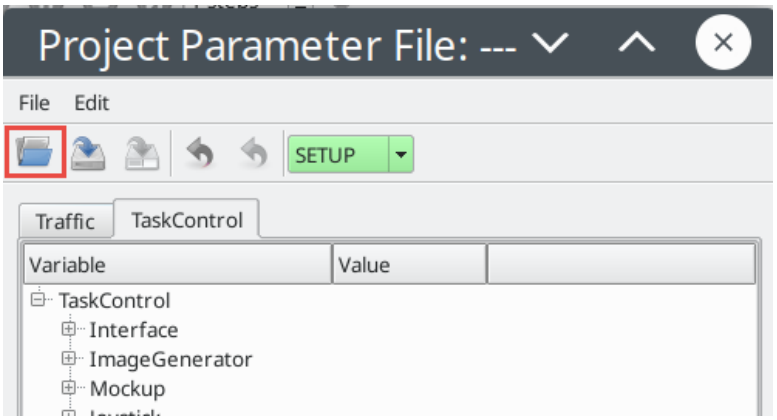
- If you save the configuration data from within the SETUP mode, the filename will be determined automatically, and the file will be stored in `VTD.x.y/Data/Setups/Current/ConFigure`
- If you save in the **PROJECT** mode, you may determine the filename by yourself:



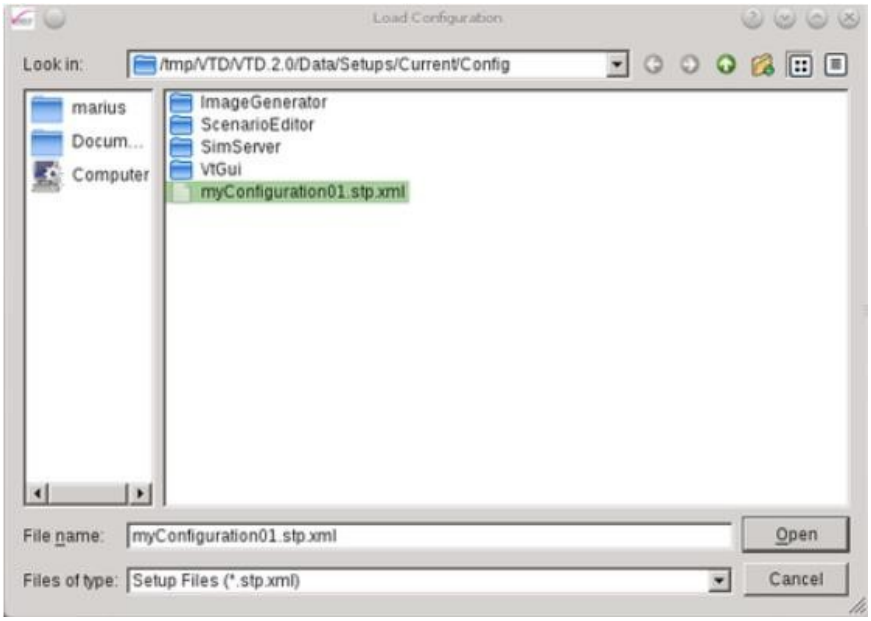
4. Do the following for loading a configuration file:
   a. Click the corresponding button to open the **Parameter Browser** window by on the left side of the GUI.



b. Click **Open**.



c. Select *from which context* you want to load the data (PROJECT vs. SETUP).
d. Select the file in the subsequent dialog (in PROJECT mode only; the filename is not configurable in SETUP mode, and an available file will be loaded automatically upon pressing the open button).



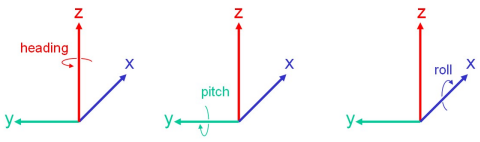
e. Load the configuration data (PROJECT mode only).

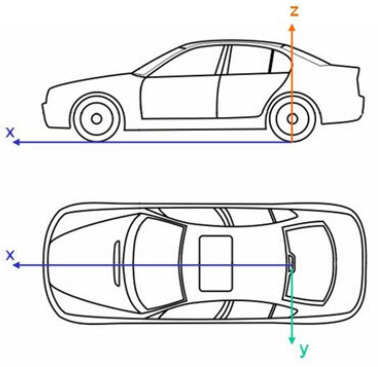
Starting from Command Prompt

Instead of using the GUI, you may directly invoke a start script.

Instructions for operating from the command line only are available in our online wiki (see **Advanced Setups** and **autoStart** or **autoConfig** therein).

Common Definitions

Units	All physical parameters, unless otherwise stated, will be stored and transferred in SI units.
Co-ordinate Systems	All co-ordinate systems are right-hand systems
Inertial Co-ordinates	The position of the inertial co-ordinate system is as follows: • x-axis: east • y-axis: north • z-axis: elevation Seen from the top (e.g., in ROD or the ScenarioEditior), the following orientation can be noticed: • x-axis: right • y-axis: up • -axis: to the viewer The internal coordinate system is represented as follows:  The inertial coordinate system is typically used for: position of players/objects in • space position of cameras in space • position of triggers in space position • of symbols in space The entire simulation is based on the inertial coordinate system. All other systems are derived from this system.

Vehicle Co-ordinates	The orientation of the vehicle coordinate system is as follows: • x-axis: forward • y-axis: left • z-axis: up The vehicle coordinate system is as follows:  The origin of the vehicle coordinate system in neutral loading conditions (e.g., spring deflection of all wheels is zero) is on street level at the center of the rear axle. The system is linked to the chassis so that elements positioned in vehicle co-ordinates will also perform, e.g., pitch and roll motions of the vehicle accordingly. The vehicle coordinate system is typically used for • position of sensors and detected objects • position of cameras • position of symbols

Screen Co-ordinates	The screen coordinate system is a two-dimensional system: • x-axis: right • y-axis: up  Screen is the rectangular region of the desktop (workspace window) managed by an application. The origin of the screen is in the lower-left corner. Screen coordinates are normalized, i.e., valid in a range between 0.0 and 1.0. The screen coordinate system is typically used for symbols. Other coordinate systems may be defined in additional documents that may be subject to nondisclosure restrictions and are not included in this documentation.

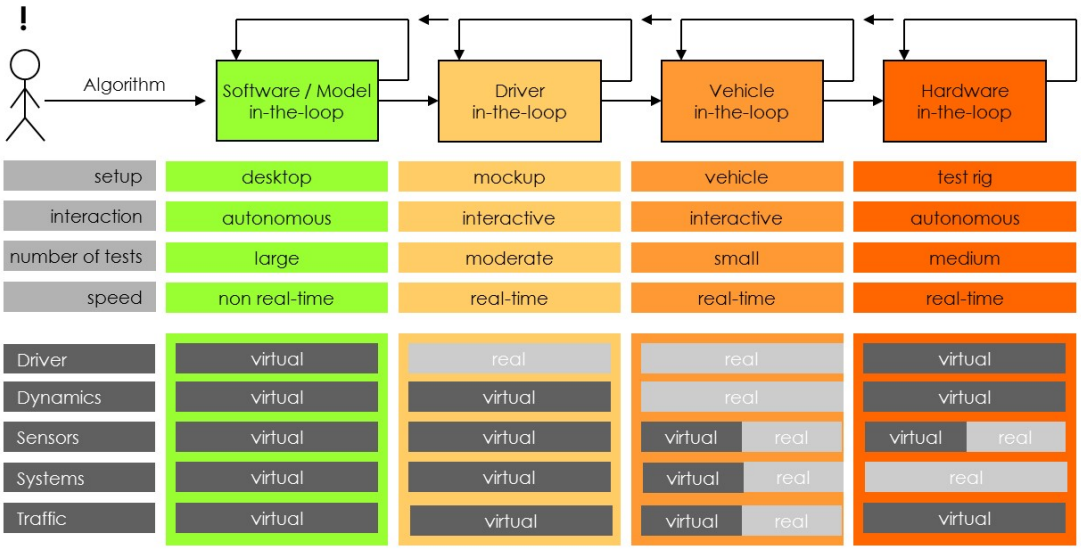
System Design

VTD was designed to fulfill the needs of engineering users involved in developing advanced driver assistance systems (ADAS) and active safety systems. These primary use cases are still the ones that shape the core design of VTD.

Runtime Environments

VTD can be configured to involve various software and hardware modules (see description of „setups“) above. Overall, the runtime environment may represent any of the following four use cases:

Software-in-the-Loop / Model-in-the-Loop:	simulation with vehicle dynamics and driver model
Driver-in-the-Loop:	simulation with vehicle dynamics and human in a mockup
Vehicle-in-the-Loop:	simulation with a real car and human driver
Hardware-in-the-Loop:	simulation with a hardware test bench, using again vehicle dynamics and driver model



According to the functional requirements of these environments, the involvement of VTD and 3rd party components into the overall systems varies. All key components listed as “virtual” in the above figure may be provided by VTD but do not necessarily have to.

VTD may be run in real-time and non-real-time modes with user-defined step sizes and even with individual simulation frame control.

Layout

The following figure shows the components that may be found in a VTD installation. Depending on the actual use case, additional components may be present, or some components may be missing.

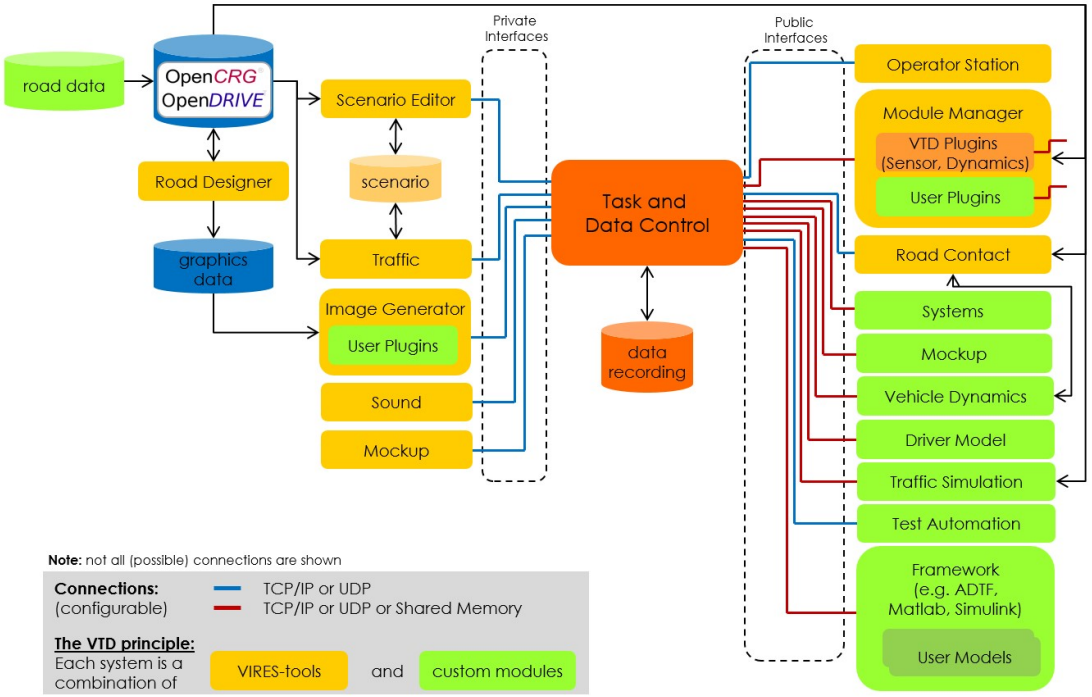


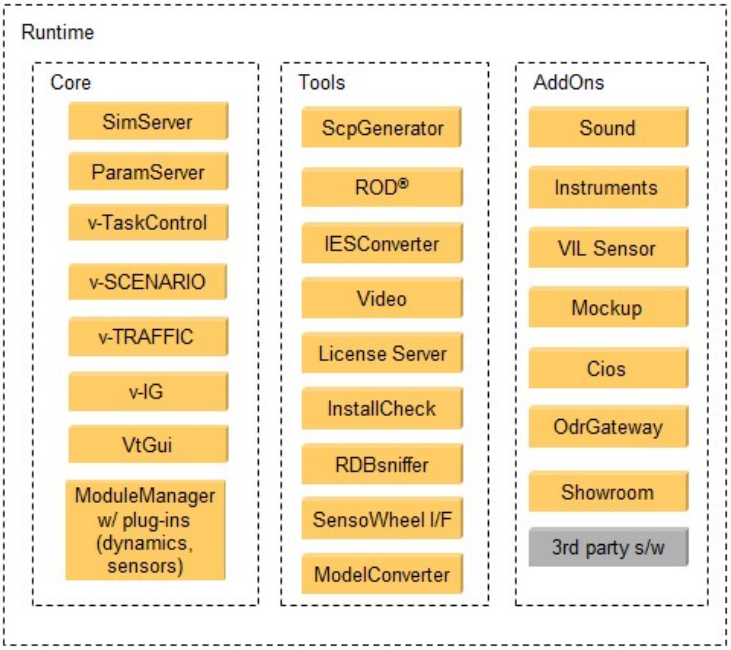
Figure 1. System Overview

Components

The system is organized into several categories of components. These are:

core components	the base components for standard setups
add-ons	optional add-on tools required for some setups
tools	utilities and other tools

The typical components making up a system are shown in System Overview [31]. Additional components may be available for certain setups. The following list describes the maximum (disclosed) extent of currently available components:



Core Components

SimServer:	simulation server; manages all VTD processes, one instance per host
ParamServer:	parameter server; manages all VTD configuration parameters, one instance per simulation
Vtgui (IOS):	graphical user interface, operator station
Image Generator (vIG):	display of 3d scenery and traffic scenarios; generation of video data
TaskControl (TC):	control of all data flows, synchronization of the simulation
ScenarioEditor:	configuration and monitoring of traffic and test scenarios
Traffic Simulation (Traffic):	animation of traffic (vehicles, pedestrians, objects) and scenarios (i.e., actions at run-time)
Module Manager (MM):	manager for plug-ins evaluating environment information (sensors) or providing dynamic inputs for externally controlled vehicles (i.e., vehicles not controlled by the traffic simulation). Users may write their plug-ins for the ModuleManager
Basic Vehicle Dynamics:	simulation of vehicle dynamics for the own vehicle, based on a single track model which is also used for all vehicles in the
Complex Vehicle Dynamics:	simulation of vehicle dynamics for the own vehicle, based on a five-mass model using the “bullet” physics engine; this vehicle dynamics runs as a plug-in of the ModuleManager

Add-Ons

Ego-Dynamics:	simulation of own vehicle’s dynamics, based on complex tools by 3rd parties
Instruments:	display / animation of driver’s central display (e.g. in mockup)
VILSensor:	Interface for VIL sensors (inertial, LaserBird)
Sound:	sound simulation of own vehicle and surrounding traffic
Cios:	customizable GUI which lets the user associate buttons with icons and simulation control commands which are issued upon pressing the button

OdrGateway:	tool for high frequency (e.g. 1kHz) query of the road at a small number of contact points; connected to vehicle dynamics via UDP ports
Showroom:	configuration tool for material properties of vehicles and objects

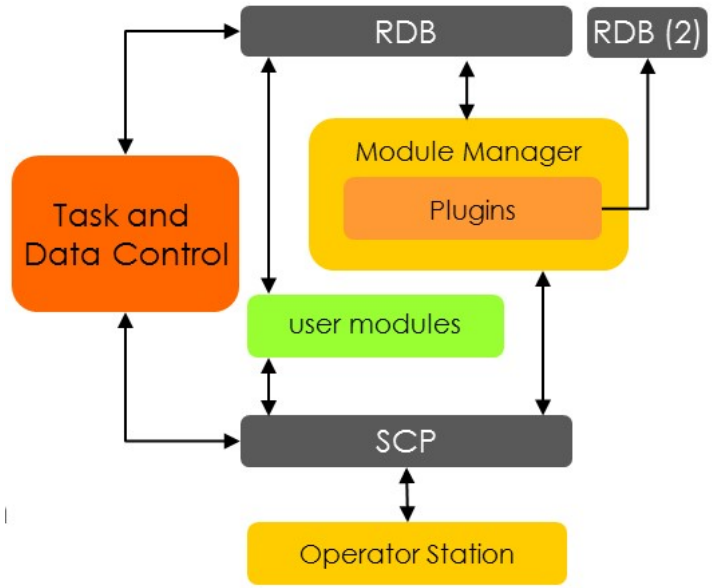
Tools:

Road Designer:	tool for the generation of visual and logical road networks
ScpGenerator:	internal test tool for automated / individual tests based on the SCP protocol
IESConverter:	Converter for the generation of light distribution textures from measured or computed files in .ies-format
Drivers:	drivers for license server dongles
LicServer:	license server daemon
Installation:	tools for checking a system's capability to run VTD
ModelConverter:	3d-model conversion tools so that custom 3d-models can run within VTD
RDBsniffer:	tool for monitoring and recording RDB data streams; may also be used for analyzing recorded RDB communication
RDBcopy:	tool for copying RDB data from a source (e.g. shared memory) to a sink (e.g., network); used for image transfer
Video:	tools for creating video files from recorded data
Showroom:	tool for the material configuration of 3d models (mainly vehicle models)

A detailed description of the components is given in Component Management [51], and following.

Interfaces

The components communicate via two categories of interfaces: private and public ones. The private interfaces are used for optimized internal communication of some VTD. The public interfaces are used for the broad majority of communication with either components provided by us or with 3rd party components.



The public interfaces are:

- **Runtime Data Bus (RDB)**: RDB is a unified network interface for all information available within VTD, which may be of interest to external components. Its binary communication protocol is documented (see Doc/ directory). Using RDB, run-time data may be exchanged directly between the VTD core components and, e.g., 3rd party tools. Two classes of RDB data streams are available: general run-time data and image data. Both streams run via dedicated network ports.
  So-called sensor plug-ins of the Module Manager may be used to extract data in an area of interest (sensor range) from the full RDB data stream; they will output their data also using the RDB protocol so that a user may connect his components either to the original RDB data stream (from TC) or to a sensor output (RDB(2) in the figure above).
- **Simulation Control Protocol (SCP):** SCP is a unified bi-directional network interface for configuration and command sequences. This interface's low-level protocol is based on a data stream that combines a binary header followed by ASCII data. The ASCII data is formatted as a human-readable stream with XML syntax.
  Multiple components may connect to the SCP data port. All commands sent from one participant into the system will be reflected in all other participants. Via SCP the user may control parts or all the simulation. The VtGui, which usually provides the user interface for simulation control also uses SCP as a means to connect to the simulation

Workflow

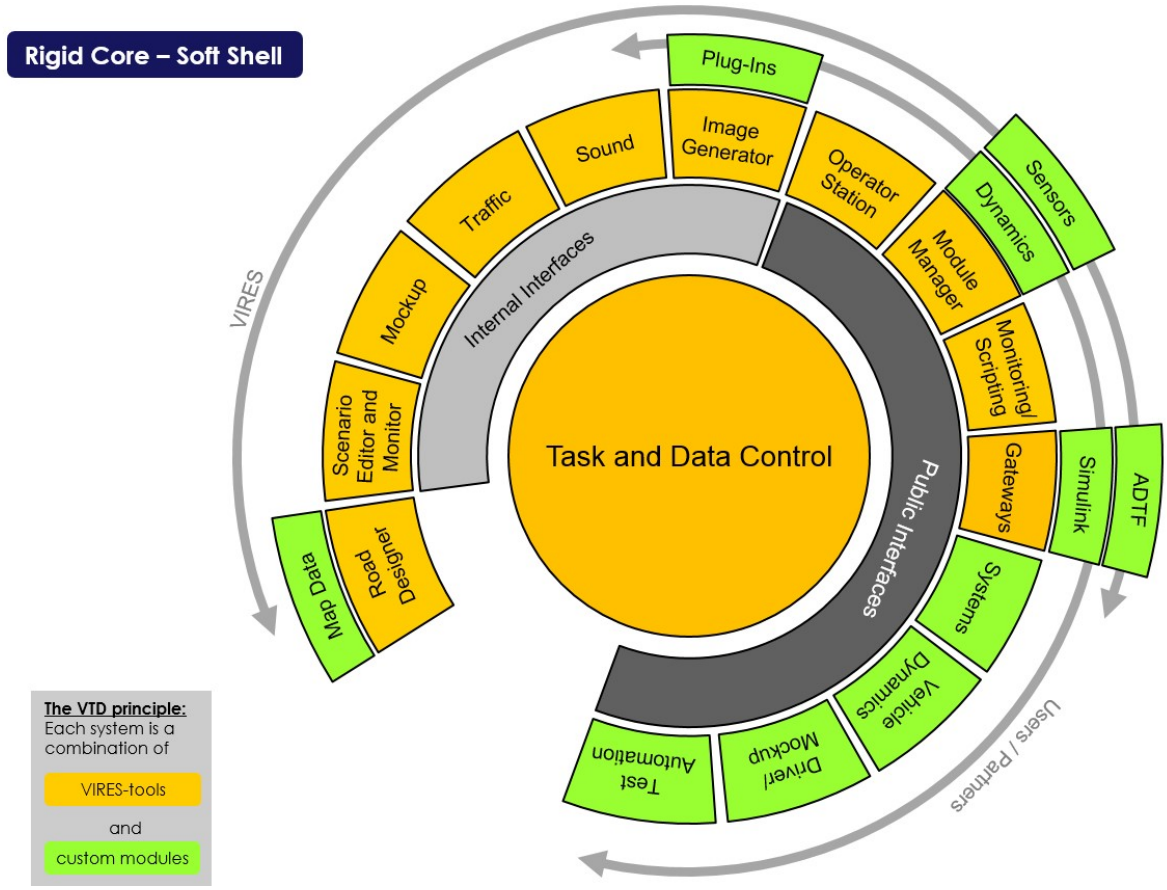


Figure 2. Components of VTD

The figure illustrates the basic workflow within VTD. It is split into three phases:

- Design
- Runtime
- Post-processing

The first two phases are part of VTD and are, therefore, subject to this documentation. The Post-Processing may either be realized as a real post processing, meaning that data is processed after the simulation (e.g., from a recorded simulation) or during the simulation by reading the public interfaces and processing the incoming data. As can be seen from the figure, the TaskControl is the core component for simulation control and data flow management. Keeping this property in mind is the key to understanding the communication and simulation control mechanisms' following description.

Design Phase

In the design phase, the user typically models road networks, creates traffic scenarios, and configures individual simulation settings in so-called projects.

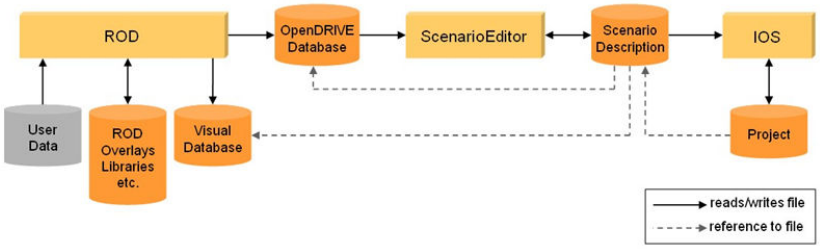


Figure 3. Components of the Design Phase

Figure shows the components of the design phase with their relevant data files. Usually, the components will refer to further files. These will be described below in the detailed view of each component.

The workflow during the design phase is as follows:

- With the RoadDesigner **ROD,** the user generates road networks that are to be used for the actual simulation task. The networks may be built from scratch or derived from user data that can be imported in different formats (see ROD manual). ROD saves its data in so-called „overlays“ and exports two files for the simulations:

OpenDRIVE description:	logical database (reference lines, lane definitions, signals, etc.) according to the OpenDRIVE de facto standard
Visual Database:	graphical representation of the simulation environment (3d databases)

- The **ScenarioEditor** reads the OpenDRIVE description of the road network from the file. Based on this information, the user defines the actual traffic scenarios, including control commands for the simulation, etc. The scenario description contains references to the two files created by ROD, which contain the database's logical and graphical representation.
- The **IOS** (VtGui) refers to the scenario description file. Based on this reference. It also stores additional configuration information, e.g., camera positions, display parameters, sensor configuration, etc. All configuration information will be stored in a so-called “project”.

Runtime Phase

In the runtime phase, the simulation is running, interacting with exterior components, and generating data. Each simulation session may be operated in one of three different modes:

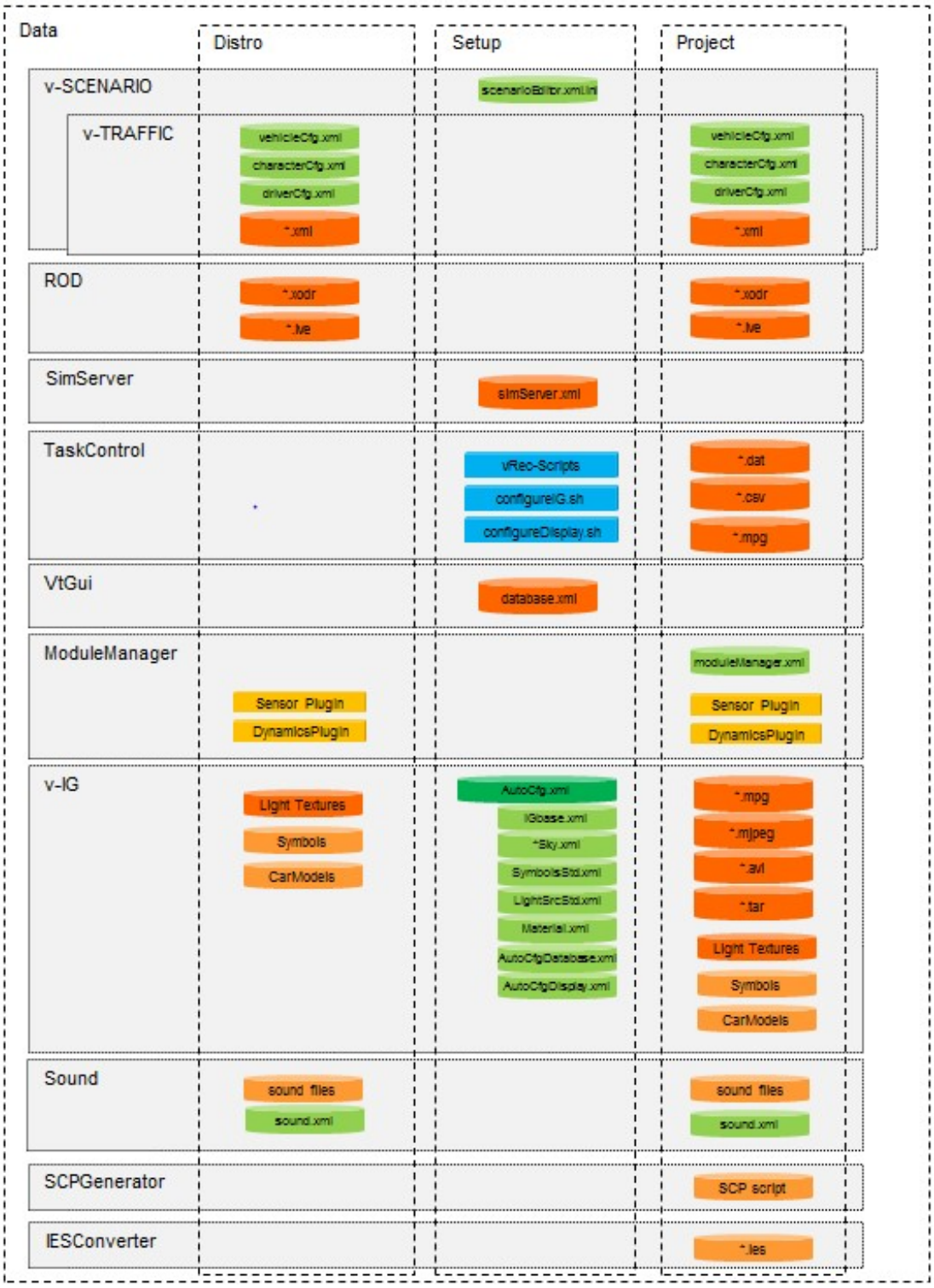
- In **Preparation** mode, a simplified model is used for the control of one dedicated (external) vehicle, usually representing the ownship (“Ego” car). A special panel will pop up on the IOS upon activation of this mode. This mode aims to provide an easy means for testing scenarios while still working on the overall scenario design. It may also be used for simple test cases where a simplified motion of the ownship is sufficient.
- In **Operation** mode, the simulation is started in its target configuration, i.e., with target mock-up, driver model (if applicable), and complex vehicle dynamics.

- In **Replay,** a recorded simulation is played frame by frame. This can be done in real-time, slow motion, fast forward, or single steps.

In all three operation modes, the interfaces of the components linked to the TC will be served with data in the same manner. So, it is not transparent for the components themselves, which one of the operating modes is the currently active one.

File Management

The following figure gives a comprehensive overview of files associated with certain components and their possible and recommended locations.



Configuration Files

The configuration files of VTD are used for the configuration of individual components. They will either be read during the start-up of a component or, finally, upon the start of a simulation. There is no central file that would refer to all required configuration files. Instead, each component tries to locate its corresponding configuration file within the directory structure using a *file finder*. Alternatively, a component may refer explicitly to a configuration file at a given path.

Starting with VTD 2.0, the individual components' configuration will be "streamlined" so that there are fewer config files involved. As a first step, there will be one central configuration file for the setup of the processes and their command line parameters. This file (`simServer.xml`) is stored in the current setup (`VTD.x.y/Data/Setups/Current/Config/SimServer/simServer.xml`)

Figure 4: Dependencies of configuration files and components [38] gives an overview of key configuration files. Details are given in the descriptions of the individual components.

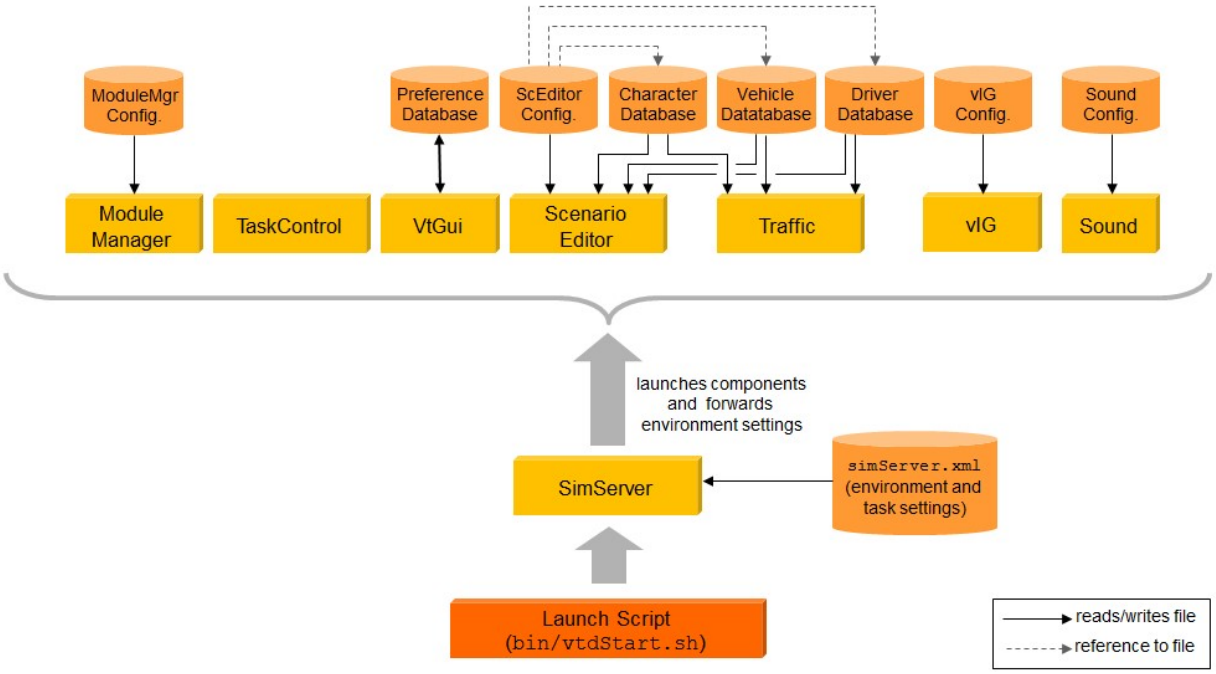


Figure 4. Dependencies of configuration files and components

Simulation Files

The simulation files are split into input files that first have to be read to perform the simulation and into output files resulting from the simulation (e.g., data and video recording). Figure 5 provides an overview:

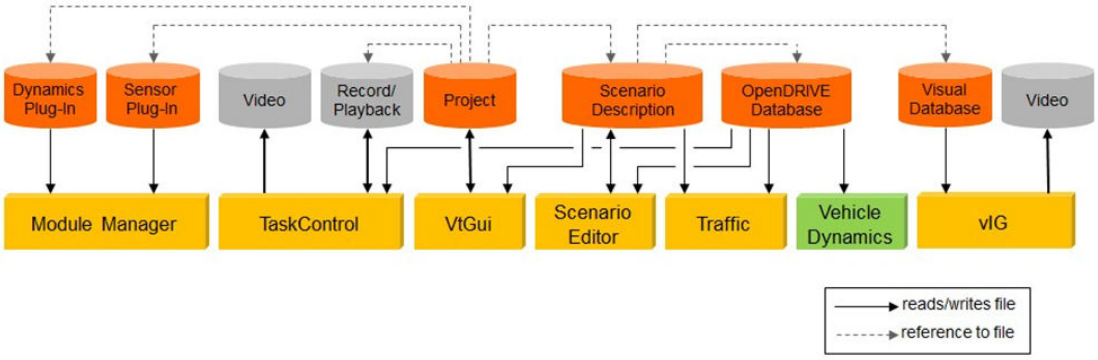


Figure 5. Overview of simulation files

All simulation files belonging to a certain simulation configuration (i.e., test series) are referenced directly or indirectly via a project file. Due to the referencing policy, it is required that backups be made of all configuration and simulation files to be able to restore a complete project. As a matter of portability, references are made using relative file paths. It is recommended to stick to this policy also for user-specific projects.

Dependencies of Configuration and Simulation Files

Figure 6 finally shows the dependencies of configuration and simulation files. To reduce the figure's complexity, some files that have been shown above and which only belong to single components are not shown.

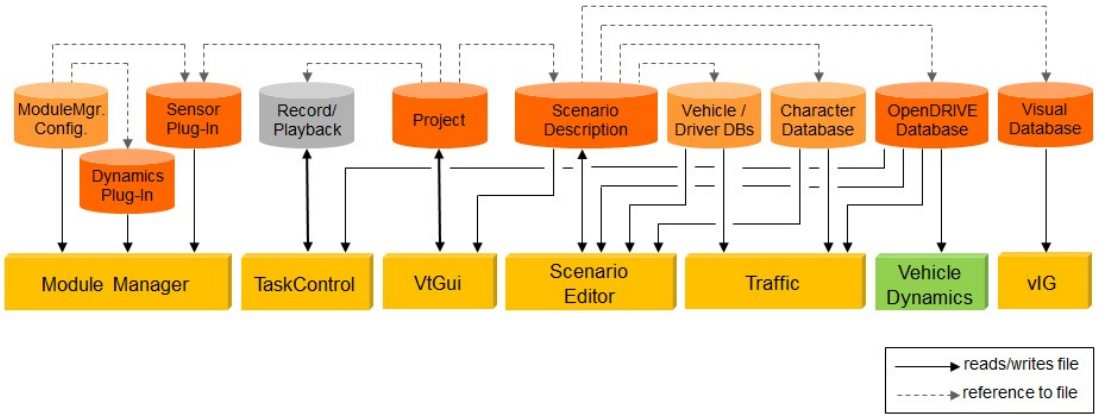


Figure 6. Dependencies of Configuration and Simulation Files

File Finding

As already noted above, the components (usually) locate their respective configuration and simulation files with a so-called "file finder". This tool parses a given search path for the first occurrence of a given file and provides the component with this file's location.

The search path parsed by the file finder is defined in:

`Data/Setups/Current/Config/SimServer/simServer.xml`

The required environment variables are:

- `lVI_FILE_PATH`
- `VI_FILE_SUB_PATH`

The file finder parses (in order of definition in the configuration file) all combinations of paths given by each entry in VI_FILE_PATH combined with each entry in VI_FILE_SUB_PATH.

Example

```
VI_FILE_PATH    a:b/c:e
VI_FILE_SUB_PA  .:x:y/z
TH
```

Resulting file search path:

```
a/.:a/x:a/y/z:b/c/.:b/c/x:b/c/y/z:e/.:e/x:e/y/z
```

Usually, the file finder used within the components will print a short notice into the shell about a given file's actual location.

NOTE

The file finder is **not** implemented in the Image Generator; therefore, the location of the IG's configuration files has to be given explicitly in the respective master configuration file (usually found at `Data/Setups/Current/Config/ImageGenerator/AutoCfg.xml`)

Directory Structure

The directory structure of VTD resembles the modularity concerning the applicable hardware setups and components. It is split into various levels, which will be described in this chapter. The description is based on a typical configuration of VTD. So, your actual configuration may contain additional entries or may have some entries missing.

Main Directories

The top levels of the directory are organized as follows:

```
<rootDir
|----bin                    start directory for VTD <---- START HERE!!
|----Data                   User and configuration data
|    |----Distros           the actual config / system data delivered
|    |    |----Current      the current distribution (symbolic link)
|    |    |----Distro
|    |----Setups            all system setups may be found here
|    |    |----Common       common settings for all setups
|    |    |----Current      the current system setup (symbolic link)
|    |    |    |----Bin     start directory for current system setup
```

```
|      |      |      |----  configuration files for current system setup
Config

|    |    |----Standard |

|    |----VIL

|    |    |----Template
|    |----Projects         all projects may be found here
|    |    |----Current     the current project (symbolic link)

|          |          |----
SampleProject

|    |    |----Default
|----Doc                   documentation (sort of)

|----Runtime

|    |----Core
|          |          |----  key component of the simulation
TaskControl
|    |    |----Lib          common libraries of the core components
|    |    |----Traffic      traffic simulation
|    |    |----VtGui        graphical user interface
|          |          |----  symbolic link to applicable image generator (32bit or 64bit)
ImageGenerator
|    |    |----IG64         image generator (64bit version)
|          |          |----  manager for user-furnished module plug-ins (sensor / dynamics)
ModuleManager
|          |          |----  parameter server (for TC only in VTD 2.0)
PrammServer
|          |          |----  editor for traffic scenarios
ScenarioEditor
|    |    |----SimServer    simulation server (base process)
|    |    |----Framework    simulation framework libraries etc.

|    |----AddOns
|    |    |----VILSensor    sensor task for laser sensor etc, VIL only
|          |          |----  cockpit simulation for mock-up
Instruments
|    |    |----Sound.3.2    sound simulation

|    |----Tools
|    |    |----Video        scripts for video conversion etc.
|          |          |----  tool for interfacing the SCP communication (read/write)
ScpGenerator
|          |          |----  converter from .ies files to .rgb files (light maps)
IESConverter
|    |    |----ROD          the famous road designer ;-)
```

`|----Develop`	Development environment for sensors etc.

Data Directory

The data directory is split into three groups:

`<rootDir>`	
`|----Data`	
`|    |----Distros`	the actual config/system data delivered
`|    |----Setups`	all system setups may be found here
`|    |----Projects`	all project data may be found here

Resource and configuration data may be located in any of these directories. Most of the tools implement a search strategy for locating individual data through a two-level search path (major exception: vIG); see also File Finding [39].

NOTE

If you intend to modify any of the configuration files, **do not** modify the ones which are part of the standard distribution; you risk that they'll be overwritten with the next update. Instead, copy them to another location that is of higher priority to the file finder (e.g., your current project or your current setup) and modify only these copies.

If files are referred to by an explicit path from within other configuration files, adapt the respective files' paths.

Data/Setups

A setup contains the actual basic configuration of an installation. It is specific to the underlying hardware, software, and user account structure.

The setup data contains the following elements (only files of interest to or maintainable by the user are listed):

`|----Setups`	
`|    |----Current`	symbolic link to the current setup
`|    |----Standard`	
`|    |    |----Config`	
`|    |    |    |----ImageGenerator`	
`|    |    |    |    |----IGbase.xml`	core IG configuration file (link to actual file)
`|    |    |    |    |----normalSky.xml`	sky model configuration
`|    |    |    |    |----AutoCfgDisplay.xml`	current display configuration (generated by system)
`|    |    |    |    |----AutoCfg.xml`	current overall configuration file (generated by system)

```
|         |         |         |         |----  core IG configuration file
IGbase.Standard.xml
|         |         |         |         |----  reference to database in auto-generated
AutoCfgDatabase.xml                              config files

|    |    |    |----SimServer
|    |    |    |    |----simServer.xml         simulation settings (list of tasks etc.)

|    |    |    |----ScenarioEditor
|         |         |         |         |----  configuration of scenario editor
scenarioEditor.xml.ini.2.0.0
|         |         |         |         |----  links required for the operation of the IG
scenarioEditor.xml.ini

|    |    |----Bin
|    |    |    |----stopTasks                  stop all tasks of the current simulation
|    |    |    |----IGLinks                    links required for the operation of the IG
|    |    |    |    |----Distro
|    |    |    |    |----Setup

|    |    |    |    |----Project

|    |    |    |    |----Vig
|    |----Template                             the template setup, used as a prototype
                                                 for new setups
|    |    |----Config                          see above

|    |    |    |----ImageGenerator

|    |    |    |----SimServer

|    |    |    |----ScenarioEditor

|    |    |----Bin
|    |----Common                               data common to all setups
|    |    |----Scripts                         configuration scripts etc.
```

Data/Projects

Project data defines the actual content of a simulation. It is therefore located in a dedicated directory tree. Due to the implemented search strategy for configuration and data files, all data not contained in the current project directory may be retrieved from other directories (usually below the `Setup` or `Distro` directories).

The project directory provides the following hierarchy. The actual version numbers of the respective components may be different from the ones shown here.

```
|----Data

|    |----Projects
|    |    |----Current                        symbolic link to the current project
|    |    |----SampleProject                  the sample project (current upon delivery)
|    |    |    |----Videos                    location for generated video files
```

| | | |----Databases	project-specific databases or symbolic link to common databases
| | | | |----Town	
| | | | |----SmartDB	
| | | | |----ObjectsCommon	
| | | | |----Cars	
| | | |----ImageGenerator	additional data for the image generator
| | | | |----LightSources	light source textures
| | | | |----Symbols	symbol textures
| | | |----Config	
| | | | |----Players	individual player configuration data (supersedes the Distro files)
| | | | |----ImageGenerator	
| | | | | |----SymbolsStd.xml	symbol configuration for IG
| | | | | |----LightSrcStd.xml	light source configuration for IG
| | | | |----ModuleManager	
| | | | | |----moduleManager.xml	module manager configuration
| | | |----SampleProject.	GUI's project configuration file
| | | |----Plugins	
| | | | |----ModuleManager	additional plug-ins for the module manager
| | | |----Scripts	
| | | | |----ScpGenerator	test scripts for the SCP generator
| | | |----Scenarios	the project's scenarios (here: test scenarios)
| | | | |----TownPathLong.xml	
| | | | |----carTypes.xml	
| | | |----Recordings	location for generated data recordings
| | | | |----Format6.3	
| | | |----Sounds	
| | |----Default	default project used as a template for new projects

```
|    |    |    |----Videos
|    |    |    |----Template.vpj
|    |    |    |----Databases
|    |    |    |    |----Town
|    |    |    |    |----SmartDB
|    |    |    |    |----Audi
|    |    |    |    |----ObjectsCommon
|    |    |    |    |----Cars
|    |    |    |----ImageGenerator
|    |    |    |    |----LightSources
|    |    |    |    |----Symbols
|    |    |    |----Config
|    |    |    |    |----Players
|    |    |    |    |----ImageGenerator
|    |    |    |    |----ModuleManager |
|    |    |----Plugins
|    |    |    |    |----ModuleManager
|    |    |    |----Scripts
|    |    |    |----Scenarios
|    |    |    |----Recordings
|    |    |    |----Sounds
```

Data/Distros

A full data tree will be delivered with each distribution, which contains the basic setup and configuration files. These may be superseded by files defined in the current setup and project (provided that the search path is configured correctly).

The distribution provides the following hierarchy. The actual version numbers of the respective components may be different from the ones shown here.

```
|----Data
|    |----Distros                    collection of available distributions
|    |    |----Current               link to current (active) distribution
|    |    |----Distro                distribution of VTD
```

				----Databases	visual and logical databases
				|----Cars	
				|----Crossing8Course	
				|----ObjectsCommon	
				|----SmartDB	
				|----SmartDB.2014	
				|----Town	
				----ImageGenerator	basic image generator file
				|----Fonts	
				|----Misc	
				|----Symbols	
				|----LightSources	
				----Config	
				|----Players	player configuration files (vehicles, drivers, characters)
					|----Vehicles	
					|----Objects	
					|----driverCfg.xml	
					|----characterCfg.xml	
				|----ImageGenerator	IG configuration files
					|----IGConfig.xml	
					|----SymbolsStd.xml	
					|----LightSrcStd.xml	
					|----Materials.xml	
				|----ModuleManager	module manager (fall-back) configuration
					|----moduleManager.xml	
				----Plugins	
				|----ModuleManager	module manager plug-ins

```
|        |        |        |        |----
libModulePerfectSensor.so

|        |        |        |        |----
libModulePerfectSensor.so.4

|        |        |        |        |----
libModulePerfectSensor.so.4.0

|        |        |        |        |----
libModulePerfectSensor.so.4.0.0
|        |        |        |        |---- perfect sensor plug-in
libModulePerfectSensor.so.4.0.0
|        |        |        |        |----
libModuleTrafficDyn.so

|        |        |        |        |----
libModuleTrafficDyn.so.4

|        |        |        |        |----
libModuleTrafficDyn.so.4.0
|        |        |        |        |---- dynamics plug-in
libModuleTrafficDyn.so.4.0.0
```

Runtime Directories

These directories contain the actual run-time environment of the toolchain. The following tree provides an overview of the most important files contained in the Runtime directory and its sub-directories. The actual version numbers of the respective components may be different from the ones shown here.

```
|----Runtime
|    |----Core
|    |    |----TaskControl
|    |    |    |----taskControl.4.0.1          the master of disaster
|    |    |    |----taskControl                link to current version of taskControl
|    |    |----Lib                             collection of (3rd party) libraries shared
                                                by several components
|    |    |----Trffic
|    |    |    |----ghostdriver.2.0.0
|    |    |    |----ghostdriver                link to the current version of the traffic
                                                simulation
|    |    |----VtGui

|    |    |    |----Vtgui.2.0.0
|    |    |    |----Vtgui                      link to the current version of the GUI
|    |    |----ImageGenerator                  link to an applicable image generator
```

```
|    |    |----IG64                                  64bit image generator
|    |    |    |----bin
|    |    |    |----data
|    |    |    |----doc
|    |    |----ModuleManager
|    |    |    |----moduleManager                    link to the latest module manager
|    |    |    |---- moduleManager.4.0.0
|    |    |    |----lib                              system libraries for module manager
|         |         |         |         |----
libVTDModulePlugin.so
|         |         |         |         |----
libVTDModulePlugin.so.4
|         |         |         |         |----
libVTDModulePlugin.so.4.0
|         |         |         |         |----
libVTDModulePlugin.so.4.0.0
|    |    |----ScenarioEditor
|    |    |    |----scenarioEditor
|    |    |    |----startCharacterViewer
|    |    |    |----scenarioEditor.2.0.0
|    |    |----Framework
|    |    |    |----lib
|    |    |    |    |----libVTDFramework.so
|         |         |         |         |----
libVTDModulePlugin.so.4
|         |         |         |         |----
libVTDModulePlugin.so.4.0
|         |         |         |         |----
libVTDModulePlugin.so.4.0.0
|    |----AddOns
|    |    |----Sound.3.2
|    |    |    |----vroom                            latest sound module with traffic sounds
|    |----Tools
|    |    |----Video                                 scripts for video conversion etc
|    |    |----ScpGenerator                          tool for interfacing the SCP
                                                      communication (read/write)
|    |    |    |----scpGenerator.4.0.0
|    |    |    |----scpGenerator                     link to current SCP generator
|    |    |----RDBSniffer                            tool for interfacing the RDB
                                                      communication (read/write)
|    |    |    |----rdbSniffer.4.0.0
```

```
|    |    |    |----scpGenerator                link to current RDB sniffer
|    |    |----Scripts
|    |    |----ROD                              the road designer
```

Develop Directory

The Develop directory contains the components for the development of their sensors, dynamics plug-ins, etc. Its hierarchy is as follows:

```
|----Develop
|    |----Modules                                development environment for module plug-ins
|    |    |----makefile
|    |    |----Common
|    |    |    |----inc                          interface files for the plugins and data structures
|    |    |    |    |----ModuleIface.hh
|    |    |    |    |----ModulePlugin.hh         interface files for a module plugins (base class)
|    |    |    |    |----DynamicsIface.hh
|    |    |    |    |----DynamicsPlugin.hh       interface files for a dynamics plugin (derived class)
|    |    |    |    |----SensorIface.hh
|    |    |    |    |----SensorPlugin.hh         interface files for a sensor plugin (derived class)
|    |    |    |----lib
|    |    |    |    |----libVTDModulePlugin.so
|    |    |    |    |----libVTDModulePlugin.so.4
|    |    |    |    |----libVTDModulePlugin.so.4.0
|    |    |    |    |----libVTDModulePlugin.so.4.0.0
|    |    |----DummyDriver                       example of a driver implementation as a plug-in
|    |    |----ImperfectSensor                   example of a sensor implementation as plug-in
|    |    |----SampleDyn                         sample dynamics plug-in
|    |    |    |----src
|    |    |    |    |----SampleDyn.cc
|    |    |    |----SampleDyn.pro
|    |    |    |----inc
|    |    |    |    |----SampleDyn.hh
```

|	|	|----PerfectSensor	sample sensor plug-in
|	|	| |----src	
|	|	| | |----PerfectSensor.cc	
|	|	|----PerfectSensor.pro	
|	|	|----inc	
|	|	| |----PerfectSensor.hh	
|	|----inc		
|	|	|----CommonQmakeDefs.pro	
|	|----Framework		framework interface files
|	|	|----inc	
|	|	| |----Plugin.hh	
|	|	| |----scpIcd.h	SCP interface files
|	|	| |----Iface.hh	
|	|	| |----Coord.hh	
|	|	| |----viRDBIcd.h	RDB interface files
|	|	|----RDBHandler	
|	|	| |----example.cc	example for reading RDB buffers of IG's SHM
|	|	| |----RDBHandler.cc	
|	|	| |----RDBHandler.hh	
|	|----Vig		SDK for ImageGenerator (optional component)
|	|	|----3rdParty	3rd party libraries
|	|	| |----Cuda	
|	|	| |----Optix	
|	|	|----bin	vIG binaries
|	|	|----Framework	vIG development framework
|	|	|----Plugins	vIG sample plugins

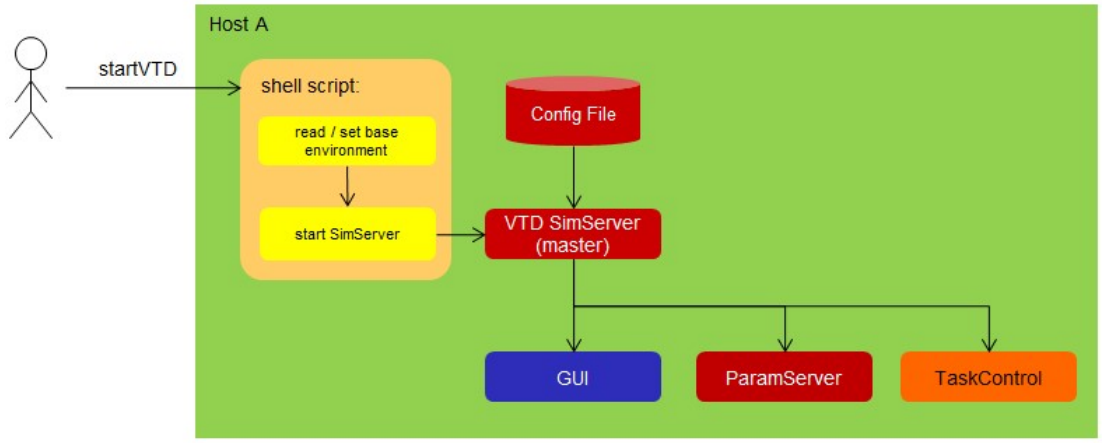
Component Management

Here we shall discuss the individual components of VTD and their use within the simulation. We shall discuss the following components:

- SimServer
- VT-GUI: the UI of the Virtual Test Drive Suite
- ScenarioEditor
- Image Generator (IG)
- TaskControl (TC)
- Traffic Simulation
- Dynamics and Driver Model of the Ego Vehicle
- Sound Simulation
- ModuleManager
- SCP-Generator
- RDB-Sniffer
- Databases
- Road Designer (ROD)

SimServer

Starting with VTD 2.0, all components are managed (i.e., started and stopped) by a ***SimServer*** (simulation server). This is applicable for both single and multi-host operations.



One instance of the SimServer will be started on each host that is involved in the simulation. The SimServer on the local host will act as the master server; the other instances will act as its clients. They will be started via ssh using parameters in the SimServer's configuration file (see next chapter).

The shell script started by you is:

```
bin/vtdStart.sh
```

The shell script can be started with an optional argument:

```
bin/vtdStart.sh -select
```

If the argument is specified, the script will prompt you to select the setup that will be set as the current setup.

The local SimServer process will read the file:

`Data/Setups/Current/Config/SimServer/simServer.xml`

It contains entries for environment settings, search paths, and instructions for starting individual components on each host. The total number of hosts, their names, etc., will be determined in this file.

You may communicate with the SimServer using the SCP commands. The default port for this communication, however, is 32512. The relevant commands for communication with the SimServer will be explained below.

The basic settings (communication ports) are defined at the beginning of the file:

```
<SimServer name="Standard" comPort="32512" notifyLvl="INFO">
```

Do not modify any of these settings unless otherwise instructed!

The next section of the file describes the environment variables that are set for a given setup. Most of these variables should not be modified by the user unless otherwise instructed. You may add your own environment variables at your own discretion:

```
<!-- Task settings -->
<!-- VTD Runtime -->
<EnvVar name="VI_RUNTIME_DIR"       val="$VTD_ROOT/Runtime" />
<EnvVar name="VI_CORE_DIR"          val="$VI_RUNTIME_DIR/Core" />
<EnvVar name="VI_TOOLS_DIR"         val="$VI_RUNTIME_DIR/Tools" />
<EnvVar name="VI_ADDON_DIR"         val="$VI_RUNTIME_DIR/Core" />

<!-- VTD Data -->
<EnvVar name="VI_DATA_DIR"          val="$VTD_ROOT/Data" />
<EnvVar name="VI_SETUP_DIR"         val="$VI_DATA_DIR/Setups" />
<EnvVar name="VI_PROJECT_DIR"       val="$VI_DATA_DIR/Projects" />
<EnvVar name="VI_DISTRO_DIR"        val="$VI_DATA_DIR/Distros" />
<EnvVar name="VI_CURRENT_SETUP"     val="$VI_SETUP_DIR/Current" />
<EnvVar name="VI_CURRENT_PROJECT"   val="$VI_PROJECT_DIR/Current" />
<EnvVar name="VI_CURRENT_DISTRO"    val="$VI_DISTRO_DIR/Current" />


<EnvVar name="PATH"                 val=".:$PATH" />
<EnvVar name="LD_LIBRARY_PATH"      val="$LD_LIBRARY_PATH:$VI_CORE_DIR/Lib"                 />
<EnvVar name="LD_LIBRARY_PATH"      val="$LD_LIBRARY_PATH:$VI_CORE_DIR/Traffic"             />
<EnvVar name="LD_LIBRARY_PATH"      val="$LD_LIBRARY_PATH:$VI_CORE_DIR/ImageGenerator/bin"  />
<EnvVar name="LD_LIBRARY_PATH"      val="$LD_LIBRARY_PATH:$VI_CORE_DIR/Framework/lib"       />
<EnvVar name="LD_LIBRARY_PATH"      val="$LD_LIBRARY_PATH:$VI_CORE_DIR/SensorManager/lib"   />
```

For each host, you may specify processes that shall run on it. These process specifications have to be defined under a common <Host> tag. There are two possibilities:

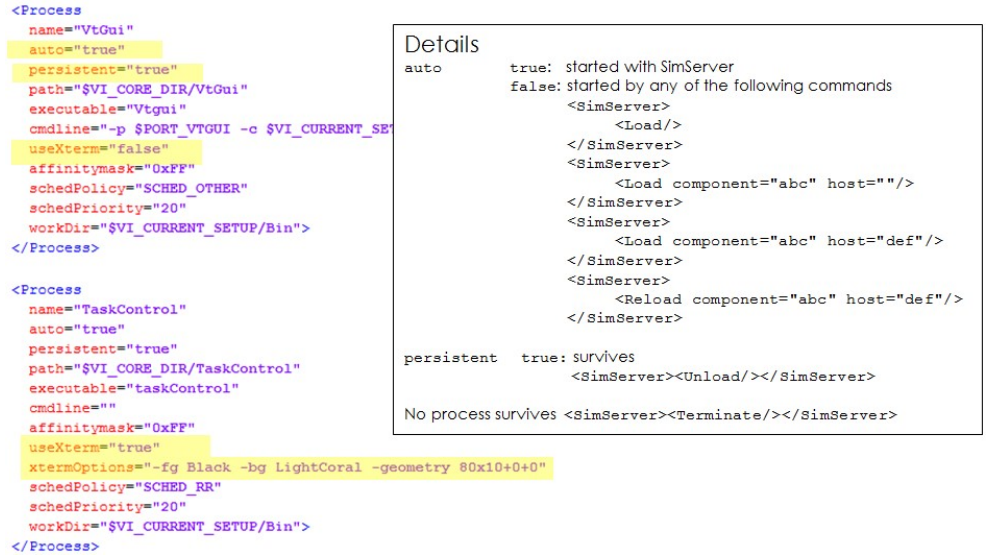
- Use remote host `<Host name="sepp" vtdRoot="/home/vtdlocal/VTD.2.0" username="vtdlocal">`
- Use the localhost `<Host>`

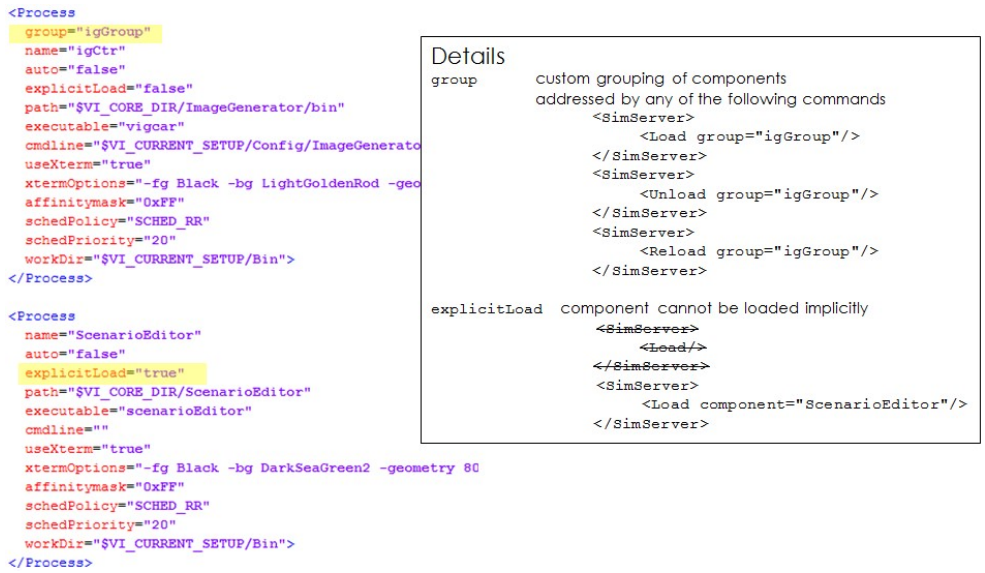
NOTE

For the remote host, you have to specify the following:

- its name
- the absolute path to the VTD directory
- the name of the user with which to establish the ssh connection

When and whether a process starts depends on the individual settings. These are illustrated in the following two images:





Adding Custom Components

You may add custom components to the overall component management. To add your own component, perform the following steps:

1. Install your component in the preferred location.
2. Open the file `Data/Setups/Current/Config/SimServer/simServer.xml`.
   This file already contains the start commands for all other components within a setup.
3. Look for an entry of a component that is similar to the one you want to add, copy this one under the <Host> entry where you wish to run the component and adjust the parameters accordingly.
4. Click **Apply**. Now your component will start.

VT-GUI

VT-GUI is the Graphical User Interface (GUI) of the Virtual Test Drive Suite. It cannot be used parallel to other control software operating the SCP interface (e.g., ScpGenerator).

Due to some functional restrictions in various versions of VTD you may experience that some panels - even if described in the following chapters - are not or not completely operable in your version of the software. The respective panels will be switched to an operable state in upcoming releases of VTD.

NOTE

The layout and operation of the GUI changed slightly with VTD 2.0. The following snapshots are not yet up to date but will be adapted until the next version of this manual. However, you may still operate VTD following the instructions below with two exceptions:

Components → Load components does no longer exist (use APPLY instead, see Resources [9])

Tools → TaskControl no longer exist (use the CONFIGURE command a panel instead)

GUI Layout

The GUI consists of the main window containing several tasks-oriented sub widgets. Sub widgets are either docked to the main window, or they pop up as separate windows. They can be resized by handles or hidden if not needed. The size of the GUI is minimized to allow its usage also on low-resolution screens.

The essential information and operation areas are shown in the following figure:

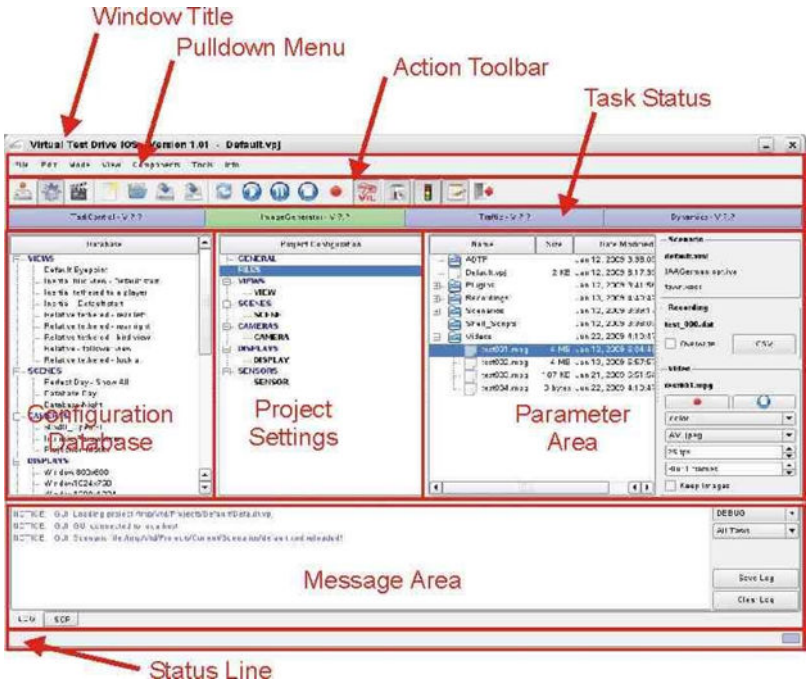


Figure 7. VT-GUI function and display areas

- **Window title** shows the GUI version and the names of the active setup and the loaded project.
- **Parameter area** allows the modification of the selected configuration items either from the database or the project view.
- **Message area** contains a tab widget showing either the messages from the GUI or VTD tasks or a LOG of important SCP messages.
- **Status line** shows hints, information, and a summary of the task status.

The other components are described as follows.

Pulldown Menu

The menus allow the selection of actions and views like

- creation, loading, and saving of projects
- editing of preferences
- selection of the operation mode (Preparation / Operation / Replay)
- viewing of widgets and windows (Task State / Messages / Configuration Database / VIL Sensor Info)
- Start and Stop of the VTD components
- Start of VTD tools (ROD / Scenarioeditor)
- Start of the information window

Action Toolbar

The toolbar elements allow the following operation:

- Selection of the simulation mode (Preparation / Operation / Replay)
- Creation, loading, and saving of projects
- Hide/Show of additional widgets and windows (VIL / Database / Task State / Messages, Parameter Graph)
- VCR operation of the replay
- Selection of replay speed and single-step width

- GUI exit

The tool buttons may be hidden or inactive due to simulation mode and other dependencies. Tooltips and status bar messages ease the operation.

Task Status

This area shows the status of VTD tasks. Each task sending its status via the TaskControl will create a color-coded button in this area.

- RUNNING green
- READY blue
- PENDING yellow
- ERROR red
- UNKNOWN magenta
- TIMEOUT grey

If there is no connection to the TaskControl, the area remains empty. If a task is sending no status update for 3 seconds, the status will change to UNKNOWN, and after 10 seconds, it will be removed from the list. The timing can be adapted in the preference settings.

The status bar summarizes all task states as a small color-coded rectangle with the color of the worst status from all tasks. The TaskControl has no specific button because it is always in RUN mode or unavailable. If a task goes to an error state, this widget will popup automatically.

Configuration Database

The main window can be extended via the **View** menu or the **Show Database** tool button with an additional widget showing preconfigured settings for

- VIEWS
- SCENES
- CAMERAS
- DISPLAYS
- SENSORS
- LIGHTSOURCES

This information is stored in a file called database.xml, which is loaded and saved automatically during start-up and shutdown of the GUI. The settings are shown in a tree view that can be expanded by double clicking the title line of the widget. The tree view items provide context menus for adding and deletion, renaming, and transferring an item from the database to the project. The selection of an item brings up the corresponding controls in the parameter area. Double click to copy the settings to the project.

Project Settings

The project tree view contains the selected project settings and will be operated analog to the Configuration Database. In addition to the described groups, it shows an entry for the project description and the project-related files. The selection of an item is done by a double click and makes that item active. A bold green font type identifies the selected item. While being in RUN mode, selection and modification may be restricted. Modifications on the project settings must be saved explicitly.

Project Files

The users always work with data in a single PROJECT. It is located in a dedicated directory (e.g. `VTD.x.y/Data/Projects/_nameOfProject_`) and contains a GUI configuration file with the name of the project plus the `extension .vpj` (`e.g., _nameOfProject_.vpj`).

Projects can be created by selecting new from the file menu and giving the project a unique name. This process copies basic data from a template. Another possibility is the **Save As** function for already configured projects.

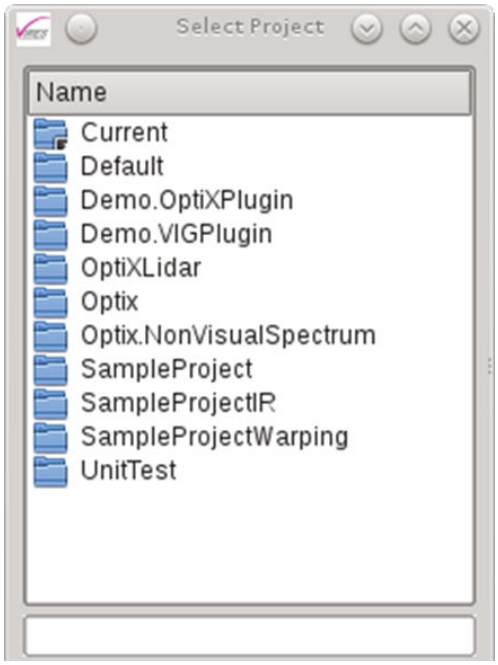


Figure 8. Project Dialog

Operating Modes

As written in Runtime Phase [35], VT-GUI supports the simulation's three operating modes, which can be selected from the Mode-menu or via the toolbar. A mode change stops a running simulation or replay. If the simulation is not running or correctly configured, controls (e.g., START/STOP) may be unavailable.

- **Preparation Mode**: It offers a simple path following algorithm along the path of the first external player. With the preparation controls' help, the user can set speed and change lanes along the path. The animation stops at the end of the path.

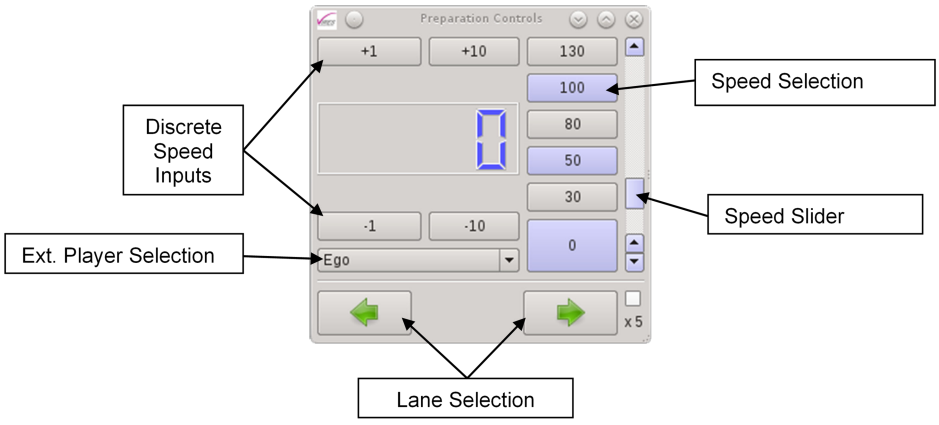


Figure 9. Preparation Controls

- **Operation mode**: In this mode the animation is done via the configured driving dynamics model.
- **Replay mode:** It allows the replay of recorded simulations. The replay speed can be adapted in the toolbar, and parameters can be displayed in the graph window.

Toolbar Functionality

The toolbar buttons will be adapted to the simulation mode and will offer actions for

- Mode Switching
- File Operation
- Simulation or Replay Control
- Widget Controls

Depending on the simulation state or configuration, some actions may be set inactive.

Figure 10. Toolbar Preparation/Operation Mode

Simulation mode switching is done with the mode select buttons to the left. When selecting the preparation mode, the speed control window opens automatically.

- The **File** functions allow the creation, loading, and saving of projects.
- The **simulation** control buttons initialize, start, pause, or stop the simulation.
- The **record** button triggers data recording after the next start. If the IG is configured for live video recording, the corresponding button turns this feature on.

  The widget controls open additional widgets or windows, and the exit button shuts down the program

Figure 11. Toolbar Replay Mode

The replay mode exchanges some buttons with the VCR style replay controls and the GRAPH window button. It allows setting the single step size and the replay speed. A LOOP tool button selects whether a single replay file is repeated or all replays in one directory are looped.

Project Configuration

A project is defined by a set of parameters organized in a tree for easy access and modification. It needs a description, the definition of necessary external files like scenario, record file, etc., and at least one active item out of the following groups to allow the start of a simulation.

- VIEW
- SCENE
- CAMERA
- DISPLAY

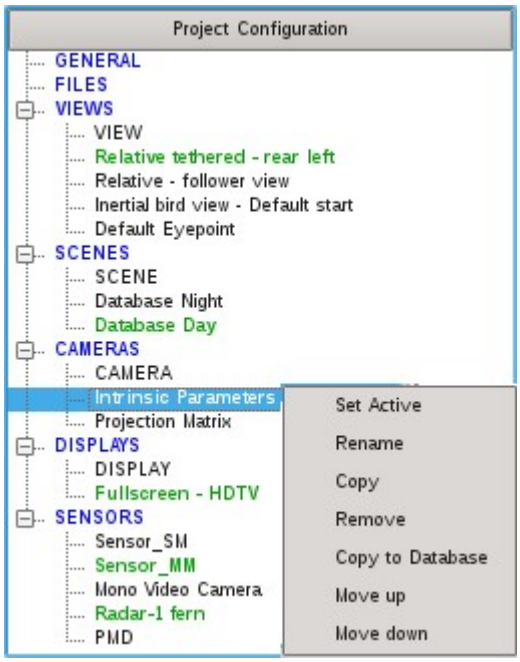


Figure 12. Project Function groups with context menu

Each group can have several subitems, but only one can be active (except the SENSOR group can have more than one active sensor). By selecting one item (single left click), the parameters are shown in the parameter widget and modified. A double click (de-)activates an item, and a right-mouse click brings up a context menu. Active items are displayed in bold green font style. For certain conditions modification or selection of parameters is forbidden (e.g., the display is not changeable during running simulation).

General

The **General** parameter widget contains a project description saved with the project file.

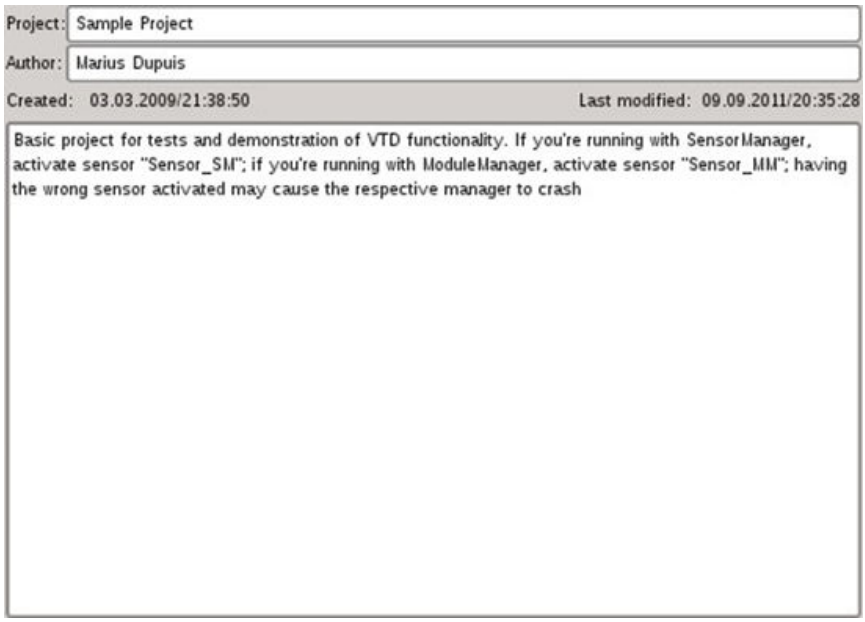


Figure 13. The General Project Parameters

Files

The file parameter widget shows a view of the project directory and allows selecting scenarios, record, and video files with a single left-click.

Figure 14. The FILES Project Parameters

Double click actions are implemented for:

config files	opens the file in an editor
scenario files	select and start the scenario
SCP files	excecutes the SCP scripts with the scpGenerator
sound files	play the sound file
symbol files	start this file in an image viewer
recording files	select and start the recording
video files	select and start this file in a movie player

Context menus allow additional functionality like edit, delete, and rename. Files must be placed in the corresponding directory for this function to work properly. The scenario group box shows the selected scenario and the used OpenDrive and database file. Tooltips show the complete pathname.

To enable a recording, a record file must be selected. If no file is available, a new filename must be created via the context menu in the directory tree. Unless the **overwrite** checkbox isn't tagged, a number is added to the filename and incremented for each simulation start. The CSV Export button creates a comma-separated ASCII file of the recording.

Video generation requires a record file that is used to generate images for each frame. After this step, the images are combined into a video. As this is a time-consuming process, directories should be located on the local machine. Controls are available to select the image buffer (color/depth), video parameters, and a checkbox to maintain the generated images.

CSV and video generation are separate triggered processes and do not run in parallel to a simulation.

Views

This widget allows to set position and viewing direction of the out of the window view.

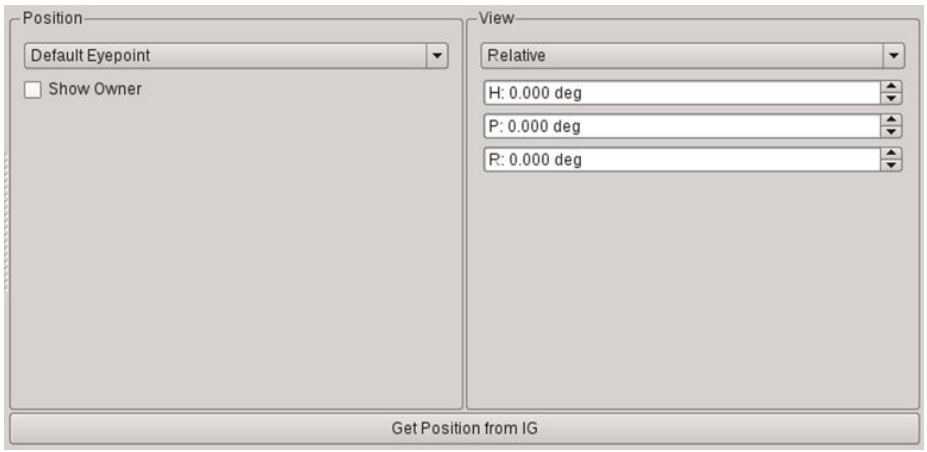


Figure 15. Project View Tab

For the positioning of the eyepoint, the following options are available:

Default Eyepoint	Eyepoint of the ,,EGO" (external player)
Eyepoint	Eyepoint of the player selected in the combo box
Inertial	inertial XYZ Coordinate
Relative	XYZ relative to the pivot of another player
Tethered	polar coordinate (DAE) relative to another player (car system)
Tethered Inertial	polar coordinate (DAE) relative to another player (inertial system)
Sensor	eyepoint positioned at the sensor position

The viewing direction can be

- Inertial — HPR (heading, pitch, roll)- inertial coordinate system
- Relative — HPR (heading, pitch, roll)- car coordinate system
- Look at Position — look at an inertial coordinate
- Look at Player — look at a player position

By pressing the “Get Position from IG” button, the current IG eyepoint coordinate is retrieved. Using the IG's built-in free movement modes, it is possible to easily search a coordinate in 3D and retrieve it from the GUI. When loading different scenarios, the players may change, and therefore the view settings have to be modified.

Scenes

The scenes widget allows the selection of the visual environment.

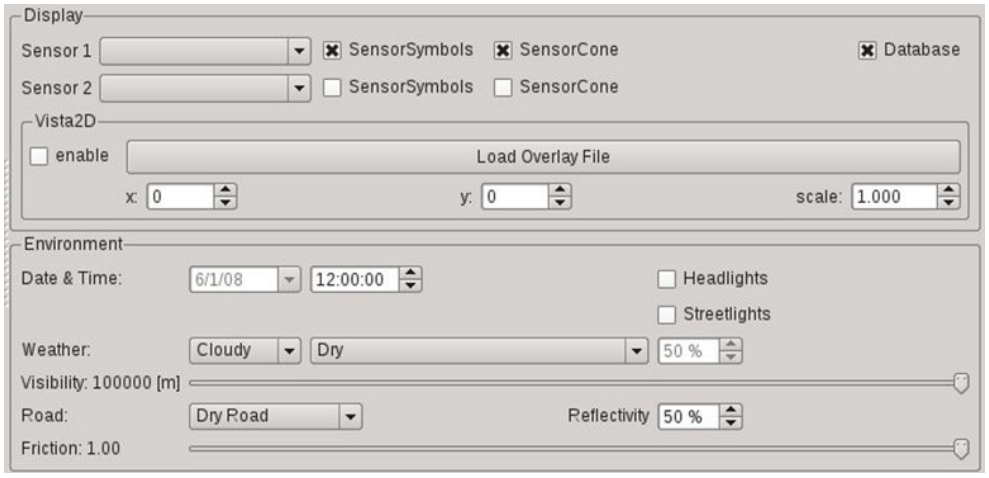


Figure 16. Project „Scene Tab“

Following options are available:

- Sensor — selection of 2 different sensors (must be enabled in the sensor widget)
- Sensor Symbols — display the sensor symbols
- Sensor Cones — display of the sensor cone
- Database — display of the database

A Vista2D overlay can be specified for the IG

- enable — enable display
- Overlay file — the Vista2D symbology overlay file
- x/y coordinates — overlay screen coordinates
- scale size — scale of the overlay

The environment group allows the control of following parameters:

- simulation date - not yet enabled
- simulation time
- car headlight s
- streetlights (only on supported databases)
- sky occlusion (no sky, sunny, cloudy, overcast, rainy)
- precipitation (dry, rain, snow)
- precipitation rate (0% - 100%)
- visual range (fog)
- road condition (dry, wet)
- road reflectivity factor
- road friction value overwrite

Cameras

The camera widget allows the settings for the projection matrix by three different methods:

- **OpenGL Frustum**: The frustum is defined by horizontal and vertical viewing angles and the near and far clipping planes. An offset can be given for asymmetric viewing frustum, which defines the frustum's middle axis's offset angle.

Figure 17. The Camera Tab - OpenGL Frustrum Project

- **Intrinsic Parameters**: The viewing matrix can also be defined by specifying the intrinsic camera parameters.

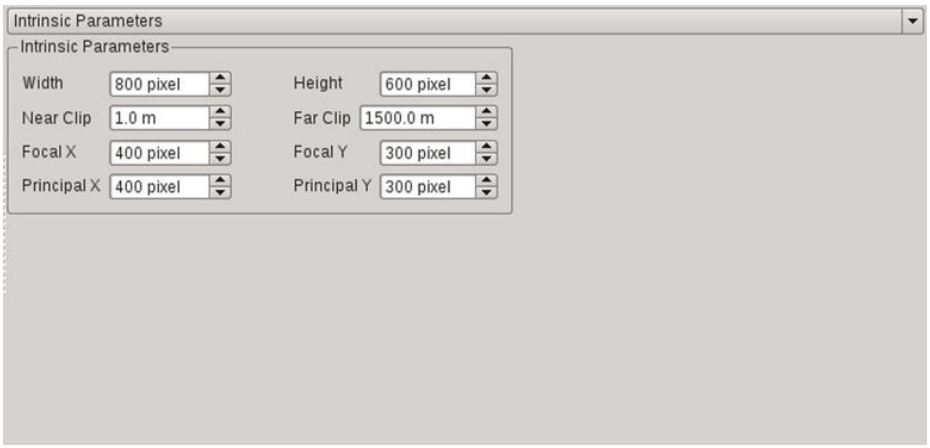


Figure 18. The Camera Tab – Intrinsic Parameters Project

- **Projection Matrix**: As a third option, the camera matrix can be given directly. Use with caution since wrong settings can cause the image generator to evaporate in the Nirwana of NaNs.

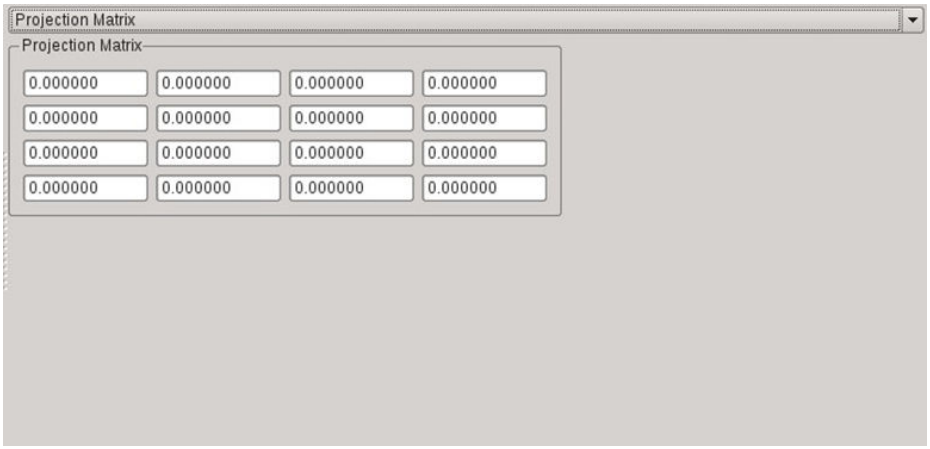


Figure 19. the Camera Tab – Projection Matrix Project

Displays

This widget allows defining the rendering window.

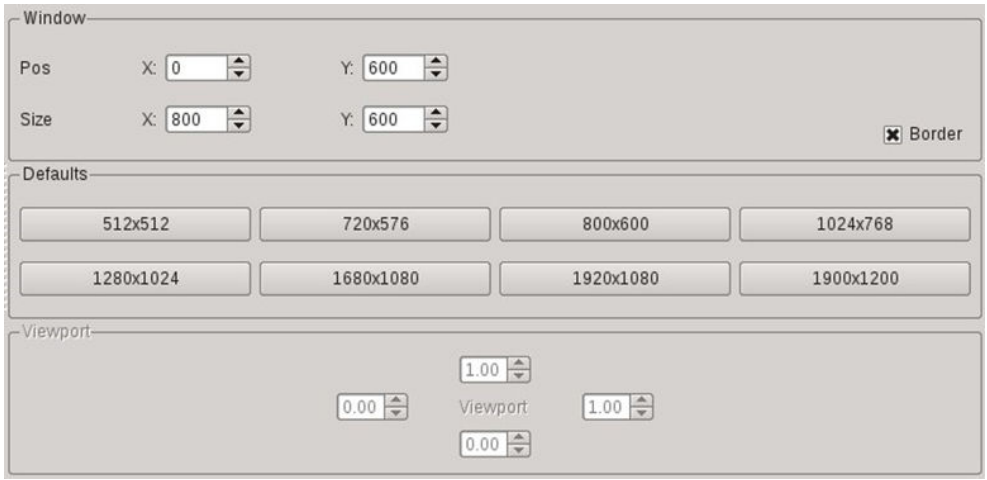


Figure 20. Display Tab Project

Following parameters can be defined:

- pos window position relative to the top left corner of the screen [ixel]
- Size size of the window
- Border window border ON/OFF
- Defaults predefined window settings for quick selection
- Viewport viewport area inside the window [0..1] - not yet enabled

Sensors

This widget allows the control of the sensor parameters.

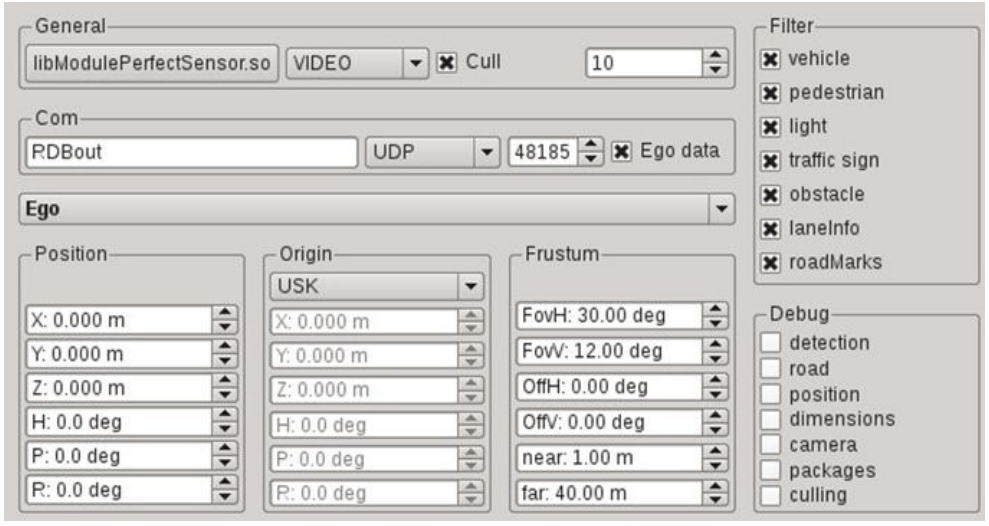


Figure 21. Sensor Tab Project

Following controls are available:

- General selection of the sensor plug-in, type, frustum cull, num. of objects
- Filter selection of objects to be detected by the sensor
- Com sensor communication settings
- Sensor owner player carrying the sensor
- Position sensor position and altitude (car coordinates) - not enabled
- Origin sensor origin - not enabled
- Frustum sensor frustum - not enabled
- Debug debug flags for the sensor/module manager output

Message Widget

This widget records messages and commands from and to simulation tasks. Notifications and messages from the GUI will be shown in the LOG tab. Filter functions allow sorting for priority or sender.

The SCP tab shows the messages from the SCP port. Filter functions allow sorting for type or sender. Some cyclic messages (e.g., simulation task state) are suppressed. A command-line allows to transmit SCP messages from a command line. The logs can be cleared or exported to a text file. The SCP log update can be disabled in case of heavy SCP traffic. Although the number of displayed lines is limited, the full log will be written.

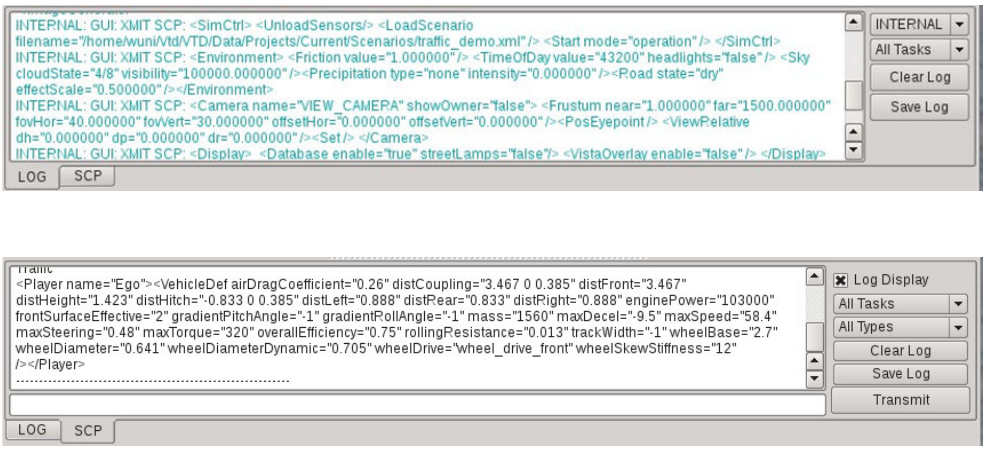


Figure 22. Message and SCP Widget

Preferences Window

You can access the preference window via the **Edit** pulldown menu. It shows the files tab, which allows selecting default directories and tools for editing and display. The settings are stored in the file database.xml (see Configuration Database [56]).

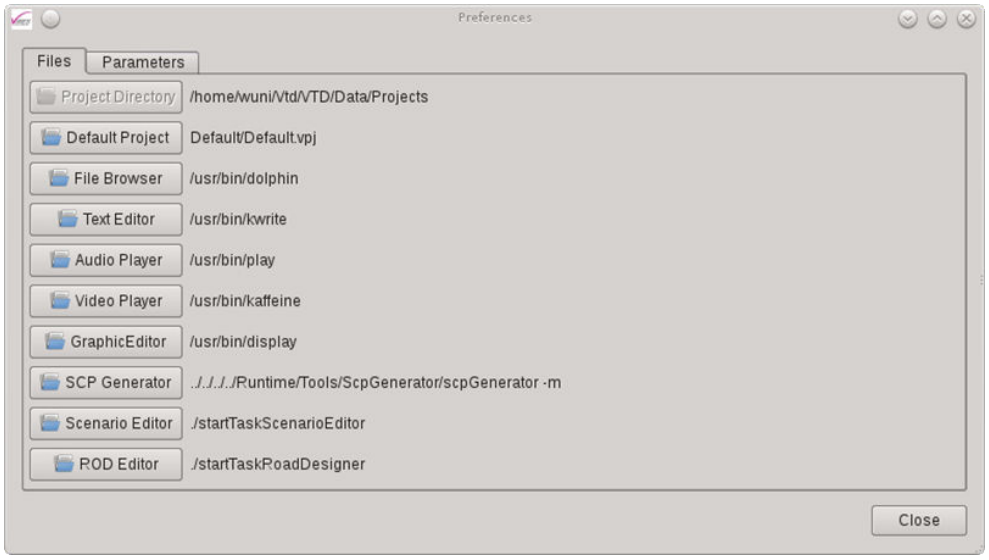


Figure 23. Preferences Window – Path and executable settings

The **Parameters** tab allows the selection of the GUI font, the settings for the displayed lines in the message widget, the modification of the timing for the task status indication, the notification settings for the shell output, and the timeout before unloading the components.

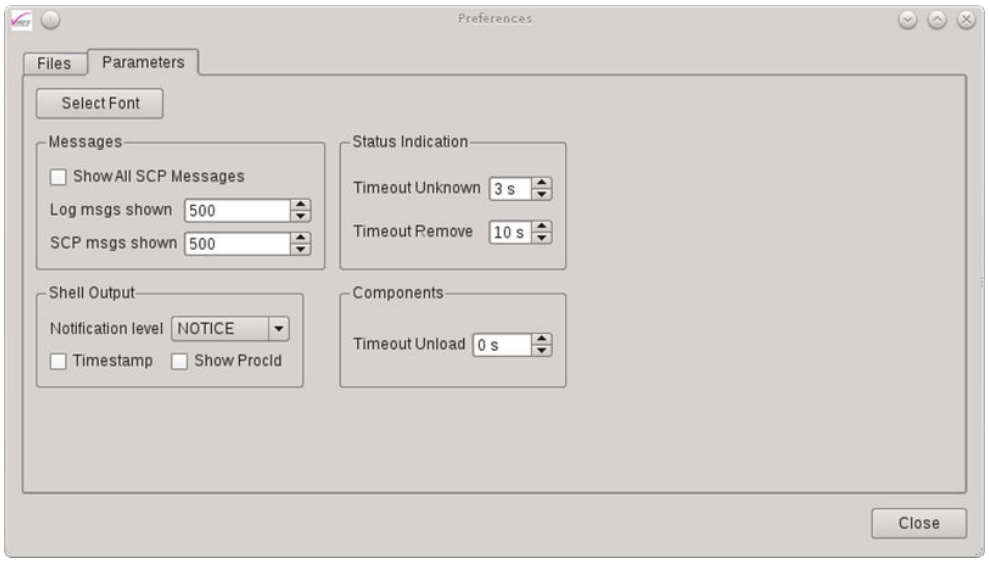


Figure 24. Preferences Window – Parameters

Configuration Database

GUI settings will be loaded from and stored into the configuration database (see VT-GUI (IOS) [138]). Project-specific references (e.g., relation to a player) will be lost. Transfer of the settings is initiated via the context menu (copy to/from database). The modification of parameters described in the previous chapters also works for database entries. The database is saved automatically into the file `database.xml` (see Figure File Dependencies of the IOS [67]) in the distribution.

Files and File Formats

Figure 25 shows the file dependencies of the VtGUI.

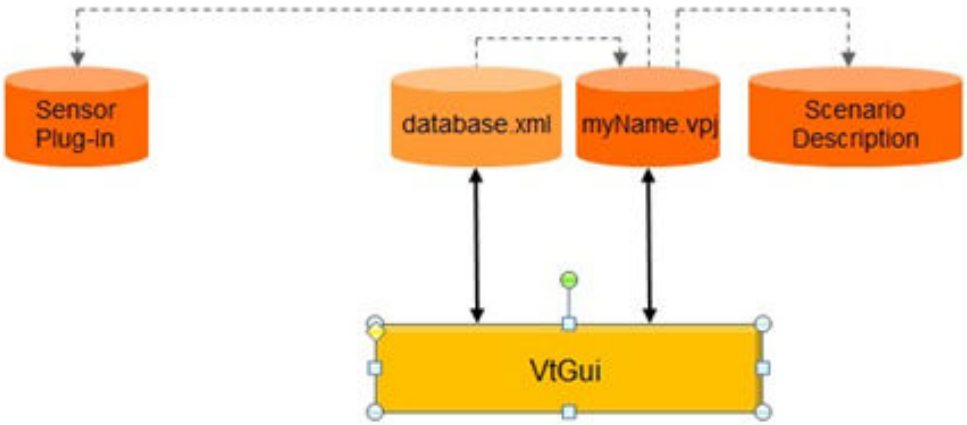


Figure 25. File Dependencies of the IOS

The IOS manages the following files:

Configuration database:	`database.xml`
Project:	`<NameProject>.vpj`
Default Project:	`Data/Projects/Default`

The file formats of the configuration and project file are described in VT-GUI (IOS) [138].

Communication

The IOS communicates with the simulation via the SCP channel. It is connected to the channel as a client with the TaskControl acting as a server (see Figure Communication with IOS (GUI) [68]).

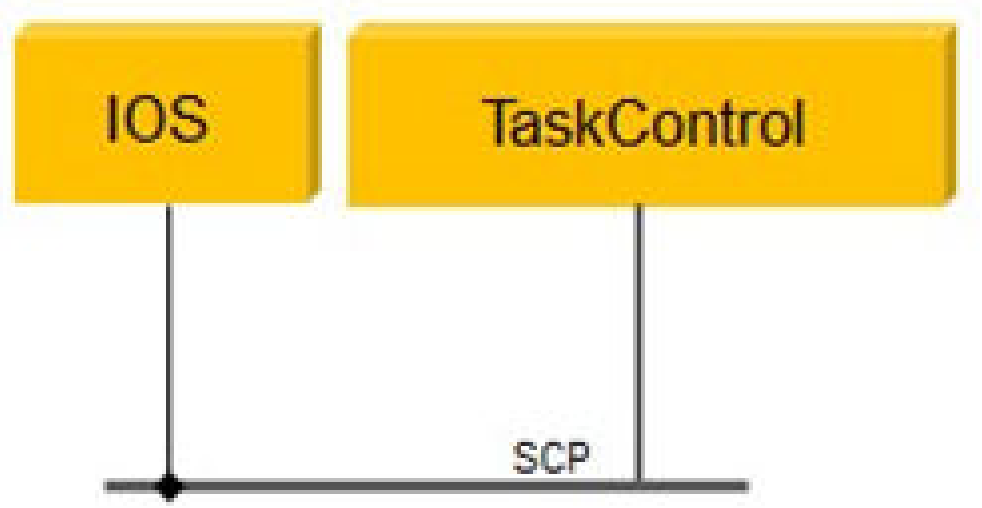


Figure 26. Communication with IOS (GUI)

Installation / Options

The GUI is installed at `Runtime/Core/VtGui`

Executable file:		`Vtgui`	
Command parameters:	line	`-p <portNo>`	Number of the SCP port
		`-s <server>`	name of the host running the TaskControl

Start Procedure

The start parameters of the GUI are configured in the file

`Data/Setups/Current/Config/SimServer/simServer.xml`

ScenarioEditor

Operation and configuration of the ScenarioEditor are described in detail in [3], which is located in the `Doc/` directory. In the following chapters, only the details relevant to the VTD context will be described.

Files and File Formats

The scenario editor is linked to the simulation's configuration via the files and dependencies shown in Figure File Dependencies of the ScenarioEditor [69].

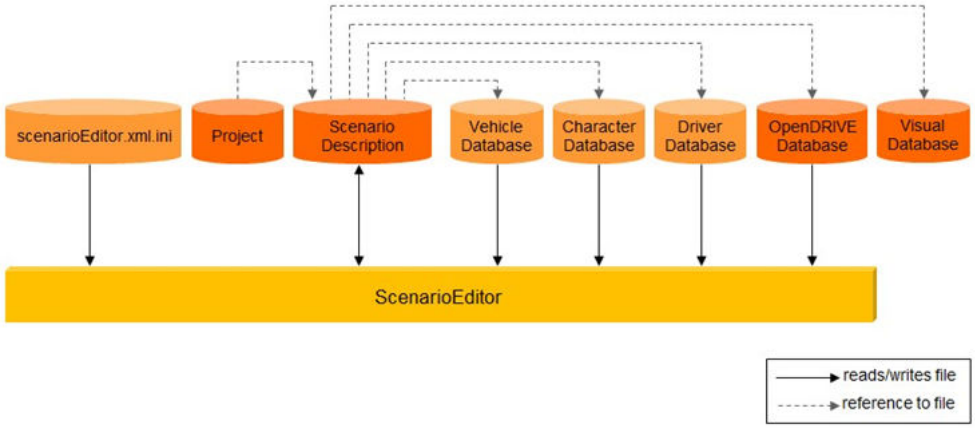


Figure 27. File Dependencies of the ScenarioEditor

The complete content of a scenario is saved in the "Scenario Description", an XML file. This file contains references to other files containing the logical description of the road network, settings for vehicles, drivers, pedestrians, and the visual database. The XML scheme of the scenario file is located in VTD's Doc directory.

Starting with VTD 2.1, the file format `OpenSCENARIO` is used for storing scenario data. This functionality is still in beta state and must not yet be used without the support team's prior consultation.

Communication

The ScenarioEditor may be used as a standalone application, e.g., for the preparation of a scenario. During a simulation, the ScenarioEditor may be switched to the on-line mode and, thus, serves as a monitor tool. It is only connected to the TaskControl. The parameters of the connection (port numbers and types) may be defined in the ScenarioEditor's configuration file. For the TaskControl, the environment variables are given in Environment Variables [129].

Figure Connecting the ScenarioEditor [70] shows the connections to the ScenarioEditor:

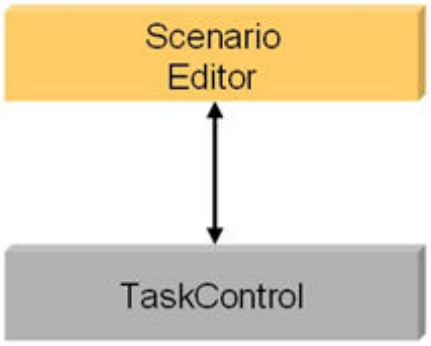


Figure 28. Connecting the ScenarioEditor

The connection may be UDP or via the loopback port (default). In case of a UDP connection, the TaskControl will send broadcast messages so that the ScenarioEditor may be located anywhere in the network. For loopback operation, the ScenarioEditor must run on the same computer as the TaskControl.

Installation / Options

The ScenarioEditor is located at	`Runtime/Core/ScenarioEditor`	
Name of the executable:	`scenarioEditor`	
Command line parameters:	none	
Environmental variables:	`DIGUY`	installation directory of `DI-Guy`
	`LANG`	language settings, default: "`en_US.UTF-8`

Start Procedure

The start parameters of the ScenarioEditor are configured in the file

`Data/Setups/Current/Config/SimServer/simServer.xml`

The editor is started via script:

Directory:	`Setups/Current/Bin`
Name of the process:	`scenarioEditor`
Command line:	`startTaskScenarioEditor`

Alternatively, it may be started via GUI, using `Tools-ScenarioEditor`

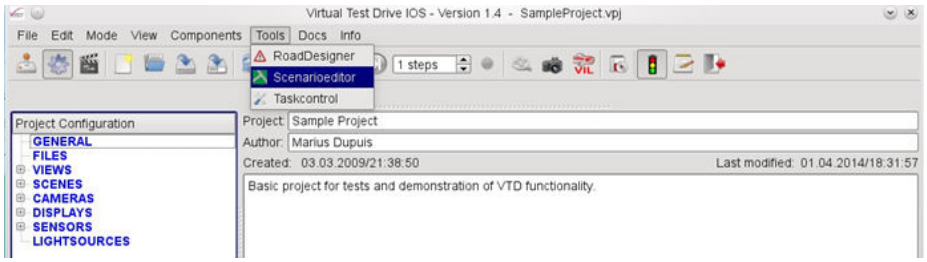

Virtual Test Drive IOS - Version 1.4 - SampleProject.vpj
File Edit Mode View Components Tools Docs Info
RoadDesigner
Scenarioeditor
Taskcontrol
1 steps
Project Configuration
GENERAL
FILES
VIEWS
SCENES
CAMERAS
DISPLAYS
SENSORS
LIGHTSOURCES
Project: Sample Project
Author: Marius Dupuis
Created: 03.03.2009/21:38:50
Last modified: 01.04.2014/18:31:57
Basic project for tests and demonstration of VTD functionality.

Image Generator (IG)

This chapter contains information regarding the image generator (IG) properties, which are relevant to the VTD application. Various versions of the IG (Standard, HDR, Optix, PBR) are available. Only the standard version will be described here. The image generator requires NVIDIA graphics cards. The faster, the better!

Image Generation

Need an introductory line regarding what this is:

- **Standard**: At each given point in time, each instance of the IG may render exactly one image from a given eyepoint. The rendered image is displayed on a monitor.
- **Special Output Formats**: In addition to displaying the image on a monitor, the image data of either the color or the depth buffer may also be transferred to the TaskControl via network or shared memory.
- **Real-Time Capabilities**: The real-time behavior of the image generator depends to a great extent on the underlying hardware. CPU speed, graphics card, and complexity of the database (content, number of animated vehicles) are the main influencing factors.
  The image generation rate strongly influences real-time behavior. For interactive simulations, a frequency of 60Hz or 120Hz is typically used. In this case, the IG will also be used as a frame-trigger for the entire simulation.
  Complex image renderings and image transfers will usually interfere with real-time requirements. In this case, the IG will be operated in frame-synchronous mode, i.e., it delivers images upon discrete requests and is tightly linked to the TaskControl.

Databases

The databases which are to be displayed must be available in one of the following file formats:

- OpenFlight
- OpenSceneGraph IVE
- OpenSceneGraph OSGB (preferred)

The best performance is achieved with databases that have been generated and optimized with the road designer ROD.

Animations

- **Vehicles**: Vehicles may only be animated if they comply with the internal model hierarchy. Its description may be disclosed upon explicit request.
- **Pedestrians**: For the display of pedestrians, the library "DI-Guy" from "Boston Dynamics" is used. This requires the purchase of additional licenses. VTD 2.0 and onwards only supports DI-Guy Version 13 and higher.

Overlays and Symbols

- **3D Objects**: Sensor cones may be displayed as frusta composed of a series of lines. They may be activated via the GUI.
- **2D Symbols**: Symbols that have been specified in a configuration file in advance (see below) may be shown as textured areas. Two modes for the symbol positions are available:
  - Screen
  - 3d space

  The activation and configuration of the symbols are performed during run-time via the TaskControl using SCP commands (see Simulation Control Protocol (SCP) [131]). Symbols in

space may be positioned at absolute (inertial) positions or relative to a player. The orientation will always be facing the camera (billboard).

- **2D Line Objects**: 2D-line objects may be positioned in space. The number of lines per symbol is unlimited. However, it should be taken into account that a high number of lines may considerably reduce the image generator's performance. Line objects may be positioned relative to a player. Shape, color, and line width may be defined via SCP (see Simulation Control Protocol (SCP) [98]).
- **Text Symbols:** Text may be displayed on-screen or in 3d-space. As with the other symbols, positions in space may be absolute (inertial) or relative to a player. Text size and color may be defined per text object. The configuration is performed via SCP (see Simulation Control Protocol (SCP) [98]).

Weather

The standard simulation is performed in dry weather conditions. The sky model may be switched into one of five different states resembling various cloud conditions. The daytime may be varied continuously, affecting the calculation of real-time shadows and the switching of vehicle head-lights.

The visualization of rainy weather may be performed by selecting the corresponding conditions in the GUI.

Illumination of the Scene

The scene may be illuminated via the sun or artificial light sources.

- **Sunlight/Shadow**: The position and color of the sun depending on the actual time of day. The position of the sun is given via an ephemeris model. The color composition is defined in a calibration table in the IG configuration files. The user may adapt to this table. The IG provides real-time full-scene shadow, i.e., each object casts shadows according to the sun's actual position.
- **Artificial Light Sources**: Artificial light sources that influence the scene are usually attached to their own vehicle or other players. To minimize the performance requirements, only the light sources of the own vehicle (or explicitly defined light sources) may illuminate the entire scene. Other players' light sources usually only illuminate an area on the street right in front of their current location.

  The shape and color of light sources may be defined via textures. These are referred to in the IG configuration file. Unless light sources are linked to the own vehicle or are of the simplified type attached to other players, they may be defined via the TaskControl using the RDB and SCP protocols (see Simulation Control Protocol (SCP) [98]).

Camera Frustum

The Task Control manages camera frusta and projection matrices (see there). For the camera model, the intrinsic parameters may be used in order to compute the projection matrix. The following parameters are relevant:

image width	[pixel]
image height	[pixel]
focal point x / y	[pixel]
principal point x / y	[pixel]
near clip	[m]
far clip	[m]

The TaskControl computes a 4 x 4 projection matrix from these parameters, which is sent to the image generator. The computation of the projection matrix is as follows:

```
matrix[0]  = 2.0 * focalX / width;
matrix[5]  = 2.0 * focalY / height;
matrix[8]  = -2.0 * ( principalX / width - 0.5 );
matrix[9]  = 2.0 * ( principalY / height - 0.5 );
matrix[10] = -( far + near ) / ( far - near );
matrix[11] = -1.0;
matrix[14] = -2.0 * far * near / ( far - near );
```

All other values are zero.

Operation

The image generator provides a means to navigate through a database using mouse and keyboard independent of the co-ordinates received via a network.

- **Keyboard controls:**

~~F1~~	~~switch to GAME model~~
~~F2~~	~~switch to TRACK model~~
~~F3~~	~~switch to DRIVE model~~
~~F4~~	~~switch to MOTIONREPLAY model~~
~~F5~~	~~switch to FLY model~~
F6	re-connect to network eyepoint
0..9	jump to reset position 0..9
.	take a screenshot (if FramebufferReader is configured); the image will be saved at Data/Setups/Current/Bin
PgUp	set eyepoint independent of network relative to next displayed vehicle model (use a mouse and left/right mouse button to change eyepoint)
PgDown	set eyepoint independent of the network relative to previous displayed vehicle model (use a mouse and left/right mouse button to change eyepoint)

  - **DRIVE MODEL**: Drive motion model resembles a car. It employs an intersection to find the terrain height. A mouse is used for acceleration, deceleration, and turning.
  - **FLY MODEL:** The FLY_MODEL is a simple roll rate model with no physics. It integrates for the each time step v = a * t and x = v * t

- **Keyboard controls:**

a	accelerate
s	stop translations
d	decelerate
p	print position
cursor left	slew left
cursor right	slew right
cursor up	slew up
cursor down	slew down
SHIFT	acceleration multiplier for translations

- **Mouse controls:**

mouse centered in channel	no rotation
mouse left/right	roll
mouse up/down	pitch
left/right mouse buttons	yaw
scroll wheel up/down	inc-/decrease rotation accelerations by 20%
SHIFT scroll wheel up/down	inc-/decrease translation accelerations by 20%

  - **GAME MODEL**: In-game motion model direction is changed with left click + mouse move. Arrow buttons are used to go forwards, backward, and sidewards. Using mouse wheel speed can be adjusted. Shift + mouse wheel changes the rotation speed.
  - **TRACK FOLLOW MODEL**: In track following motion mode Motion component follows a pre-loaded track. Use the following:
    - **+** and **–** to change the current track.
    - **a** to accelerate
    - **d** to decelerate
    - **s** to stop.
    - **Left click + mouse** to move direction relative to track direction can be changed
    - **Arrow** buttons to move vertically and laterally.

Files and File Formats

The image generator (v-IG) is used for various tasks in driving and flight simulation applications. Here, v-IG will only be described to the extent necessary for its use within Virtual Test Drive. Figure 29 shows the main file dependencies of v-IG.

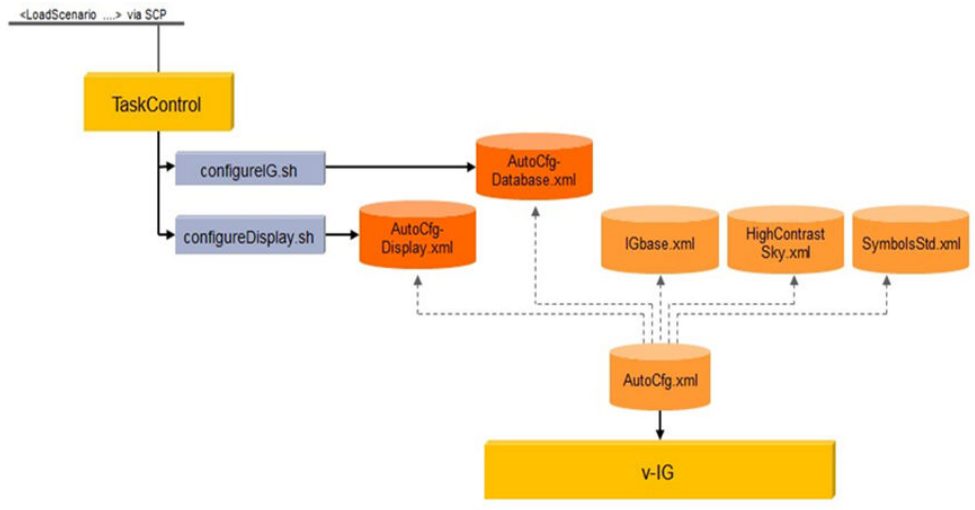


Figure 29. File Dependencies of the Image Generator

v-IG cannot exchange the loaded database on-the-fly. Therefore, each re-configuration of a scenario that also changes the applicable visual database will cause the TaskControl to terminate the running v-IG process, reconfigure it, and re-start it.

If you perform a resta re-start, the following actions will be performed:

termination of running v-IG	
execution of script	`Data/Scenarios/Common/Scripts/configureIG.sh`
execution of script	`Data/Scenarios/Common/Scripts/configureDisplay.sh`
start of v-IG	

As shown in File Dependencies of the Image Generator [75], the user may perform individual IG configurations by adjusting the files referenced by the file `AutoCfg.xml`. Unless the automatic IG configuration is turned off, the user should not edit the files `AutoCfgDatabase.xml` and `AutoCfgDisplay.xml`.

`AutoCfg.xml`, references:

`IGbase.xml`

`HighContrastSky.xml`

`SymbolsStd.xml`

`AutoCfgDatabase.xml`

`AutoCfgDisplay.xml`

Other files may also be referenced, depending on the individual configuration.

The `AutoCfgDatabase.xml` file is created via the `configureIG.sh` script. The TaskControl will invoke this script. The script will not be called only if the automatic configuration is disabled in the TaskControl's configuration file.

Output Window

The window for graphical output is defined in the file `AutoCfgDisplay.xml`. If in automatic configuration mode, this file will be generated by the script `configureDisplay.sh`, invoked by the TaskControl.

Example:

```
<IGconfig>
  <SystemConfig>
   <Graphics smallFeatureCullPixelSize="3.9"  lodscale="0.9" gamma="1 1 1"
enableCursor="0" enableDatabasePagerThread="0" drawThreadCPUAffinity="1"
appCullThreadCPUAffinity="0" enableTransparencyAntialiasing="1"
        enableMultisampling="1">
       <RenderSurface name="mainRS" x="0" y="600" width="800" height="600"
         depthBits="24" sampleBuffers="0" samples="0" borderVisible="1"
overrideRedirect="0"/>
         <Camera name="cam1" renderSurface="mainRS" viewPortX="0" viewPortY="0"
viewPortWidth="800" viewPortHeight="600">
   <SymmetricPerspectiveAngles  fovX="45.00" fovY="30.00" near="1" far="1500"
        offsetHPR="0 0 0" offsetXYZ="0 0.0 0"/>
      </Camera>
    </Graphics>
  </SystemConfig>
</IGconfig>
```

Explanations are provided for relevant items only.

The structure of the definition complies with XML hierarchy rules.

`<Graphics>`

`lodScale`: global scale of level-of-detail

<gamma>:	global gamma correction of image output

<RenderSurface>:

depthBits:	XServer bit depth [bit]
x:	x position on screen [pixel]
y:	y position on screen [pixel]
width:	size in x direction [pixel]
height	size in y direction [pixel]
borderVisible:	show/hide window border [1 / 0]
overrideRedirect:	force window to run at given position; override XServer [0 / 1]
displayNum:	number of the X display on which to open the render surface
screenNum:	number of the X screen on which to open the render surface

<Camera>

viewPortX	x position of viewport within render surface [pixel]
viewPortY	y position of viewport within render surface [pixel]
viewPortWidth	size of viewport in x direction [pixel]
viewPortHeight	size of viewport inin y direction [pixel]

<SymmetricPerspectiveAngles>:

fovX:	horizontal field-of-view [deg]
fovY:	vertical field-of-view [deg]
near:	near clipping plane [m]
far:	far clipping plane [m]
offsetHPR:	angular offset from camera position [deg / deg / deg]
offsetXYZ:	spatial offset from camera position [m / m / m]; note: the coordinate system is different from the vehicle system; the correlation between both is gfx X = -vehicle Y gfx Y = vehicle X gfx Z = vehicle Z

Instead of the symmetric viewing frustum, an asymmetric frustum may be defined:

`<AsymmetricPerspectiveAngles>`:

`left`:	field-of-view to the left [deg]
`right`:	field-of-view to the right [deg]
`bottom`:	field-of-view to the bottom [deg]
`top`:	field-of-view to the top [deg]

`near`:	near clipping plane [m]
`far`:	far clipping plane [m]
`offsetHPR`:	angular position from camera position [deg / deg / deg]
`offsetXYZ`:	spatial offset from camera position [m / m / m]
	the co ordinate system is different from vehicle system
	the correlation between both is
	gfx X = -vehicle Y
	gfx Y = vehicle X
	gfx Z = vehicle Z

System Configuration

The file `IGbase.xml` contains the basic settings of the system. Apart from performance parameters and plug-in configurations, file paths to databases, shaders, symbols, etc., are defined.

Example:

```
<SystemConfig>
    <EnvVariable value="5" var="__GL_FSAA_MODE" />
    <EnvVariable value="1" var="__GL_SYNC_TO_VBLANK" />
    <EnvVariable value="4" var="__GL_LOG_MAX_ANISO" />
    <EnvVariable value="WARN" var="OSG_NOTIFY_LEVEL" />
    <EnvVariable value="/usr/local/DI-Guy_8.0.5" var="DIGUY" />
    <FilePath path="../data" />
    <FilePath path="../data/Fonts" />
    <FilePath path="../data/Shaders" />
    <FilePath path="../data/ObjectsCommon" />
    <FilePath path="../../Data/Database/ObjectsCommon" />
    <FilePath path="../../Data/Database/Cars" />
    <FilePath path="../../Data/Database/AudiTerrain" />
    <FilePath path="../../Data/Database/Town/TexturePoolCommon" />
    <FilePath path="../../Data/Database/Town/TexturePoolGer" />
    <FilePath path="../../Data/Database/Town/TexturePoolUK" />
    <Notification notifyLevel="WARN"/>
</SystemConfig>
```

`SystemConfig`

`EnvVariable` Set an environment variable as key (`var`) – value (`value`) -pair.

WARNING

Wrong settings may drastically reduce the performance of v-IG.

The main environment variables are:

- `__GL_FSAA_MODE:` Antialiasing-Mode, depending on graphics card
- `__GL_SYNC_TO_VBLANK:` must be 1
- `__GL_LOG_MAX_ANISO:` maximum computed anisotropy
- `OSG_NOTIFY_LEVEL:` OpenSceneGraph notification level
- `DIGUY:` Installation directory of DI-Guy

This is required only if DI-Guy is to be used and if the variable set in the `envSettings.cfg` of the Setup are not applicable.

`FilePath` Search path for files that are to be loaded by v-IG

`Notification` Minimum criticality of messages which are to be displayed. Possible values are:

`FATAL`

`WARN`

`INFO`

`NOTICE`

`DEBUG`

Symbols

Symbols are usually defined in the file `SymbolsStd.xml`.

For 2D symbols Lower left corner of the screen is 0,0, and the upper right is 1,1. Note that this does not account for the screen aspect ratio. To draw symbols with automatically correct aspects, activate the **autoHeight** feature of the symbol (from XML or code). If this is done, width of a symbol will be taken from user input, and the height of the symbol will be computed using the texture aspect ratio and screen aspect ratio. In effect, a symbol that has the same look as in the image file is obtained.

For this to work correctly, you have to define screen Aspect Ratio in the XML configuration. It should be defined in the <symbols> tag like:

```
<symbols name="MySymbols" screen aspect ratio="1.3333">
```

Screen aspect ratio is a hardware property (so it is not the aspect ratio of the window). A standard monitor has 4/3 ar. and this is also the default value.

Line symbols are simple lines with user-definable colors. Corner points of the lines have to be provided by the user. The corners are three dimensional and are in world coordinates. Line symbols will go through viewing and projection transformations as any other 3D object in the scene. 2D symbols are, unlike line symbols, in screen coordinates, i.e., even though the camera's direction, position, or projection changes, 2D symbols will remain in the same position. It is possible to draw a user-defined text associated with line symbols. The text's position is also in 3D; however, the text plane is always parallel to the screen, and the size of the text is constant.

Textured polygons are a mixture of line symbols and 2D symbols. A line symbol that refers to a 2D symbol as a texture source is interpreted as a polygon by the component. If *useOverlayAsTexture* argument in LineSymbol constructor is not equal to -1 line symbol will be drawn as a textured polygon. In this case, users have to provide four corner points in the *lines* argument. The texture of the 2D symbol referred to by *useOverlayAsTexture* will be mapped onto this polygon.

An example of the XML configuration is provided below.

```
<Symbols name="MySymbols">
        <Symbol id="0" x="0.55" y="0.2" width="0.1" height="0.1" scale="1.0"
              enabled="0">
            <Texture file="../data/Misc/ACC_LDW_1Balken_transparent.png"/>
            <Texture file="../data/Misc/ACC_LDW_2Balken_transparent.png"/>
            <Texture file="../data/Misc/ACC_LDW_3Balken_transparent.png"/>
            <Texture file="../data/Misc/ACC_LDW_4Balken_transparent.png"/>
            <Texture file="../data/Misc/LDW_Pfeile.tga"/>
            <Texture file="../data/Misc/Kollisionswarnsymbol_transparent.rgb"/>
        </Symbol>

        <Symbol id="4" x="0.3" y="0.2" width="0.03" height="0.03" scale="1.0"
              enabled="0">
            <Texture file="../data/Misc/hudDigit_0.rgb"/>
            <Texture file="../data/Misc/hudDigit_1.rgb"/>
            <Texture file="../data/Misc/hudDigit_2.rgb"/>
            <Texture file="../data/Misc/hudDigit_3.rgb"/>
            <Texture file="../data/Misc/hudDigit_4.rgb"/>
            <Texture file="../data/Misc/hudDigit_5.rgb"/>
            <Texture file="../data/Misc/hudDigit_6.rgb"/>
            <Texture file="../data/Misc/hudDigit_7.rgb"/>
            <Texture file="../data/Misc/hudDigit_8.rgb"/>
            <Texture file="../data/Misc/hudDigit_9.rgb"/>
   </Symbol>     </Symbols>
```

`Symbols`

`Symbol` Multiple symbol entries can be defined in a component.

- *id*: An integer id that has to be unique within a Symbols instance. The component does not check uniqueness.
- *x* and *y*: Normalized screen positions of the symbol. The bottom left of the screen is (0,0), and the upper right is (1,1).
- *width* and *height*: Normalized size of the symbol.
- *scale*: Scale of the symbol. Actual size of the symbol will be (width*scale, height*scale).
- *enabled*: show symbol upon start-up of the image generator (0/1).
- *Texture*: <texture> entries define the texture that can be mapped onto the 2D symbol. Multiple texture entries can be defined. The first texture corresponds to the 0th state of Symbols::Symbol.
- *file*: Texture file name.

Vehicles

Vehicles are usually configured in the directories `Data/Distros/Current/Databases/Cars` and `Data/Distros/Current/Config/Players/Vehicles`. One file is given per vehicle model, which specifies a vehicle's 3d mesh (Database) and occurrence (Config). Upon initializing the simulation, the vehicle definition files are read by the traffic simulation and are then forwarded to v-IG via the TC.

Pedestrians

For pedestrians, the 3rd party software DI-Guy has to be installed, and a valid license file must be present. Pedestrians are usually configured in the file `Data/Distros/Current/Config/Players/characterCfg.xml.` Upon initializing the simulation, the character definition file is read by the traffic simulation, and the relevant content is then forwarded to v-IG via the TC.

Sky/Lighting

The parameters of the sky model may be defined in the file `HighContrastSky.xml.`

- This class implements a sky model and atmospheric events such as fog.
- By default OpenGL light 0 is used for sun or moonlight. Ambient, diffuse, and specular components of the light can be set.
- To serve as background, an ive file, Skybox.ive, is loaded. This class also manages atmospheric fog.
- It is possible to define certain skylight, background, and fog parameters in a table as a sun elevation function. Sky automatically interpolates the values depending on current time-of-day.

An excerpt of the API documentation (XML configuration) is as follows:

```
<Sky lightID="0" activeSky="1" name="MySky" >
      <SkyModel followInZ="1" />
      <SunModel sunModelDistance="400" size="12" textureFile="sun.png" />
<ColorTable>
<Entry isSunVisible="1" skyColor="0.05 0.05 0.05" elev="-1.0" sunLightAmbient="0
       0 0" sunColor="0 0 0" sunLightDiffuse="0 0 0"
       sunLightSpecular="0.0 0.0 0.0" fogColor="0.05 0.05 0.05" />
<Entry isSunVisible="1" skyColor="0.2 0.2 0.2" elev="-0.15"
       sunLightAmbient="0.01 0.01 0.02" sunColor="1 0 0" sunLightDiffuse="0 0 0"
       sunLightSpecular="0.0 0.0 0.0" fogColor="0.3 0.3 0.3" />
<Entry isSunVisible="1" skyColor="0.25 0.25 0.25" elev="-0.1"
       sunLightAmbient="0.0 0.0 0.0" sunColor="1 0.5 0.5" sunLightDiffuse="0 0 0"
       sunLightSpecular="0.0 0.0 0.0" fogColor="0.45 0.45 0.45" />
<Entry isSunVisible="1" skyColor="0.27 0.27 0.27" elev="-0.066"
       sunLightAmbient="0.0 0.0 0.0" sunColor="1 1 0.8" sunLightDiffuse="0 0 0"
       sunLightSpecular="0.1 0.1 0.0" fogColor="0.47 0.47 0.47" />
<Entry isSunVisible="1" skyColor="0.3 0.3 0.3" elev="-0.009"
       sunLightAmbient="0.0 0.0 0.0" sunColor="1 1 0.8"
       sunLightDiffuse="0.2 0.2 0.25" sunLightSpecular="0.2 0.2 0.0"
       fogColor="0.5 0.5 0.5" />
<Entry isSunVisible="1" skyColor="0.4 0.4 0.4" elev="0.01"
       sunLightAmbient="0.00.0 0.0" sunColor="1 1 0.8"
       sunLightDiffuse="0.4 0.4 0.5" sunLightSpecular="0.3 0.3 0.2"
       fogColor="0.6 0.6 0.6" />
 <Entry isSunVisible="1" skyColor="0.7 0.7 0.7" elev="0.08"
       sunLightAmbient="0.05 0.05 0.05" sunColor="1 1 0.8"
       sunLightDiffuse="0.70.7 0.6" sunLightSpecular="0.85 0.85 0.7"
       fogColor="0.7 0.7 0.7" />
 <Entry isSunVisible="1" skyColor="0.9 0.9 0.9" elev="0.5"
       sunLightAmbient="0.40.4 0.4" sunColor="1 1 0.8"
       sunLightDiffuse="1.0 1.0 1.0" sunLightSpecular="1.0 1.0 0.9"
       fogColor="0.9 0.9 0.9" />
 <Entry isSunVisible="1" skyColor="1 1 1" elev="1" sunLightAmbient="0.5 0.5 0.5"
       sunColor="1 1 0.8" sunLightDiffuse="1.0 1.0 1.0"
       sunLightSpecular="1.0 1.0 0.95" fogColor="1 1 1" />
</ColorTable>
 <Fog mode="FOG_OFF" density="3.9" end="100" start="5" />
</Sky>
```

`Sky`

- *lightID* : OpenGL light that will be used as sunlight.
- *activeSky* : Active sky background. An integer value is expected.

`SkyModel`

followingZ : A boolean ("0" or "1") attribute. If followInZ is "1"sky box model's center follows the camera position also in Z direction. Normally the camera is followed only in XY plane.

`SunModel`

- sunModelDistance : Distance of the textured quad that represents the sun from the eyepoint.
- size : Size of the textured quad that represents the sun.
- textureFile : The texture that has to be mapped onto the quad.

`ColorTable`

Here multiple entries are used to associate certain lighting values with the sun's elevation.

Fog

- mode : Recognized strings are FOG_OFF, LINEAR, EXP, and EXP2. Default is FOG_OFF.
- density : A floating-point number that determines the density of the fog in EXP and EXP2 modes.
- start and *end* : Fog start and end distances for LINEAR fog.

Headlights

Vehicle headlights, i.e., light sources, are defined in the file `LightSrcStd.xml`.

- By default, Headlights do not light the entire scene. The parts of the scene that has to be lit should be given to the component explicitly using addNodeToHeadlights method.
- Headlights are defined as templates in a so-called headlight factory.
- During run-time, these templates may be assigned to various instances of headlights.
- The first template definition may be afterward referred to as template no. 0, the second as template no. 1, etc.

An excerpt of the API documentation (XML configuration) is as follows:

```
<IGconfig>
  <Components>
    <HeadlightFactory name="MyHeadlightFactory">
    <HeadlightVIGLighting name="MyHeadlights1" >
<HeadlightTextures>
<LowBeamTexture            file="TxLowBeam_white.tga"  />
<HighBeamTexture            file="TxHighBeam_white.tga" />
<TopBeamTexture            file="bluespot.tga"             />
<LowBeamAndTopBeamTexture        file="redspot.tga"              />
<HighBeamAndTopBeamTexture     file="greenspot.tga"          />
<FoglightTexture               file="bluespot.tga"            />
<LowBeamWithFogTexture         file="redspot.tga"              />
<HighBeamWithFogTexture        file="greenspot.tga"            />
</HeadlightTextures>
<HeadlightVertexShader file="../data/Shaders/lightVert.glsl" />
<HeadlightFragmentShader file="../data/Shaders/lightFrag.glsl" />
<Multisampling samples="4" />
<ShaderAssociations>
<ShaderPair name="HeadlightTree"
    vertexshader="../data/Shaders/lightRotVert.glsl"
    fragmentshader="../data/Shaders/lightFrag.glsl"/>
<ShaderPair name="HeadlightCar"
    vertexshader="../data/Shaders/lightVert.glsl"
    fragmentshader="../data/Shaders/lightCarFrag.glsl"/>
<ShaderPair name="HeadlightCarWheels"
    vertexshader="../data/Shaders/lightVert.glsl"
     fragmentshader="../data/Shaders/lightFrag.glsl"/>

<texture filename="Tx*Veg*"     shaderPair="HeadlightTree"     />
<texture filename="TxVeg*"     shaderPair="HeadlightTree"     />

<node name="L1_BodyEnv"         shaderPair="HeadlightCar"     />
<node name="L2_BodyEnv"         shaderPair="HeadlightCar"     />
<node name="L3_BodyEnv"         shaderPair="HeadlightCar"     />

<node name="L1_Wheels"             shaderPair="HeadlightCarWheels"     />
<node name="L2_Wheels"             shaderPair="HeadlightCarWheels"     />
<node name="L3_Wheels"             shaderPair="HeadlightCarWheels"     />
<node name="L1_BodyParts0"     shaderPair="HeadlightCar"     />
<node name="L1_BodyParts1"     shaderPair="HeadlightCar"     />
<node name="L1_BodyParts2"     shaderPair="HeadlightCar"     />
<node name="L1_BodyParts3"     shaderPair="HeadlightCar"     />
<node name="L1_BodyParts4"     shaderPair="HeadlightCar"     />
<node name="L1_BodyParts5"     shaderPair="HeadlightCar"     />

<node name="L2_BodyParts0"     shaderPair="HeadlightCar"     />
<node name="L2_BodyParts1"     shaderPair="HeadlightCar"     />
<node name="L2_BodyParts2"     shaderPair="HeadlightCar"     />
```

```
<node name="L2_BodyParts3"     shaderPair="HeadlightCar"     />
<node name="L2_BodyParts4"     shaderPair="HeadlightCar"     />
<node name="L2_BodyParts5"     shaderPair="HeadlightCar"     />

<node name="L3_BodyParts0"     shaderPair="HeadlightCar"     />
<node name="L3_BodyParts1"     shaderPair="HeadlightCar"     />
<node name="L3_BodyParts2"     shaderPair="HeadlightCar"    />
<node name="L3_BodyParts3"     shaderPair="HeadlightCar"     />
<node name="L3_BodyParts4"     shaderPair="HeadlightCar"     />
<node name="L3_BodyParts5"     shaderPair="HeadlightCar"     />
</ShaderAssociations>
<RenderTargetTexture width="800" height="600" />
<HeadlightAttenuation    constantAttenuation="0.3"   linearAttenuation="0.02"
quadraticAttenuation="0.0005" />
<HeadlightIntensity ambientIntensity="0.2" diffuseIntensity="1.9"
specularIntensity="1.0" lightDistributionEffectOnAmbientIntensity="1"/>
  <HeadlightOffset      xyz="0 0.0 0" hpr="0 0 0"/>
  <HeadlightFrustum left="-1.7" right="1.7" bottom="-1.5" top="1.5" near="1"
  far="100" />
<HeadlightPassNearFar near="1" far="200" />
     </HeadlightVIGLighting>
    </HeadlightFactory>
   </Components>
</IGconfig>
```

`HeadlightTextures`

LowBeamTexture

file: Texture file to be used for low beam.

HighBeamTexture

file: Texture file to be used for high beam

The complete list is as follows. Each state is also associated with a numerical id, which might be required when sending a state via network communication:

LowBeamTexture	state 1
HighBeamTexture	state 2
TopBeamTexture	state 3
LowBeamAndTopBeamTexture	state 4
HightBeamAndTopBeamTexture	state 5
FoglightTexture	state 6
LowBeamWithFogTexture	state 7
HighBeamWithFogTexture	state 8

`HeadlightVertexShader`

file : Default vertex shader.

`HeadlightFragmentShader`

file : Default fragment shader

`ShaderAssociations`

Normally headlight rendering is done with default shaders as defined above. For special cases, however, special shaders might be necessary. In this section, these special shaders and where they have to be used can be defined.

`ShaderPair`

A pair of vertex and fragment shaders can be referenced in the section.

Multiple shader pair's can be defined.

name : Name of the shader pair. Should be unique within a Headlights instance.

vertexshader : Vertex shader file name.

fragmentshader : Fragment shader file name.

`texture`

Multiple texture entries can be used to attach certain shader pairs to certain textures in the scene. *filename* : Texture file name.

shaderPair : Shader pair to be attached to the geometries that have this texture.

`node`

Multiple node entries can be used to attach certain shader pairs to certain nodes.

name : Node name.

shaderPair : Shaer pair to be attached to the node.

`RenderTargetTexture`

width : Width of the render target texture. Should be the same as screen width in pixels.

height : Height of the render target texture. Should be the same as screen height in pixels.

`HeadlightAttenuation`

constantAttenuation : Constant attenuation factor.

linearAttenuation : Linear attenuation factor.

quadraticAttenuation : Quadratic attenuation factor.

`HeadlightIntensity`

ambientIntensity : Intensity of the ambient part of the light.

diffuseIntensity : Intensity of the diffuse part of the light.

specularIntensity : Intensity of the specular part of the light.

`HeadlightOffset`

Headlights component is derived from Positionable and hence has its own position. In an application position, the component has to be set to be the same as the car's position. In this case, the offset that is defined here is the offset of the headlights relative to the car.

xyz : Position of the headlights.

hpr : Orientation of the headlights in degrees.

`HeadlightFrustum`

In this section, the frustum of the headlight projector can be defined.

left, *right*, *bottom,* and *top* : Positions of the frustum's edges on the near plane. Note that these values are distances, not angles. *near* and *far* : Distances of near and far planes.

`HeadlightPassNearFar`

The implementation of headlights is such that the headlights' contribution to the scene lighting is rendered to a texture, and then this texture is added to the existing frame buffer. When rendering the scene onto this headlight texture, one can define different near - far plane distances as normal rendering.

near : Near plane distance.

far : Near plane distance.

Rear-view Mirrors

Mirrors may be displayed as picture-in-picture. They must be defined in the file IGbase.xml. Here's the excerpt of the API documentation. Mirror component implements an in-screen mirror. The mirror is a 2D surface and not a 3D object in the scene. A camera, which looks in the negative Y direction, renders its view onto the mirror viewport. This camera does not render the entire scene; instead, a user-defined subset.

An excerpt of the API documentation (XML configuration) is as follows:

```
<Mirror screenPosX="450" screenPosY="800" sizeX="400" sizeY="100" fovV="13"
near="2" far="150" positionOffset="-0.5 0.3 0.2" orientationOffset="200 0 0"/>
```

`Mirror`

- *screenPosX* and *screenPosY:* Position of the mirror on the screen in pixels.
- *sizeX* and *sizeY:* Size of the mirror.
- *fovV* : Vertical field of view of the camera that renders mirror image. The horizontal field of view is automatically computed.
- *near* and *far* : Near and far plane distances of the camera that renders the mirror image.
- *positionOffset* : Mirror is Positionable; hence it has a position in a 3D world. This parameter defines an offset to this position. In an application, Mirror's position could be set to the ownship's position in every frame. In this case, *positionOffset* is the position of the mirror camera relative to the car.
- *orientationOffset* : may be oriented explicitly in a given direction relative to the camera direction. Angles for the three axes are given in degrees.

Motion Model

The motion models for the interactive navigation of a database-independent of a simulation may be pre-configured in the file `IGbase.xml.`

Example:

```
<Motion enableText="0" motionModel="GameMotion" name="MyMotion" >
 <TrackFollow speed="0" offset="0 0 3" acceleration="10" slowSlew="1"
fastSlew="15"
                  S="0.0" >
 <TrackFile   file="../../Data/Database/SmartDB/Tracks/trTST00979.trk" />
</TrackFollow>
       <Drive inputMethod="mouse" maxAcceleration="0.02" turnRate="25.1"/>
</Motion>
```

Keywords	Parameters	Unit	Default	Description
enableText	%d	[-]	0	switch position text overlay ON/OFF

Keywords	Parameters	Unit	Default	Description
motionModel	%s	[-]	Game	Active motion model, which is one of the following: • Drive • Fly • Game • MotionReplay • TrackFollow

DRIVE MODEL

Keywords	Parameters	Unit	Default	Description
inputMethod	"mouse" or "userDefined"	-	"mouse"	"mouse" connects mouse inputs to the motion model.
maxAcceleration	f	[m/s^2]	0.05	Maximum acceleration.
turnRate	f	[deg/s]	30	Turn rate.

FLY MODEL

Keywords	Parameters	Unit	Default	Description
boost	%f	[%]	10	multiplier for linear accelerations
collide	%d	[-]	0	collision on/off
dHPR	%f%f%f	[deg/s^2]	60 60 20	rotational accelerations
dXYZ	%f%f%f	[m/s^2]	0.5 5.0 0.5	linear accelerations (x=+right, y=+front, z=+up)

GAME MODEL

Keywords	Parameters	Unit	Default	Description
speed	f	[m/s]	0.005	Translational speed
turnSpeed	f	[deg/s]	90	Angular speed

TRACK FOLLOW MODEL

Keywords	Parameters	Unit	Default	Description
speed	f	[m/s]	0.0	Initial speed
acceleration	f	[m/s^2]	2.0	Acceleration that can be applied using keys
S	f	[m]	0	Initial S coordinate (position along a track)
turnSpeed	f	[deg/s]	0	Angular speed

Track files are provided as child nodes in XML.

```
<TrackFollow speed="0" offset="0 0 3" acceleration="10"  S="0.0" >
          <TrackFile file="../data/CityUK/Tracks/trIAA00000.trk" />
```

Connections

The connections between the IG and the TaskControl must be configured in the file IGbase.xml within the block <TAKATA>.

Example:

```
<TAKATA name="TAKATA"   :
taskControlServerAddress="127.0.0.1"
taskControlServerPort="13112"
connectTaskServerTCP="1"
imageTransferServerAddress="127.0.0.1"
imageTransferServerPort="13110"
connectImageTransferTCP="1"
:
>

TAKATA
```

Keyword for the block

`taskControlServerAddress`	Address of the computer which runs the TaskControl; default is localhost
`taskControlServerPort`	Port, via which to connect to the TaskControl, Default: 13112
`connectTaskServerTCP`	Enable/disable the connection: 0 / 1
`imageTransferServerAddress`	Address of the computer which shall receive the images; this should also be the computer running the TaskControl; default is localhost
`imageTransferServerPort`	Port via which images shall be transferred, default: 13110
`connectImageTransferTCP`	Enable/disable image transfer connection: 0 / 1
`trigger`	Define the trigger communication channel: "UDP" / "TCP"

Trigger

Per default, triggers are sent via TCP. The trigger communication mechanism may be explicitly set in `IGbase.xml` within the block <TAKATA>.

Example:

```
<TAKATA name="TAKATA"
:
trigger="TCP"
:
>
```

Valid values are: "UDP" / "TCP"

Input to IG via SharedMemory

Some parameters may be sent to the IG via shared memory. The RDB interface description defines the protocol for shared memory operation. The corresponding flag must be set in `IGbase.xml` within the block <TAKATA> to enable the reading of shared memory inputs.

Example:

```
 <TAKATA name="TAKATA"
:
enableShmReader="1"
:
  >
```

Valid values are: 0 / 1

Windshield and Wiper

A windshield with an animated wiper may be displayed for single-channel applications. To activate this feature, insert the following lines within the <Components> section of `IGbase.xml`:

Example:

```
<Windshield quadCount="1500" name="MyWindShield" maxAge="10.0"
dropsPerSecond="10"   dropSize="0.012 0.012" screenAspectRatio="0.75" >
<Wiper enabled="1" height="0.75" width="0.0375" wiperPivot="0.6 -0.05"
speed="150" />
</Windshield>
```

Elements:

Element	**Parent**			
Windshield	Components			
	Attribute	**Description**	**Unit**	**Default Value**
	quadCount	max. number of droplets	-	1500
	name	name of the feature	-	myWindShield
	maxAge	maximum age of a droplet	s	10.0
	dropsPerSecond	drops hitting the screen per second	1/s	10
	dropSize	[x y] size of a drop in screen co-ordinates	scrn coord.	0.012 0.012
	screenAspectRatio	aspect ratio of the screen (for drop distortion)	-	0.75

Element	**Parent**			
Wiper	Windshield			
	Attribute	**Description**	**Unit**	**Default Value**
	enabled	on/off switch	0 / 1	1
	height	height of the wiper texture	scrn coord.	0.75
	width	width of the wiper texture	scrn coord.	0.0375
	wiperPivot	[x y] pivot point of the wiper	scrn coord	0.6 -0.05
	speed	speed of wiper motion	deg/s	150

Communication

The image generator may be linked to the taskControl via up to three connections:

- UDP-Port for fast, non-deterministic run-time data
- TCP-Port for deterministic configuration
- TCP-Port for image transfer

In addition, the image generator may be linked to the sensor task (head tracker) via the HPRReader. Figure 30 illustrates the communication with the image generator:

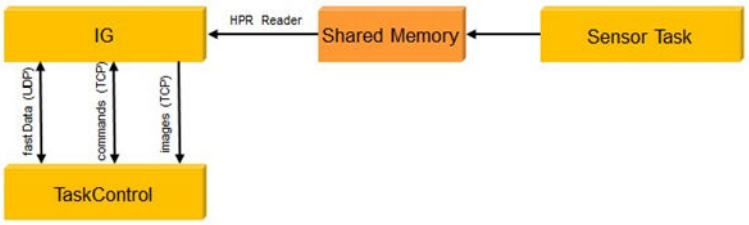


Figure 30. Connections of Image Generator

Installation/Options

The image generator is installed in the directory Runtime/Core/ImageGenerator. There, three sub-directories may be found:

- `bin/` all binary (executable) files
- `data/` additional basic data (optional)
- `doc/` specific documentation (optional)

Executable file:	`vigcar`	
Command line parameters:	`<cfgFile>`	name of the configuration file, usually: `AutoCfg.xml`
	`notrigger`	operation without generation of trigger messages; this mode is to be used for all instances of the IG which run in a multi-channel configuration concurrent to one master channel (e.g., side channels)
Environment variables:	Environment Variables [129]	

6.5.13.2 Pedestrians

The components of the pedestrian library DI-Guy" by "VT-MÄK are installed at the following location: `/usr/local/DIGuy_<versionNo.>.`

They may only be operated with the corresponding license.

Launch Procedure

The start parameters of the Image Generator are configured in the following file

```
Data/Setups/Current/Config/SimServer/simServer.xml
```

The image generator may only be operated online. Each graphics output channel requires one instance of v-IG (exception: picture-in-picture channels).

Name of the process: `vigcar`

TaskControl (TC)

The TaskControl (TC) is the central element of the simulation, managing the main command and data flow. It provides individual interfaces to each component and may serve various communication protocols. Figure 31 shows the embedding of the TaskControl within a typical simulation setup.

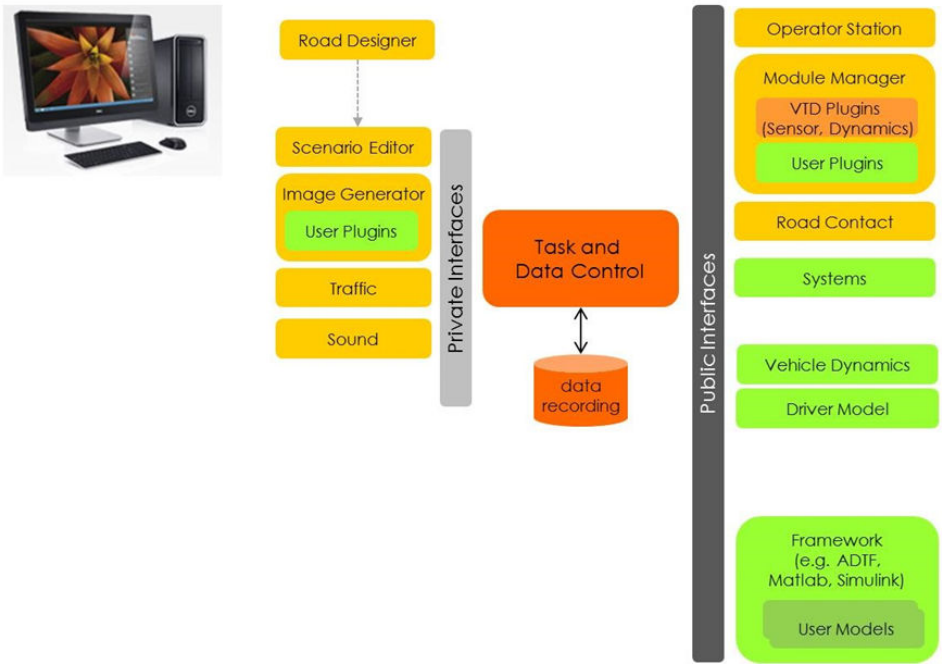


Figure 31. TaskControl within the Simulation

The TaskControl manages the timing of the entire simulation and the correct correlation of incoming and outgoing data.

Configuration

- **ParamServer Mode**: Starting with VTD 2.0, the TaskControl is configured via the ParamServer. Suppose you want to deviate from the default settings. In that case, you should configure the TaskControl accordingly using the Parameter Browser (see Resources [9]) and save the resulting parameter set in your current setup. Upon starting the simulation, load the saved parameter settings, and press **APPLY** in the GUI.
- **Legacy Mode:** The TaskControl may be still be configured via the `taskControl.xml` file which may be located in the `Data/Setups/Current/Config/TaskControl` directory . For further instructions concerning the legacy mode, see see Resources [9].
  All configuration parameters of the TaskControl comply with the SCP syntax (see separate documentation) and are defined within the command area `<TaskControl>`. For a complete list, see Resources [9].

Simulation Control

As the name already indicates, the TaskControl controls the entire simulation. This may be done in various modes, which are described in the following chapters.

Component Status

The components of the simulation send status messages to the TaskControl. These are forwarded to the GUI and may be displayed there (with colors indicating the various states).

Components that do not or no longer send a status message will automatically be set to **FAIL** after 1.5 seconds. If the taskControl fails or hangs, the GUI will stop displaying any status button after approx. two seconds (the component buttons will turn purple shortly before)

Synchronization / Real-time modes

All components which run as isolated processes may potentially do so in an asynchronous manner. For interactive simulations (driver-in-the-loop, vehicle-in-the-loop), this is actually the preferred mode since it can minimize discontinuities if components don't comply with real-time requirements for a few frames.

The synchronization is configured via the SCP parameters

```
<TaskControl>
  <Sync source="a" frameTimeMs="b" realTime="c"/>
</TaskControl>
```

The following list provides an overview of the different settings:

source="intern" frameTime="x.y":

The TaskControl is the master of the simulation and provides the frame ticks. Per frame, the simulation time will be increased by the given frame time in [ms] realtime="false": TC runs as fast as possible. This may be faster or slower than real-time (depending on the complexity of the simulation) realtime="true": TC tries to match exactly the time given in frameTimeMs and, thus, to provide real-time behavior. Usually, this should be possible for frame times higher than 15ms.

`source="intern"frameTime="x.y":`

The TaskControl is the master of the simulation and provides the frame ticks. Per frame, the simulation time will be increased by the given frame time in [ms]

`realtime="false":` TC runs as fast as possible. This may be faster or slower than real-time (depending on the complexity of the simulation).

`realtime="true":` TC tries to match exactly the time given in `frameTimeMs` and, thus, to provide real-time behavior. Usually, this should be possible for frame times higher than 15ms.

`source="extent" frameTime="x.y":`

The Iamge Generator (IG) provides the frame ticks.

`realtime="false":` TC runs synchronous to the ImageGenerator. In each frame, the simulation time will be increased by the given frame time in [ms] so that the entire simulation may run faster or slower than real-time (depending on the complexity of the simulation).

`realtime="true":` The frame time given as a configuration parameter will be ignored. Instead, the ImageGenerator provides the ticks and the delta frame time. In each frame, the simulation time will be increased by this delta frame time. The simulation runs in real-time. The typical frame rate of the IG is 60Hz, resulting in a frame time of 16.666667ms (three frames per 50ms).

`source="RDB":`

The sync signal is sent via RDB port by an external component.

`source="SCP":`

The sync signal is sent via SCP commands by an external component.

Components may register to become additional sync sources, i.e., that without their corresponding signal, the next simulation frame will not be released. For details, see [9]. Sometimes sync sources do not prove as reliable as they should. The TC may be globally set to ignore all additionally registered sync sources – e.g., for debugging purposes – by providing the following configuration line:

```
<TaskControl>
  <RDB ignoreSync="true"/>
</TaskControl>
```

Link to Vehicle Dynamics (VD)

The timing between an external vehicle dynamic and the TaskControl can be configured with the

SCP parameter

```
<TaskControl>
<Dynamics syncMode="a"/>
</TaskControl>
```

Possible settings are:

`syncMode="frame":`

TC and VD run in a strict sequence, i.e., the TC will only continue after it has received a full RDB frame (i.e., the END_OF_FRAME message, see [9]) from the external component or after a time-out; if you are not using a VD or if you want to prevent the time-out, then set the following configuration parameters

```
<TaskControl>
<RDB ignoreEndOfFrame="true"/>
</TaskControl>
```

`syncMode="free":`

TC and VD run asynchronously, i.e., VD runs autonomously in real-time and sends its results whenever available. This mode is primarily recommended for interactive real-time simulation. It is not deterministic!

Interfaces

TC provides various interfaces. Most of them emerge from the internal use of the software and will not be described in detail here. The interfaces which are open and relevant for the user are:

- **Simulation Control Protocol (SCP)**: The SCP is used to exchange commands between simulation components and to control the simulation from outside. It is based on a TCP server port opened by the TC, which accepts up to five clients. Each command sent to the TC will be distributed to all connected clients. However, some exceptions apply for commands which are known to be relevant for certain components only.
- **Runtime Data Bus (RDB-data)**: The RDB (data) distributes run-time data about objects, vehicle states, etc., to any data consumer. The interface may be configured to run via TCP, UDP, shared memory, or loopback port. In TCP, the TC will act as a server and accepts up to 16 clients.

The configuration of the RDB-data is done with the SCP parameter

```
<TaskControl>
    <RDB ..../>
</TaskControl>
```

- **Runtime Data Bus (RDB-image):** The *(RDB-image)* is used to transfer rendered images of the IG to a given consumer. The interface is based on a TCP port with the TC acting as a server and accepting only a single client.
  The configuration of the RDB-image is done with the SCP parameter

```
<TaskControl>
   <RDB ..../>
</TaskControl>
```

Extended Configuration

Some aspects of the TC configuration are described in the following chapters in more detail. For the others, please refer to the SCP documentation, which is available as an HTML document in your distribution.

- **Image Transfer**: To enable the transfer of images, you have to configure this feature in the IG and the TC. For the IG configuration, see 6.5.11.11. In the TC configuration, set the following parameters:

```
<TaskControl>
   <ImageGenerator imgPortConnect="true" ctrlPortConnect="true"/>
</TaskControl>
```

- **Automatic Activation of Headlights:** If the headlights of the Ego vehicle and the surrounding traffic shall be activated automatically at specific times of day, set the following configuration parameters:

```
<TaskControl>
   <ImageGenerator ..autoHeadlight="true"../>
</TaskControl>
```

- **Input Devices:** The input devices for the direct control of the Ego vehicle are managed via so-called ***Mockups***. The following mockup types are available:

`none`	genuine SIL mode
`Joystick`	throttle, brake and steering via joystick
`Smart`	Smart mockup
`VILSI`	VIL-operation

Other mockup types may be implemented for specific use-cases and are not subject to this general documentation.

For the mockup type **Joystick**, the task control already contains a series of joysticks which it can detect automatically. For other joysticks, the axes and buttons may be configured individually using the SCP parameters.

```
<TaskControl>
         <Joystick>
          <Axis name="steering" .../>
          <Axis name="throttle".../>
          <Axis name="brake".../>
          <Button index="0".../>
         </Joystick>
</TaskControl>
```

For a complete description of this feature, please see [8].

Interactive mockup configuration should always be operated in real-time mode since the human being is not adapted to non-realtime conditions (except for some slower-than-real-time fellows…)

Record/Playback

Upon activation of the record functionality, the TC will record each simulation frame in a user-defined file.

- **Filename**: The name of the record file will be created in one of two ways.
  - Unique Numbers: If the **overwrite** flag is not set for the user-defined file name, the TC will happen a unique three-digit number to the file name, which will be increased with each new record. This allows for the recording of various simulations in a row without modifying the file name in between.

    > **NOTE**
    > If a series of names already exists, then the TC will always choose a filename corresponding to the lowest available number, i.e., it may first fill gaps in the series.

  - User-defined: If the **overwrite** flag is set in the GUI, then the TC will use the user-defined filename for the record file even if a file with the same name already exists.
- **Record**: The data stored in the record file are described in Data Recording (binary) [140].
- **Replay**: When a simulation is replayed, the various states recorded are played back into the system. Figure 32 illustrates which components are provided with simulation data during the replay phase:

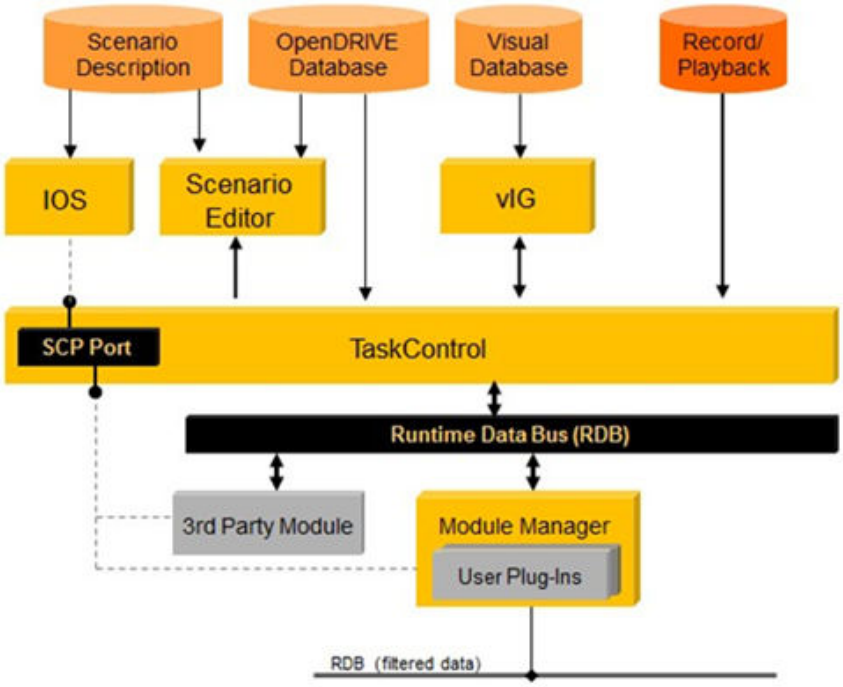


Figure 32. Components during Replay

The following speeds are available for replay: 25%, 50%, 100%, 200%, and 400%. A reduced speed means that the recorded frames are being replayed at a reduced rate, i.e., the time gap between two frames is artificially increased. There will be no interpolation of any values between every two frames.

The replay also provides a means to proceed in single step mode, i.e., the replay pauses after each step. The step width can be 1 to 16 frames.

Image Transfer and/or Video Generation

The image/video generation consists of four components:

- data transfer **Image Generator > TaskControl**
- data transfer **TaskControl > RDB**
- storing images on disk
- conversion of an image sequence into a video

These components may be configured individually, thus providing potential for inconsistent configuration. So, be aware that it's the user's responsibility to perform a configuration in accordance with this documentation, which fulfills the purpose. Pay special attention that a system used to transfer image data at run-time should usually run in the frame-synchronous mode without real-time requirements.

Basic Configuration

To be able to process image information of the IG, the TC must be configured accordingly in its configuration file taskControl.xml (legacy mode) or via the ParamServer settings. The parameters are described in [8].

Images may be made available as standard color information or as grey-scale depth information.

Image Transfer via RDB

To forward images to other consumers, the RDB-image has to be activated. This is also done in the file `taskControl.xml`.

```
<TaskControl>
      <RDB ..imageTransfer="true"../>
</TaskControl>
```

If, as is usually the case, images shall be transferred during run-time (i.e., not only during a replay), then the following parameters must be set:

```
<TaskControl>
     <Video..buffer="color" liveStream="true"../>
</TaskControl>
```

For depth images, the parameters are:

```
<TaskControl>
    <Video..buffer="depth" liveStream="true"../>
</TaskControl>
```

If TC and IG are running on the same computer, their data exchange may also be tunneled via shared memory. In this case, also provide the attribute

```
<TaskControl>
    <Video..streamShm="true"../>
</TaskControl>
```

This does not influence the data transfer method, which you have specified for consumers linked via RDB.

Video Generation from a Replay File

The video generation is triggered via SCP (typically using the GUI). A user-defined replay file will be played frame by frame, and the resulting image sequence will be converted into a video. Before the conversion, the images will be stored at

`/tmp/vidSampleXXXXX.rgb` (XXXXX = sequence number)

Unless otherwise defined by the user, the images will be deleted automatically after the video conversion.

The commands for the video generation are described in 8.2.1 in the command area <Video>.

The name of the video output file may also be defined via SCP. The default name is /tmp/vtdVideo.xxx. The file extension will be automatically assigned in accordance with the selected output format:

AVI uncompressed:	.avi
AVI jpeg compressed:	.mjpg
MPEG4:	.mpg
Archive (single images):	.tar (images are .gz-files)

Important: When converting to AVI format, the requirement that the video's width and height be multiples of 16 must be observed. The TC will scale the images automatically to the best suitable size. However, the height/width ratio may become skewed.

The IG also provides live video recording capabilities. For details, check [9]. The live video recording may also be triggered by pressing the "**t**" key while the focus is on the IG window.

Images at Run-time

To create images at run-time and store them on disk without converting them into a video in a post-processing step, please set the following parameters in the configuration file taskControl.xml:

```
<TaskControl>
   <Video..saveToFile="true" liveStream="true"../>
</TaskControl>
```

The images will be stored at /tmp/vidSampleXXXXX.rgb. They will not be deleted automatically before the start or after the termination of the simulation. Therefore, it's the user's responsibility to perform the necessary data management.

Motion Blur

The motion blur is an enhanced function. The motion blur may be activated in live mode as well as during the video conversion. However, both algorithms differ considerably. The algorithms' common feature is that the blur is generated by adding successive images of the simulation.

If motion blur is activated for **video conversion**, the user may configure how many previous steps of the replay shall be used for the blur. Due to the replay's nature, these steps are discrete, and there is no such thing as a sub-sampling between the steps. In the following example, the last five replay steps will be used for the motion blur during the video conversion:

```
<Video>
  <Control..motionBlur="5"../>
</Video>
```

In **live mode,** the motion blur can be controlled in more detail. The configuration is, as usual, performed in the file taskControl.xml.

Two parameters influence the image generation:

- motionBlurRes: This parameter controls how many individual images are used to compose the resulting image. In the example, the current simulation frame image will be combined with the images of four previous (sub-)steps.
- motionBlurMs: This parameter controls which time period is covered by the images combined into the resulting image. Usually, this period will be smaller than the duration of a simulation frame. In this case, the TC will perform a sub-sampling, i.e., take computer intermediate simulation frames. In the example above, the sub-frames would be computed with a frame time of 2ms. This subsampling is not transparent to external components (e.g., RDB). For them, the simulation time frame remains at the nominal value specified by the basic simulation settings.

Example:

```
<TaskControl>
<Video..motionBlurRes="5" motionBlurMs="10"../>
</TaskControl>
```

If `motionBlurMs` is non-zero while `motionBlurRes` is zero, then `motionBlurRes` will be set to five. Ensure that the ratio of simulation frame time and sub-sampling frame time be an integer number.

Camera Settings

The TC manages the camera settings and transmits the current one to the IG. Camera settings may be custom settings, corresponding to a driver's eyepoint or a sensor's position. Usually, the settings are performed via the GUI and will be stored within the current project.

Camera settings may also be performed via SCP commands. For details, see the SCP documentation for the command area `<Camera>`.

Export Formats

Record files may be exported into various data formats. These are described in the following chapters.

- **CSV File**: The CSV file contains user-readable (i.e., ASCII) data for every player of the simulation. It consists of a header section with general information about the simulation and a data body that contains one line per simulation frame. The file format is described in chapter Data Recording (CSV) [140]
  The CSV file's filename, which corresponds to a given record file, is generated by exchanging the .datextension with .csv.
- **Video Files**: The record files may be converted into video sequences, which may be of one of the following formats:
  - AVI un-compressed
  - AVI jpeg compressed
  - mpeg4
  - archive of single images

  The file formats comply with the available standards and will not be described within this document.

Files and File Formats

Figure 33 shows the files being read and written by the TC.

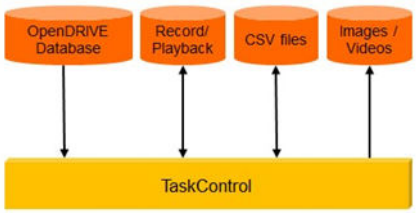


Figure 33. File Dependencies of the TaskControl

Communication

Being the central task, the TC exchanges data with all components of the simulation. These data may be composed of command/control and run-time data.

The TC supports the following communication protocols:

Protocol	TC writes	TC reads	Run-time data	Command data
internal	x	x	x	x
SCP	x	x	-	x
RDB	x	-	x	-

Details of these protocols are given in separate documents (RDB_HTML and SCP_HTML). Some of the chapters only provide a quick overview of the protocols which are not dedicated to the communication with a single component.

Simulation Control Protocol (SCP)

The SCP allows for the transfer of non-real-time, discrete actions and events to the TC and, via the TC (acting as a proxy), to other components. The SCP interface is based on a TCP port with the TC acting as a server.

SCP is a text-based protocol (with XML syntax), and its instruction set may be extended without modifying basic communication mechanisms. The SCP commands which can be interpreted by the TC are described in the separate documentation SCP_HTML. An introduction is given in Simulation Control Protocol (SCP) [131].

Runtime Data Bus (RDB)

The RDB contains the maximum extent of run-time information relevant to other components (e.g., sensor simulation). The RDB consists of two physical connections – one for the run-time data and the other for image data. The connections may be based on TCP, UDP, or loopback ports. They are configured in the file `taskControl.xml` (legacy mode) or via the ParamServer settings. If a connection is TCP, then the TC acts as a server and accepts five clients.

The RDB-image connection may only be realized as a point-to-point TCP-connection with the TC acting as a server. The protocol of the RDB is described in Runtime Data Bus (TC > any) [133] and separate documentation (RDB_HTML). During the replay of a previously recorded simulation session, the RDB will also be served with data so that for data consumers attached to the RDB, it may not be transparent whether the system is in live or replay mode.

Installation/Options

The TaskControl is installed at `Runtime/Core/TaskControl.`

Executable file:		`taskControl`	
Command line arguments:		`-h`	show help
		`-c <configFile>`	load a configuration file

The start parameters of the TaskControl are configured in the following file:

```
Data/Setups/Common/Config/SimServer/simServer.xml
```

Name of the process: `taskControl`

Traffic Simulation

The traffic simulation is closely linked to the Scenario Editor (see chapter Scenario Editor [69]) and is described in more detail in the *ScenarioEditor's User's* Guide.

Traffic Simulation: File Formats

The traffic simulation is configured via the scenario file and the files referenced by the former one.

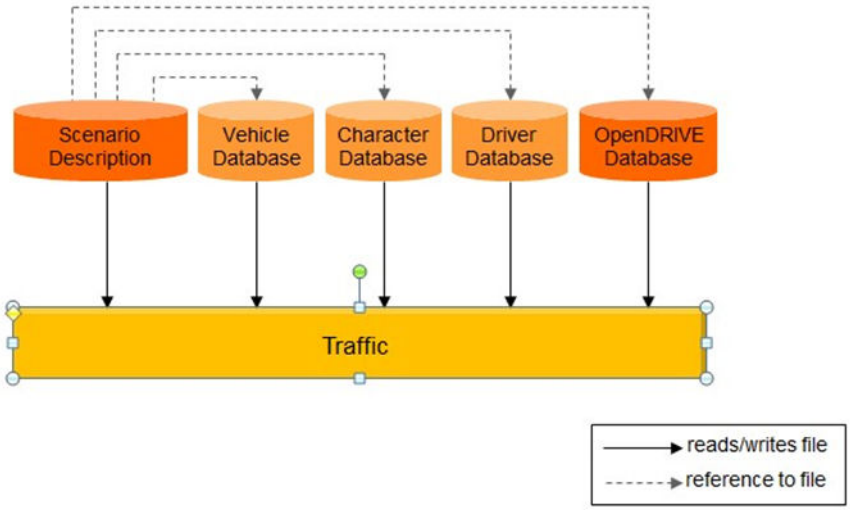


Figure 34. File Dependencies of the Traffic Simulation

Traffic Simulation: Communication

The traffic simulation is only connected to the TC. The connection parameters (ports, protocols, etc.) are defined in the respective configuration files or via environment variables (see Environment Variables [129]).

Figure Linking the Traffic Module to the Simulation [99] provides an overview of the data flows:

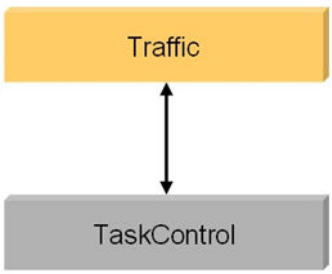


Figure 35. Linking the Traffic Module to the Simulation

Traffic Simulation: Installation

The traffic simulation is installed at `Runtime/Core/Traffic`.

Executable file:	`ghostdriver`

Command line arguments:

```
interface vt    Interface for VTD
lefthand              left-hand traffic
seed <n>     random seed n
h                     help
ped                   activate DI-Guy
```

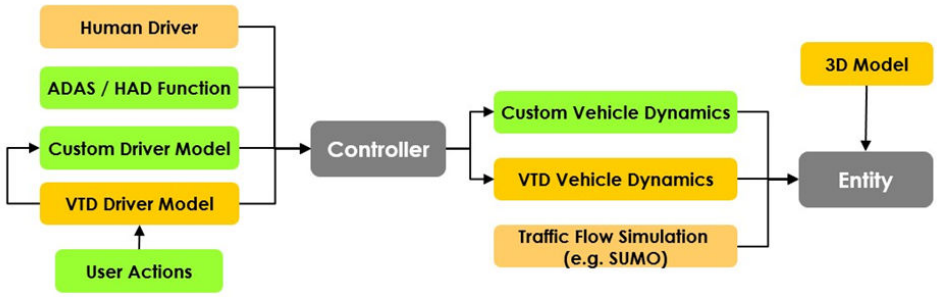
The start parameters of the traffic module are configured in the file

`Data/Setups/Current/Config/SimServer/simServer.xml`

Name of the process: `ghostdriver`

Dynamics and Driver Model of the Ego Vehicle

VTD provides various means to combine different vehicle dynamics simulations and different driver models to control a vehicle.



Two ways for the animation of the Ego have to be distinguished in general:

A. Animation by a dynamics/driver package, which is linked directly via the RDB
B. Animation by a dynamics/driver package, which is a plug-in to the ModuleManager

For mode A, see the RDB documentation; for mode B, see Sound Simulation: Installation [103].

The following sources exist for the vehicle dynamics and the driver model:

- Driver Model:
  - external Driver
  - VTD Driver (from traffic module)
- Vehicle Dynamics
  - external vehicle dynamics
  - VTD Dynamics (single track)

Depending on the operation mode (e.g., interactive, autonomous), the simulation will use

- vehicle dynamics and driver model
- vehicle dynamics only
- none (VIL mode)

If you're operating in VIL mode, you may completely skip this chapter.

The ego vehicle has to be defined in the scenario as an external vehicle.

- **Preparation Mode**: If the simulation is in preparation mode, then the vehicle dynamics is disabled, and the Ego vehicle is directly controlled via a GUI panel. Here, the user may define the speed of the vehicle and change lanes immediately. Any mockup settings of the TC, as well as actions of the scenario which influence the behavior of the Ego vehicle, will be ignored.
- **Standard: Vehicle Mechanics**: In preparation mode, the Ego vehicle reacts immediately according to speed, and lane change commands in the GUI panel. It runs along a path that must be defined in the scenario or uses the current road's centerline to navigate through a database. In the latter case, it cannot traverse any junctions, whereas, in the former, it will be able to do so. There are no physical constraints for the vehicle in this operation mode, so it may even run at Mach 1 on a sharp turn.
- **Extension: VTD-Dynamics:** In preparation mode, one may also use a car of the surrounding traffic as an Ego vehicle. For this, the respective player must be defined in the scenario with

internal animation. Taking over the control of the vehicle is initiated with the SCP command (example)

```
<Player name="FastCar"> <Control master="true"/> </Player>
```

This vehicle will run with our dynamics, i.e., it will perform continuous speed and lane change operations if these are initiated via the preparation panel.

- **Operation Mode**: The operation mode is the intended mode of the simulation. In this mode, the input to the Ego vehicle is performed according to the TC's mockup settings, and it will follow the actions defined in the scenario.

Sound Simulation

The sound simulation is an add-on to the standard installation and may not be present in all installations. The sound module simulates the sound of the own vehicle (ego) and—depending on the version—of the surrounding traffic.

Sound Simulation: File Formats

The sound simulation is configured via the sound configuration file. The actual sound samples are referenced from within the configuration file

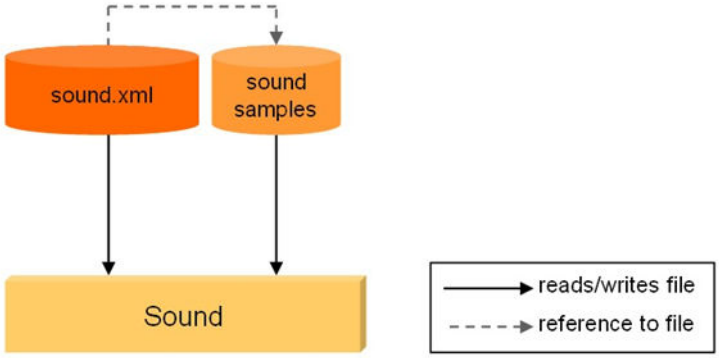


Figure 36. File Dependencies of the Sound Simulation

Sound Simulation: Communication

The sound simulation is only connected to the TC. The connection parameters (ports, protocols, etc.) are defined in the respective configuration files and/or via environment variables (see Environment Variables [129]). Below figure provides an overview of the data flows:

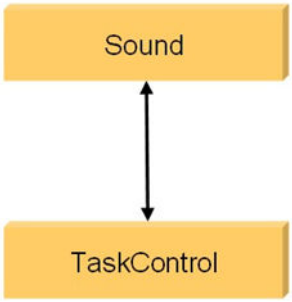


Figure 37. Linking the Sound Module to the Simulation

Sound Simulation: Installation

The traffic sound is installed at `Runtime/AddOns/Sound.3.2`.

Executable file:	`vroom`	
Command line arguments:	`-f<configFile>`	configuration file
	`-h`	show help

`-n <level>`	notification level
	-1 (always), 0 (fatal)...5(internal)

The sound samples can be found within the sound module's installation directory. The start parameters of the sound module are configured in the file `Data/Setups/Current/Config/SimServer/simServer.xml`.

Name of the process: `vroom`

The module manager provides a means to run (custom) plug-ins within the VTD environment. With the module manager, users don't have to take explicit care of network communication, etc.

Two sorts of plugins are available:

- sensor plugins
- dynamics plugins

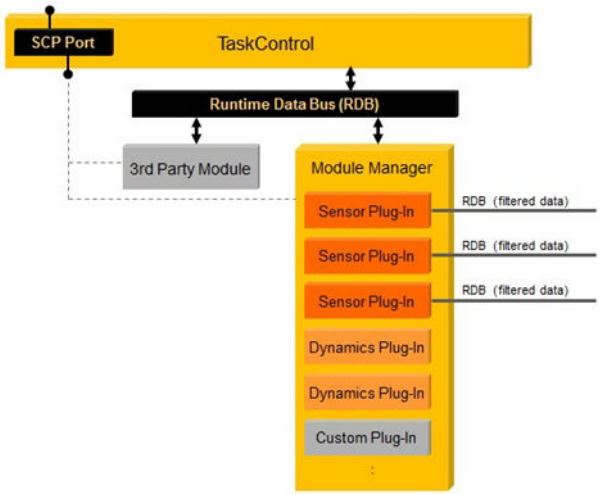


Figure 38. ModuleManager (overview)

Sensor plugins are used for the processing (e.g., filtering) of the simulated environment. The results may be used as inputs for the interpretation in algorithms of active safety and assistance systems. One key feature of the sensor models is filtering all available data into a reduced stream of relevant data within certain spatial constraints (see the following figure), which can be forwarded to other components.

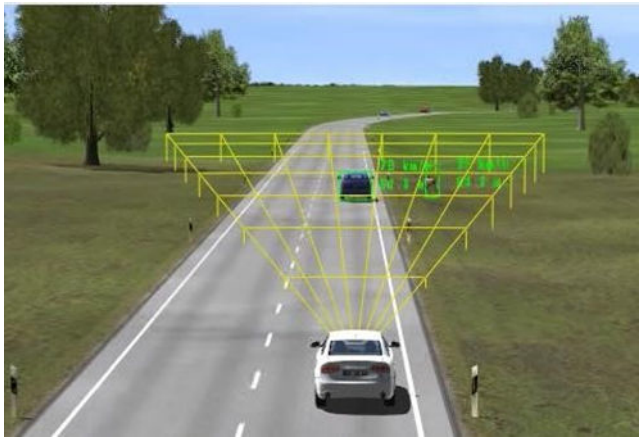


Figure 39. Perfect Sensor, extraction of environment information

Dynamics plugins are used to simulate a vehicle's dynamic behavior according to driver inputs (brake, throttle, steering).

Sound Simulation: Implementation

The implementation provided here allows for the concurrent operation of multiple plugins during runtime. Besides, a software development kit (SDK) is provided so that the user may develop his own sensor and dynamics models and plug them into VTD.

All modules are running within the so-called ModuleManager which also handles the data streams via RDB and SCP. The communication between the ModuleManager and the module plug-ins is performed via internal data structures. Sensor plug-ins may either provide their filtered data via discrete RDB-compliant interfaces to 3rd party modules or may only feedback their detection lists directly into the TC.

Dynamics plug-ins don't provide additional output interfaces; instead, there's direct feedback of the results into the TC. One or more ModuleManagers may be run concurrently, each with various plug-ins.

The ModuleManager runs at a user-configurable speed (default: 100Hz). The update routines of the module plug-ins are called in each frame.

Sound Simulation: Configuration

The basic configuration of the manager itself and the configuration of sensor plug-ins and dynamics plug-ins, may be performed via the file moduleManager.xml.

This file is located at one of the following locations:

- `Data/Distros/Current/Config/ModuleManager`
- `Data/Projects/Current/Config/ModuleManager`

The structure of the configuration file corresponds to the SCP syntax, which is described in the SCP_HTML document (separate document, see the command areas `<Sensor>` and `<DynamicsPlugin>`).

Example: The basic (default) configuration is shown in the following example:

```
<!--############################################################
#
# configuration file for modules, 15.08.2011 by M. Dupuis
#
#
#################################################################
-->
```

```
<RDB>
    <Port name="RDBraw" number="48190" type="TCP" />
</RDB>
<Debug    enable="false"
     lightSource="false" />
```

The RDB port (`RDBraw`) and communication type (`TCP`) have to be defined so that the manager knows how to connect to the TC. In case of a UDP connection, the manager will automatically open a feedback channel on a separate, pre-configured port number. Some debug options may be set to check various features of the ModuleManager or to assist during trouble-shooting.

Sensor Plug-ins

Sensor plug-ins may be configured via the GUI or via the configuration file (see above). In the configuration file, each sensor is configured within a separate section `<Sensor>`.

The following example provides a configuration with two sensors, each using the perfect sensor plug-in but differing considerably in terms of the parameterization. Also, note that the first sensor is labeled as

persistent (in the `<Load>` command) indicating that it will always remain within the ModuleManager even if a `<UnloadSensors>` command is received.

```
<Sensor name="perfect" type="video>
  <Load    lib="libModulePerfectSensor.so" path=" " " persistent=”true”/>
  <Frustum near="0.0" far="200.0" left="10.0" right="10.0" bottom="5.0"
    top="5.0" />
    <Cull     maxObjects="5" enable="true" />
    <Port     name="RDBout" number="48185" type="TCP" sendEgo="true" />
    <Position dx="2.5" dy="0.0" dz="0.5" dhDeg="0.0" dpDeg="0.0"
     drDeg="0.0" />
    <Filter   objectType="pedestrian"/>
    <Filter   objectType="vehicle"/>
    <Debug    enable="true"
detection="false"
road="false"
position="false"
dimensions="false"
camera="false"
CSV="false"
packages="true"
culling="false"
contactPoints="false"/>
</Sensor>
<Sensor name="perfect2" type="video">
        <Load lib=" libModulePerfectSensor.so" path=""/>
        <Cull     maxObjects="5" enable="false" />
        <Port     name=" RDBout" number="48186" sendEgo="true" />
        <Filter   objectType="pedestrian"/>
        <Filter   objectType="vehicle"/>
<Debug    enable="false"
detection="true"
road="false"
position="false"
dimensions="false"
camera="false"
CSV="false"/>
</Sensor>
```

The user within the interface class can set the outgoing sensor data (list of detected objects). This data is a subset of the incoming data and also corresponds to it in terms of structure. The outgoing data may be sent via a user-defined port, which also serves the RDB protocol (so-called 2nd-degree RDB).

For each sensor, the output port may be set in the configuration file individually.

Types of Sensors

- **Perfect Sensor**: A perfect sensor is provided with the VTD package as an example of the implementation of custom sensors. This sensor may be parameterized individually (see above)

and may, thus, represent various sensor types. The parameterization may be performed in the ModuleManager's configuration file (see above) or – to an almost complete extent – via the GUI.

- **Custom Sensors**: By means of the sensor plug-in SDK, the user may develop custom sensors. These may be loaded and operated by the ModuleManager. The SDK is located in the directory Develop/Modules/ under the VTD home directory. Its structure is described in Develop Directory.
  To provide an example for implementing their own sensors, the code, and compiler environment for the perfect sensor are given in the sub-directory PerfectSensor.
  Custom sensors must be derived from the class Module::SensorPlugin. The user must implement at least the method update(). The ModuleManager calls this routine in each simulation frame with the current frame number and a pointer to the interface class, which contains all data that has been received from the TC.

Before calling the update() method, the base routines of the SensorPlugin will have performed the following steps:

- receiving of RDB data
- object filtering (by type)
- object detection (by frustum)

The following code fragment shows the main elements of the base class's update() -method and shall provide further insight into the methods/algorithms applied. The complete source code is not disclosed since it contains elements that are proprietary to us.

```
int
SensorPlugin::update( const unsigned long & frameNo, Framework::Iface* data )
{
// allocate data structure for output data
if ( !mSensorOutData )
{
         std::string name = mName + std::string( "_OutData" );
mSensorOutData   = new SensorIface( name );

// set the reference in the base class         setOutDataRef( mSensorOutData );
}

// has the simulation been reset?
bool resetDetected = mSensorInData->mFrameNo < mFrameNo;

mFrameNo = mSensorInData->mFrameNo;

// map player name to ID?
if ( ( mOwnPlayerNameDefined || mUseDefaultPlayer ) && ( mOwnPlayerId < 0 ) )
{
int plId = -1;

if ( mUseDefaultPlayer )
plId = mSensorInData->getDefaultPlayerId();
else
plId = mSensorInData->getPlayerId( mOwnPlayerName );

if ( plId >= 0 )
{
setPlayer( ( unsigned int ) plId );
resetDetected = true;
}
}
// now set the reference point from the given data
if ( mNewFrame )
{
// set the basic output data
setBasicOutData( mSensorInData );

// get the player carrying the sensor
RDB_OBJECT_STATE_t* ownObject = 0;

if ( mOwnPlayerId >= 0 )
ownObject = mSensorInData->getPlayerObject( mOwnPlayerId, extended );
```

```
else if ( !mOwnPlayerName.empty() )
ownObject = mSensorInData->getPlayerObject( mOwnPlayerName, extended );

if ( ownObject )
{
mOwnPlayerId = ownObject->base.id;

setReferencePos( ownObject->base.pos.x, ownObject->base.pos.y,
ownObject->base.pos.z,
ownObject->base.pos.h, ownObject->base.pos.p,         ownObject->base.pos.r );

// set the USKs origin from the given data
setUSKOrigin( ownObject->base.pos.x, ownObject->base.pos.y,  ownObject->base.pos.z,
ownObject->base.pos.h, ownObject->base.pos.p,
ownObject->base.pos.r );

// calculate the actual position of the sensor
calcPos();

// filter objects by type
filterObjects();

// cull the objects and get resulting object list in interface
cull();

// calc Object USK pos
calcUSKPos( ownObject );

// add object info relative to sensor
addSensorInfo();
}
}

// call the update routine of the derived class!
int retCode = update( frameNo, mSensorInData );

if ( mNewFrame )
{
// if feedback is enabled, send the feedback data
sendDetectionLists( mSensorOutData );

// add own data
if ( mRDBoutSendEgo )
addEgoInfo( mSensorOutData, mSensorInData );

// if RDB out interface is defined, then send the data
sendRDB( mSensorOutData );
}
return retCode;
}
```

The highlighted line indicates the place where the `update()` -method of a custom implementation will be called. After this, the routine `sendRDB()` will send the detected objects via the RDB-out channel (2nd level RDB). File structure and contents correspond to the RDB that is established between TC and ModuleManager. By this, it is also possible to chain-link multiple instances of the ModuleManager.

By calling the method `setFeedback()` at the end of the `update()` -method, detected objects will be registered in the interface structure and will be sent from the ModuleManager to the TC. There, they will be used, e.g., for the visualization of detection information (bounding frames). The feedback will only be sent if it has previously been activated (either in the configuration file or by using the method `SensorPlugin::setFeedback()`).

Dynamics Plug-ins

Dynamics plug-ins may be configured in the configuration file moduleManager.xml (see above). In this file, each dynamics plugin is configured within a separate section <DynamicsPlugin>. The following example provides a configuration with two plug-ins, each being based on the dynamics, which is also used for the traffic vehicles but assigned to different vehicles. Note that in the example, the first plug-in

is assigned to the default player, i.e., the first external player of the scenario, whereas the second plug-in is assigned to the player named “Me2”.

```
<DynamicsPlugin name="vi">
<Load    lib="libModuleTrafficDyn.so" path=""/>
<Player    default="true" />
<Debug    enable="false"
dynInput="true"
dynOutput="true"
CSV="false"
packages="false"/>
</DynamicsPlugin>

<DynamicsPlugin name="vi2">
     <Load     lib="libModuleTrafficDyn.so" path=""/>
<Player  name="Me2" />
<Debug    enable="false"
dynInput="true"
dynOutput="true"
CSV="false"
packages="false"/>
</DynamicsPlugin>
```

The user within the interface class can set the outgoing data of a dynamics plug-in (new state of the vehicle). This data is a subset of the incoming data, and its structure also corresponds to that data. The outgoing data is sent via the RDB data connection to the TC.

Types of Dynamics Plug-ins

- **Traffic Dynamics:** The dynamics, which are also used in the traffic module, are provided with the VTD package as an example of the implementation of custom vehicle dynamics. This dynamics module may be assigned to a specific player or the first external player (so-called “default” player). The player’s name and some debug parameters can only be set in the ModuleManagar’s configuration file (see above).

- **Custom Plug-Ins:** By means of the dynamics plug-in SDK, the user may develop custom vehicle dynamics. These may be loaded and operated by the ModuleManager.
  The SDK is located in the directory `Develop/Modules/` under the VTD home directory. Its structure is described in Develop Directory [49].
  To provide an example for implementing own dynamics modules, the code and compiler environment for a very simple sample dynamics are given in the sub-directory `SampleDyn`.
  Custom dynamics modules must be derived from the class `Module::DynamicsPlugin`. The user must implement at least the method update(). The ModuleManager calls this routine in each simulation frame with the current frame number and a pointer to the interface class, which contains all data that has been received from the TC.

Files and File Formats

The ModuleManager is embedded into the simulation via the files which are shown in below figure.

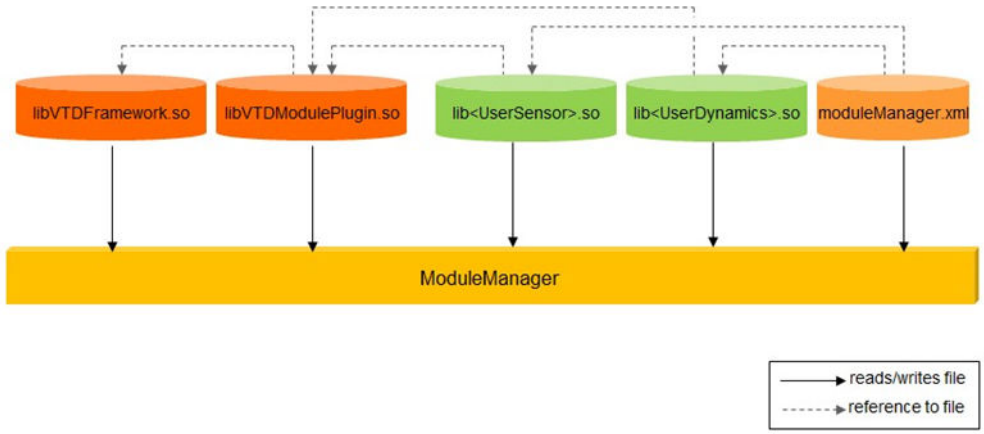


Figure 40. File Dependencies of the ModuleManager

Communication

All module plug-ins receive their data via RDB and provide their output data depending on their implementation (usually with feedback via the RDB or establishing a 2nd level RDB connection). Below figure shows the connection of the ModuleManager to the simulation.

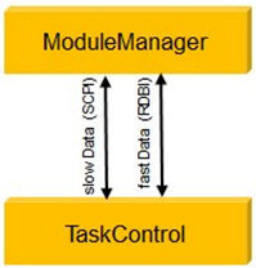


Figure 41. Communication of the ModuleManager

Installation/Options

The ModuleManager is located in the directory `Runtime/Core/ModuleManager`. The module plugins are usually located in one of the following directories:

- `Data/Projects/Current/Plugins/ModuleManager`
- `Data/Distros/Current/Plugins/ModuleManager`

The module framework libraries are located in `Runtime/Core/ModuleManager/lib`.

Executable file:	`moduleManager`	
Command line arguments:	`-i<interface>`	name of the network interface
	`-f <filename>`	`configuration file`
	`-t <frameTime>`	`frame time in [s]`
Environment variables:	see Environment Variables [129]	

Start Procedure

The start parameters of the ModuleManager are configured in the file

`Data/Setups/Current/Config/SimServer/simServer.xml`

Name of the process: `moduleManager`

SCP-Generator

NOTE
This tool is a internal test tool and is provided "as is".

The scpGenerator provides a means to send messages according to SCP protocol into the simulation and to monitor outgoing messages. Messages may be fed into the generator via the command line or—more comfortable—by using data files. Data files that are to be interpreted by the scpGenerator must comply with the rules defined in [8].

Typically, the scpGenerator runs at a frequency of around 60Hz. This should be taken into account when composing commands which are time-triggered by the generator's frame count.

The scpGenerator is installed in `Runtime/Tools/ScpGenerator`. Sample data files (scripts) can be found in the standard test project under `Scripts/ScpGenerator`.

Name of executable:	`scpGenerator`	
Command line parameters:	- c <filename>	data file (script)
	`- i <cmd>`	`command for immediate execution`
	`- s`	`open with TCP server port (usually it opens a client port)`
	`- S <address>`	`address of the server when running as client`
	`- m`	`run as monitor`
	`- H`	`show all messages in full length`
	`- l`	`loop execution of the data file`
	`- g`	`connect to RDB port`
	`- d`	`enable debug mode`
	`- f <string>`	`in monitoring mode, show only commands containing "string"`
	`- F <string>`	`in monitoring mode, show only commands NOT containing "string"`
Environment variables:	`PORT_SCP`	ID of the SCP port
	SERVER_SCP	Name of the TCP server port's host
Return values:	-1	error
	1	normal termination

Example:

Run generator with a script file:

```
scpGenerator –c myscript.scp
```

Run generator with a command line

```
scpGenerator –i '<Symbol name="mySym"><Hide/></Symbol>'
```

NOTE
You have to embed the entire command in single quotes

RDB-Sniffer

NOTE
This tool is a internal test tool and is provided as is.

The RDBSniffer provides a means to monitor / record/replay RDB data streams. It may connect as a client to a TCP server or as UDP member to messages broadcast by a sender. The RDBSniffer is installed in `Runtime/Tools/RDBSniffer`.

Name of executable: `rdbSniffer`

Command line parameters:

```
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]
```

`-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>

Databases

Apart from the custom databases which may be built using the Road Designer ROD, the two following databases are available for VTD:

SmartDB
- 2- and 3-lane motorway
- rural roads
- city (including crossings with traffic lights)
- installed at: `Data/Projects/Current/Databases/SmartDB`

Town
- rural road, ending in a small town (signs only, no traffic lights)
- installed at: `Data/Projects/Current/Databases/Town`

The town database is considered a test database for VTD and may be used in each application. The SmartDB database must be licensed individually before use.

Road Designer (ROD)

The Road Designer (ROD) is usually used in the design phase of a simulation project. ROD's functionality is described in separate documentation, which is located at `Runtime/Tools/ROD/Doc`. Only extensions and modifications relevant to Virtual Test Drive are described in this document.

ROD is the main tool for creating virtual road networks. With ROD, the user generates graphical (3D) data and logical data. The following output formats are used:

- graphics: `OpenFlight / IVE / OSGB`
- logics: `OpenDRIVE`

Extensions

- **Accuracy of Road Tesselation**: In the logical database, roads and their properties are described with high accuracy in engineering elements (lines, spirals, etc.). For the graphics, roads have to be tessellated to be available for rendering. Therefore, the graphics data will always be of less accuracy than the logical data.
  You may influence the accuracy of the graphical representation by editing the following entry in the configuration file `TT_SETUP.DAT`:

  ```
  TED_EXPORT_ANGULAR     ang max min
  ```

  `ang` : maximum angular discrepancy [deg] between two successive road polygons in any spatial direction (i.e., the maximum value emerging either from direction elevation or superelevation). Typically 1.5 deg.

  `max` : maximum length of a road polygon [m]

  `min` : minimum length of a road polygon [m]

- **Data Import:** Additional formats for importing data according to customer-specific ASCII formats have been implemented. These may be mainly used for importing road reference lines and provide the basis for further extension of the data with lane and environment information. Additional data import formats are described in chapter Road Description [138].

- **Vehicle-in-the-Loop (VIL):** For VIL applications, ROD may be equipped with special model and configuration files. It is the user's responsibility to perform the corresponding configuration. We support this process by providing a basic set of simplified configuration files on request.

ROD Operation

In an extension of the standard documentation, the following information concerning the operation of ROD is provided.

Preview Tool

The preview tool is based on OpenSceneGraph (osgviewer). After the generation of a database, it is typically invoked automatically. Alternatively, it may be launched by pressing the **Preview** button.

To stop the preview, close the corresponding window or press **Esc**.

The preview tool may be operated as follows:

```
Keyboard and Mouse Bindings:
1.              Select 'Trackball' camera manipulator (default)
2.              Select 'Flight' camera manipulator
```

3.	Select 'Drive' camera manipulator
4.	Select 'Terrain' camera manipulator
5.	Select 'UFO' camera manipulator
Drive: Down	Cursor down key to look downwards
Drive: Space	Reset the viewing position to home
Drive: Up	Cursor up key to look upwards
Drive: a	Use mouse middle, right mouse buttons for speed
Drive: q	Use mouse y for controlling the speed
Escape	Exit the application
Flight: Space	Reset the viewing position to home
Flight: a	No yaw when banked
Flight: q	Automatically yaw when banked (default)
O PrtSrn	Write camera images to "saved_image*.jpg"
Trackball: +	When in stereo, increase the fusion distance
Trackball: -	When in stereo, reduce the fusion distance
Trackball: Space	Reset the viewing position to home
UFO:	Please see: http://www.openscenegraph.org/html/UFOCameraManipulator.html
UFO: H	Reset the viewing position to home
Z	If recording camera path stop recording camera path, save to "saved_animation.path" Then start viewing from being on animation path
b	Toggle backface culling
f	Toggle fullscreen
h	Display help
l	Toggle lighting
o	Write scene graph to "saved_model.osg"
s	Toggle instrumention
t	Toggle texturing
v	Toggle block and vsync
w	Toggle polygon fill mode between fill, line (wire frame) and points
z	Start recording camera path

ROD File Formats

Figure 45 illustrates the file dependencies of ROD.

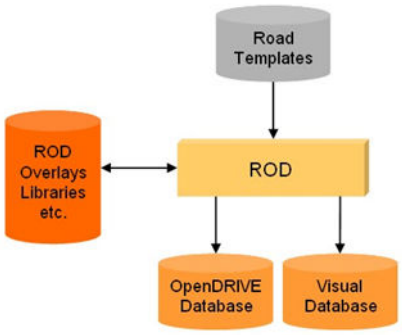
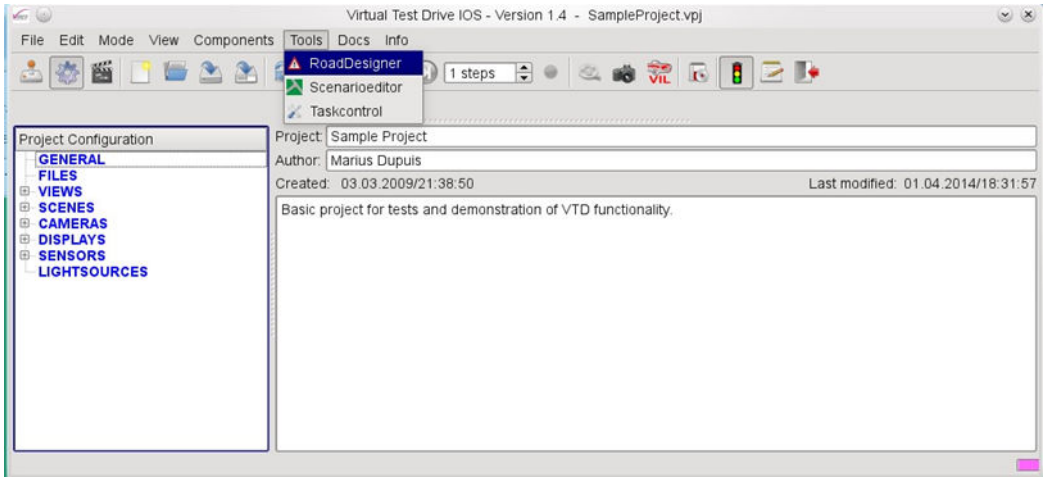


Figure 42. File Dependencies of ROD

ROD will also create a repository of its user-configurable settings and resources in your `.config` directory. ROD is installed as a **Tool** component simply by unpacking the respective archive (see Installation).

Starting ROD

From the VTD GUI, ROD may be started via **Tools > RoadDesigner.**



This invokes the following script:

Directory:	`Data/Setups/Current/Bin`
Name:	`startTaskRoadDesigner`

ROD may only be used offline, i.e., it provides no link to the running simulation.

Interfaces

Data Flows

This chapter provides an overview of the data flows disclosed and which may be of interest for a better understanding of the mechanisms within VTD. These data flows contain configuration and run-time data and may be realized via a network interface or shared memory.

The following definitions are valid for connection pairs, i.e., for both directions of a communication between two components. For genuine uni-directional connections, the definitions will be provided from the sender's point of view. Symbolic names are used for specific data packages. Internal data flows are not disclosed.

- Road Designer → Road Library
- Road Designer → ScnearioEditor
- Operator Station → TaskControl
- TaskControl → Runtime Data Bus
- TaskControl → Runtime Data Bus

These data flows are described in the following sections.

Road Designer → Road Library

content	phase	format / protocol	frequency
road layout	prepare	.tdo (proprietary)	upon load / save

Road Designer → ScnearioEditor

content	phase	format / protocol	frequency
road layout	prepare	OpenDRIVE®	upon load

The OpenDRIVE format is described in a separate document (see http://www.opendrive.org)

Operator Station → TaskControl

connection: TCP (TC is server)

content	phase	format / protocol	frequency
control / status commands	any	SCP protocol	event

TaskControl → Runtime Data Bus

connection: UDP broadcast, TCP (preferred) or loopback

TaskControl → RDB

content	phase	format / protocol	frequency
run-time simulation data	run	see 8.2.2	60 Hz

The RDB protocol consists of various packages that describe the current state of the relevant players of the simulation, of the environment, etc. It is the preferred interface for 3rd party applications that want to interact with the simulation during run-time. Other interfaces are not disclosed or not under the same high level of maintenance and quality control.

The following images give a brief overview of the communication structure:

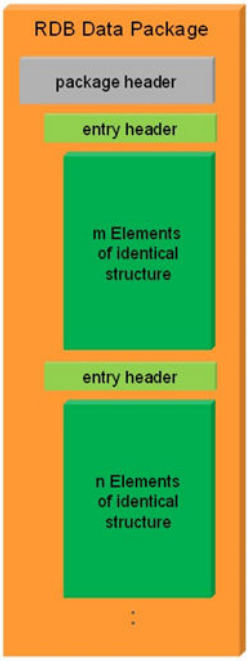


Figure 43. Structure of RDB communication

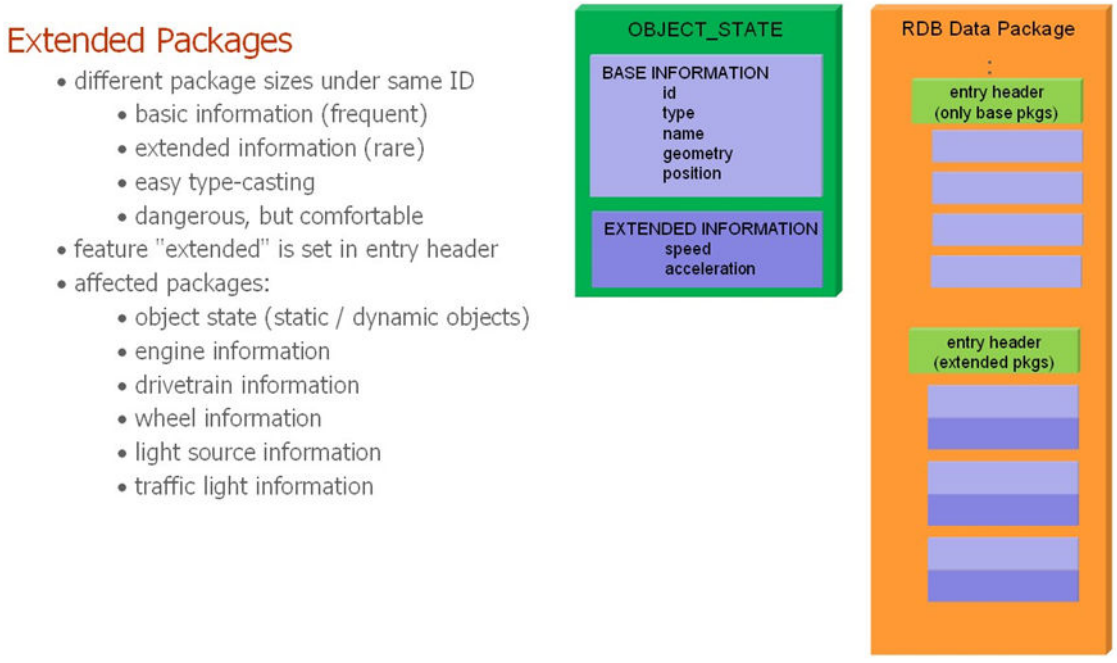


Figure 44. Extension Mechanism for RDB Packages

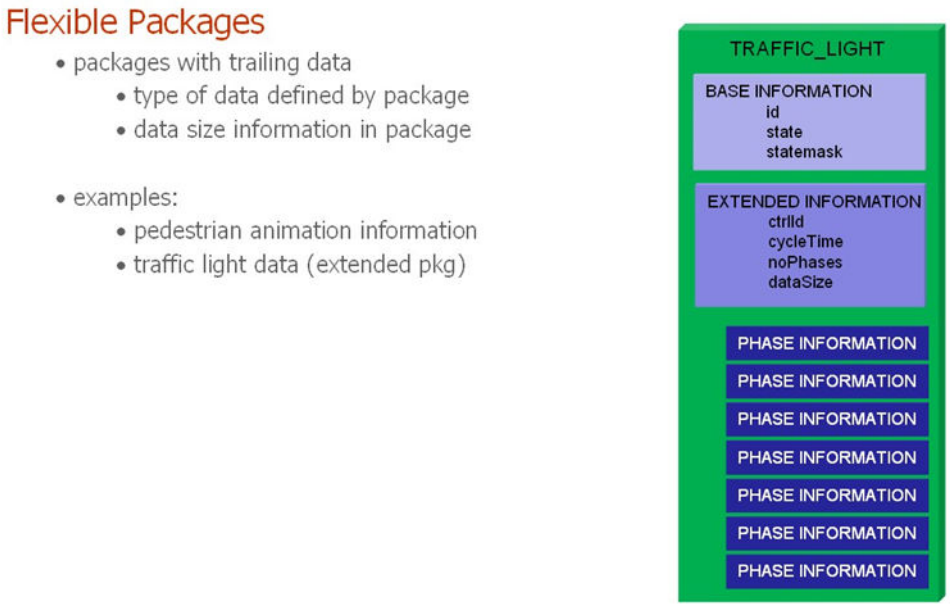


Figure 45. Variable Length of RDB Packages

TaskControl → Runtime Data Bus

connection: Shared Memory

The following figures provide an overview of the possibilities to interconnect with the taskControl using RDB and a shared memory (SHM) interface. For the SHM connection between TC and IG, which is also shown in the figure, please see the next chapter.

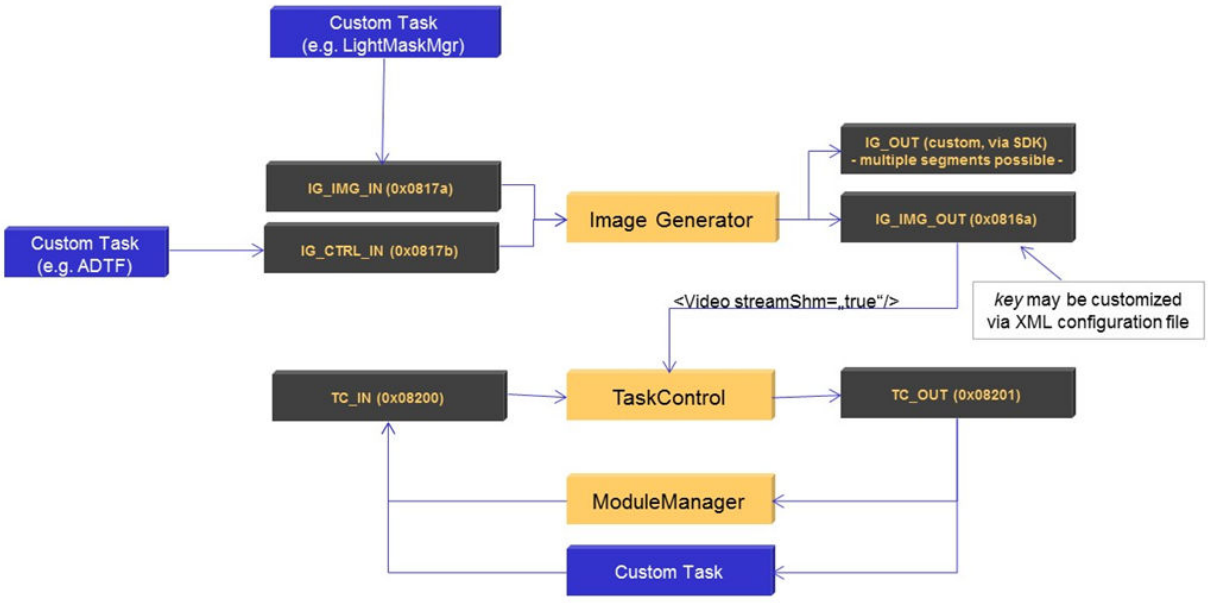


Figure 46. Overview of SHM connections with the Task Control

General Layout

The shared memory allows for a direct connection between two components using data access pointers instead of network communication mechanisms. The SHM contains RDB messages in the same

format that would be expected via the network. In addition, however, some management structures are necessary to let all subscribers of a SHM know where the data can be found (see next figure).

The memory starts with a general header of type `RDB_SHM_HDR_t`. This header contains the information about the number of buffers managed in the shared memory (multiple buffers may be used for concurrent read/write operations). Each buffer itself carries some administrative information in a structure of type `RDB_SHM_BUFFER_INFO_t`. The information blocks for all buffers are located right after the general header (see figure). The buffers' actual data is located after the last buffer info entry (i.e., after the complete SHM Header). As noted above, the structure within these data blocks complies with the same structure that is found in RDB communication via a network.

An SHM can be operated in single or double buffer mode. In *single buffer* mode, new data can only be written by the producer after the consumer has confirmed receipt (or is working completely in asynchronous mode and does not depend on the consistency of an SHM segment). In *double buffer* mode, one buffer can be written while the consumers are processing another one.

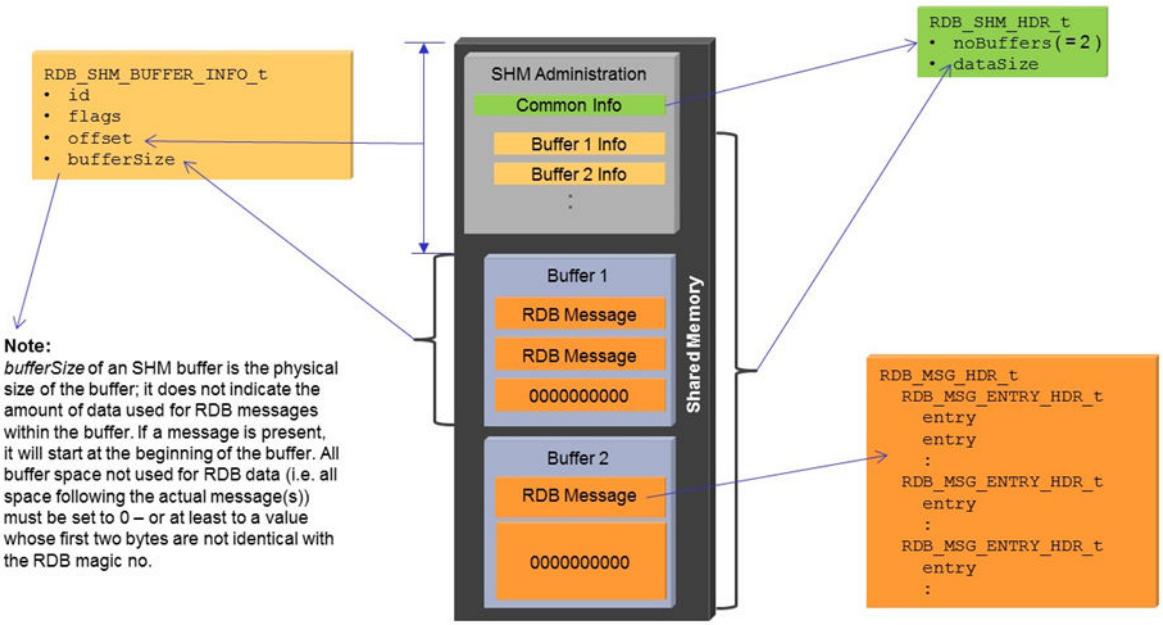


Figure 47. General buffer layout in the shared memory (here: double buffer)

Shared memory output of the Task Control

If a component wants to synchronize with the transmitting SHM of the TC, it should first register as SHM sync source so that the TC waits with any subsequent writing operation to the SHM until the consumer has read all data. The following figure illustrates this mechanism.

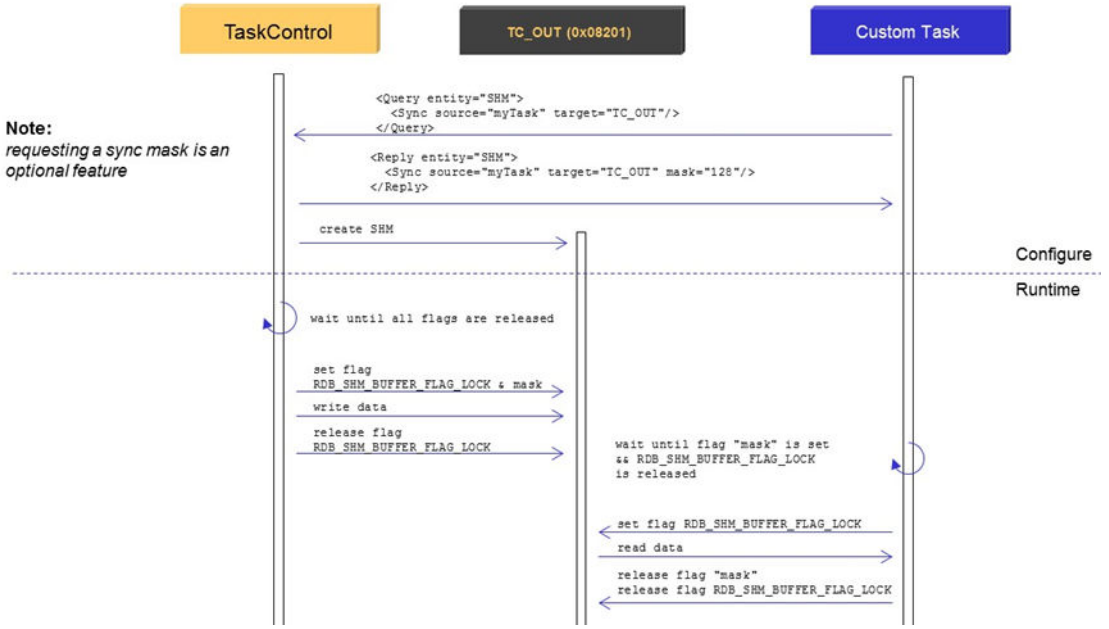


Figure 48. Synchronization with TC output via shared memory

Shared memory input into the Task Control

The Task Control may also act as a consumer of RDB data via SHM. In this case (as for the time being), only one producer is allowed to write to the TC's SHM in a given simulation setup. Therefore, users must not setup their custom task and, e.g., the moduleManager, to write concurrently to the SHM.

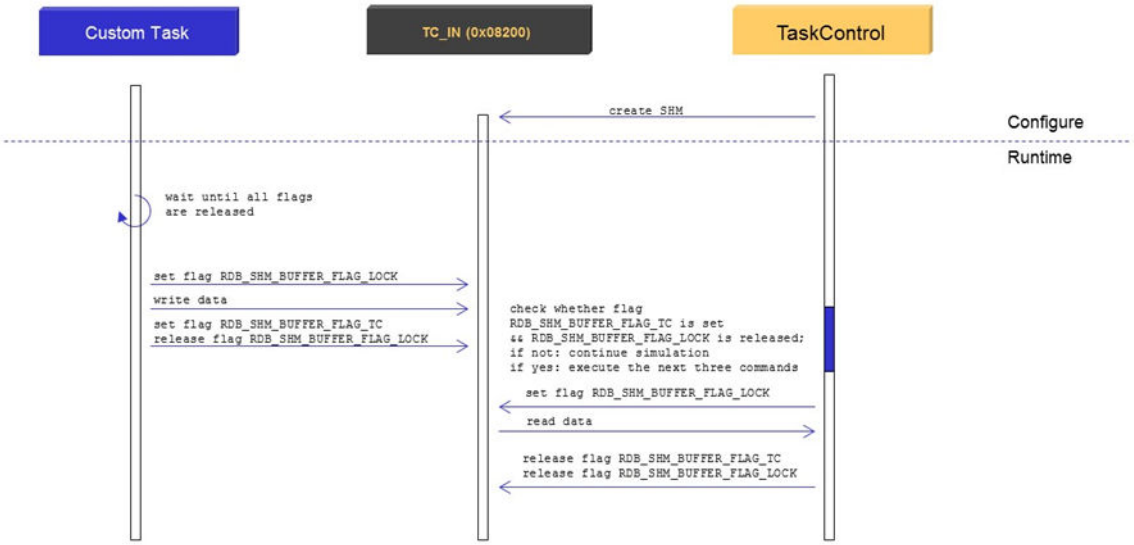


Figure 49. Synchronization with TC input via shared memory

ImageGenerator ↔ Runtime Data Bus

connection:↔ Shared Memory

IG ↔ RDB

Content	phase	format / protocol	frequency
image data (OUT) light distributions (IN)	run	see 8.2.2	60 Hz

The direct interface to the image generator via shared memory provides a means to extract image data from the IG or feed light distributions into the IG with minimum latency.

General Layout

The general layout is identical to the one depicted in the previous chapter. For the IG, two separate shared memory segments are used for input and output. The IG will create the output segment; the input segment has to be opened by the component which wants to send data into the IG. The following IDs apply:

Output from IG:	`RDB_SHM_ID_IMG_GENERATOR_OUT`
Input into IG (data):	`RDB_SHM_ID_IMG_GENERATOR_IN`
Input into IG (control):	`RDB_SHM_ID_CONTROL_GENERATOR_IN`

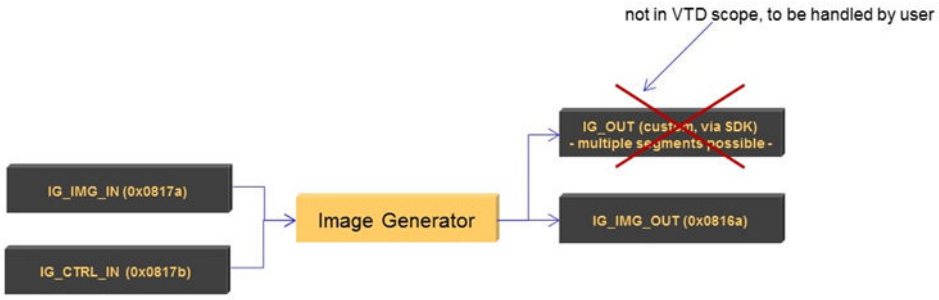


Figure 50. Shared Memory Connections of the Image Generator

Extraction of Image Data

Image data may be extracted from the IG via SHM in double-buffer mode. This means that the user may process one buffer while the other one is being filled by the IG with the next image. Depending on the settings, the IG will wait for a buffer to be free for writing or will just write its image data to alternating buffers.

To enable image output via Shm, the configuration file must be modified as follows:

A. IG Configuration

File:

`Data/Setups/Current/Config/ImageGenerator/IGbase.xml`

Entries:

1. Load the corresponding plugin:

```
<Plugins>
    :
    <Plugin file="RDBInterface"/>
    :
</Plugins>
```

2. Configure the plugin:

```
<Components>
     :
               <RDBInterface name="MyRDBInterface"
                    printDebugInfo="0"
                    ignoreShmFlags="1"/>
     :
</Components>
```

By setting ignoreShmFlags to "1", a new image is written each frame to the shared memory without waiting for any consumers. Otherwise (i.e., if set to zero), a new image will only be written to a shared memory buffer if the buffer is not locked (i.e., the buffer flag `RDB_SHM_BUFFER_FLAG_LOCK` is not set, and the TC bit `RDB_SHM_BUFFER_FLAG_TC` is cleared).

NOTE
The shared memory may be double buffered, i.e., always check the buffer flags first so that no data is read from locked buffers.

B. TaskControl Configuration

File:

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

Entries:

Enable video streaming via SHM:

```
<Video
       :
       liveStream="true"
       streamShm="true"
       :
  />
```

C. Synchronization

Reading data from the IG's shared memory may be done in anonymous mode (i.e., the IG is not aware in advance of a reader) or in synchronized mode.

In *anonymous mode*, the user may only use the flag `RDB_SHM_BUFFER_FLAG_LOCKf` to prevent the IG from altering data while a user's task is processing contents. If the flag is not set, the IG will proceed and not wait for the user.

In *synchronized mode*, the IG knows in advance which flags have to be set or released so that it is allowed to write a new data frame to the SHM. These flags may either be set explicitly in the IG's configuration file (see next figure), or they may be requested via SCP from the Task Control (see subsequent figure).

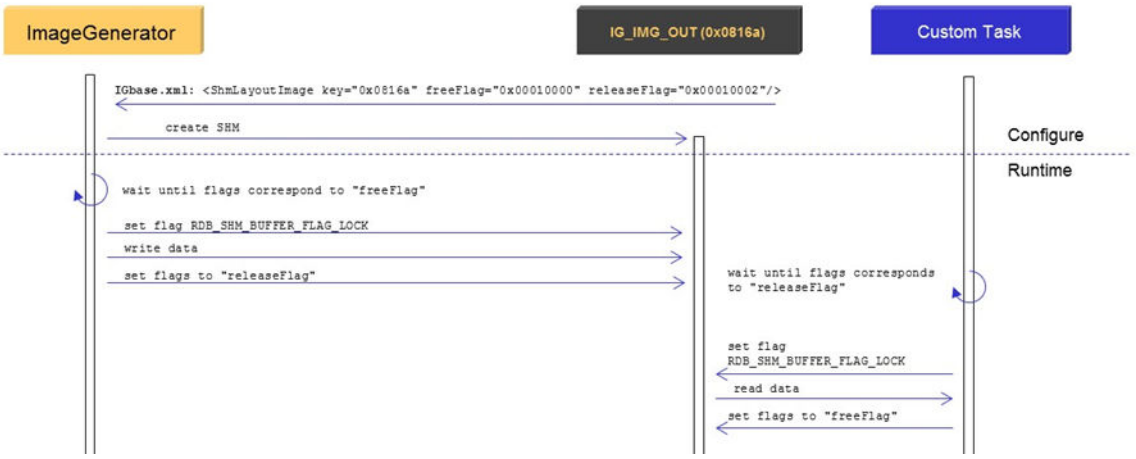


Figure 51. Synchronization of SHM Access after Configuration in IG Configuration File

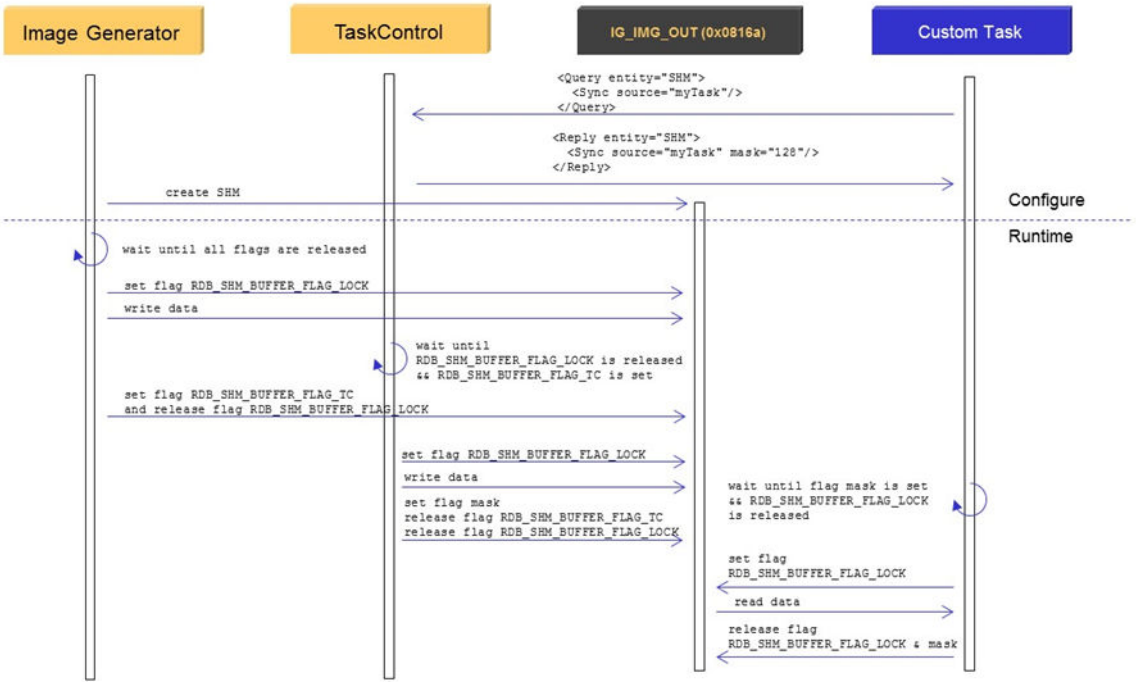


Figure 52. Synchronization of SHM Access after Configuration via SCP

D. Example for Reading the SHM

The following code fragment (from TaskControl) shall provide a hint of how to read the shared memory information from the image generator by first checking which buffer is available (i.e., unlocked) and then reading the actual information. Please note that this code fragment is an excerpt of an actual code and, therefore, contains variables and methods that are not available in a specific user's environment but have to be implemented by the user.

```
bool
IfaceIG::checkForImgData()
{
// attach to shared memory first?
if ( !mImgShm )
{
        // attach to shared memory
```

```
        mImgShm = new Framework::ShdMem( Framework::ShdMem::sKeyImg, 0, false );

                if ( !mImgShm )
return false;
    }
        // get a pointer to the shm info block
        RDB_SHM_HDR_t* shmHdr = ( RDB_SHM_HDR_t* ) ( mImgShm->getStart() );

if ( !shmHdr )
return false;

// we need double buffering if ( ( shmHdr->noBuffers != 2 ) )
        return false;

// 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_INFO_t* ) ] );
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;
    }

RDB_MSG_t* pRdbMsgA = ( RDB_MSG_t* ) ( ( ( char* ) mImgShm->getStart() ) +
                                                pBufferInfo[0]->offset );
RDB_MSG_t* pRdbMsgB = ( RDB_MSG_t* ) ( ( ( char* ) mImgShm->getStart() ) +                                                                pB
RDB_MSG_t* pRdbMsg  = 0;

// check which buffer to read
if ( pRdbMsgA->hdr.frameNo && pRdbMsgB->hdr.frameNo )
    {
        if ( ( pBufferInfo[ 0 ]->flags & RDB_SHM_BUFFER_FLAG_TC ) &&
( pBufferInfo[ 1 ]->flags & RDB_SHM_BUFFER_FLAG_TC ) )
        {
          // use the "older" buffer
                if ( pRdbMsgA->hdr.frameNo < pRdbMsgB->hdr.frameNo )
            {
                pRdbMsg = pRdbMsgA;
                pCurrentBufferInfo = pBufferInfo[ 0 ];
            }
else
{
                pRdbMsg = pRdbMsgB;
                pCurrentBufferInfo = pBufferInfo[ 1 ];
            }
            // lock while reading
            pCurrentBufferInfo->flags |= RDB_SHM_BUFFER_FLAG_LOCK;
        }
        else if ( ( pBufferInfo[ 0 ]->flags & RDB_SHM_BUFFER_FLAG_TC ) )
        {
            pRdbMsg = pRdbMsgA;
            pCurrentBufferInfo = pBufferInfo[ 0 ];
            // lock while reading
            pCurrentBufferInfo->flags |= RDB_SHM_BUFFER_FLAG_LOCK;
        }
        else if ( ( pBufferInfo[ 1 ]->flags & RDB_SHM_BUFFER_FLAG_TC ) )
        {
            pRdbMsg = pRdbMsgB;
            pCurrentBufferInfo = pBufferInfo[ 1 ];
            // lock while reading
            pCurrentBufferInfo->flags |= RDB_SHM_BUFFER_FLAG_LOCK;
        }
    }
        // no image available
    if ( !pRdbMsg || !pCurrentBufferInfo )
    {
        delete pBufferInfo;
pBufferInfo = 0;
return false;
```

```
    }

    // process the actual image data in a dedicated routine
handleIgImg( &( pRdbMsg->u.image ) );

    // release after reading
    pCurrentBufferInfo->flags &= ~RDB_SHM_BUFFER_FLAG_TC;
pCurrentBufferInfo->flags |= mTaskCtrl->getShmMask();
pCurrentBufferInfo->flags &= ~RDB_SHM_BUFFER_FLAG_LOCK;

    delete pBufferInfo;
return true;
}
```

Setting Image Data for Headlight Distributions

To enable input via Shm, the IG configuration file must be modified as follows:

1. Load the corresponding plugin:

```
<Plugins>
     :
     <Plugin file="RDBInterface"/>
     :
</Plugins>
```

2. Configure the plugin:

```
<Components>
     :
     <RDBInterface name="MyRDBInterface"
                printDebugInfo="0"
                    ignoreShmFlags="1"/>
     :
</Components>
```

3. Configure the application:

```
<TAKATA
     :
     enableShmReader="1"
     :
/>
```

The user has to open a shared memory segment with the key `RDB_SHM_ID_IMG_GENERATOR_IN` and can, thereafter, post the following data (in HDR setup only!!):

RDB package of type `RDB_PKG_ID_LIGHT_MAP` (i.e. of structure `RDB_IMAGE_t`) with the following requirements:

id	must be the id of the light source that is to be switched (default: 1)
width, height	should be powers of two
pixelSize	32
pixelFormat	RDB_PIX_FORMAT_RGBA_24
imgSize	image size in bytes (= width * height * 4)
color[4]	RGBA of light source's base color
image data	array of width*height 32bit floats in physical units (cd/m²)

Resources

All connections may only be realized if certain resources are available and configured. The consistency of port numbers and server addresses is of special importance.

Ports

UDP:

sender	receiver	port (default)
TaskControl	ScenarioEditor	13105
ScenarioEditor	TaskControl	13106
TaskControl	Sichtapplikation	52304
TaskControl	RDB	48190
RDB	TaskControl	48191

TCP/IP:

server	client	port (default)
TaskControl	Operator Station	52300
TaskControl	traffic	52301
TaskControl	visual application, run-time data	52303
TaskControl	visual application, control data	13112
TaskControl	visual application, images	13110
TaskControl	SCP	48179
TaskControl	RDB images	48192
replay	data visualization	52305

Environment Variables

Environment variables may be used to adjust settings ad hoc without having to change configuration files. The variables must be set BEFORE starting a process that shall interpret the respective variables. Usually, environment variables are set within the process's start script.

The following table contains the most relevant environment variables, the name of the interpreting process, the purpose, and the default values

variable	processes	description	default
`PORT_TC_2_TRAFFIC`	taskControl traffic	port for messages to / from traffic module	50301
`PORT_TC_2_IG`	taskControl	port for messages to image generator	50302
`PORT_IG_2_TC`	taskControl	port for messages from image generator	50303
`PORT_TC_2_SCVIS`	taskControl scenarioEditor	port for messages to scenarioEditor	50305
`PORT_SCVIS_2_TC`	taskControl scenarioEditor	port for messages from scenarioEditor	50306
`PORT_TC_2_RDB`	taskControl moduleManager scpGenerator	RDB-data	48190
`PORT_RDB_2_TC`	taskControl moduleManager	RDB-feedback	48191

variable	processes	description	default
PORT_RDB_IMG	taskControl	RDB image transfer port	48192
PORT_IG_IMG	taskControl	port for the transfer of images from IG to TC	13110
PORT_IG_CTRL	taskControl	port for control commands to / from IG	13112
PORT_TC_2_SCP	taskControl	SCP message port	48179
PORT_TC_2_INST	taskControl	port for instrument simulation (some mockups)	4001
TRAFFIC_VIS_PORT_TC2IG	vIG	port for messages from TC to IG	50302
TRAFFIC_VIS_PORT_IG2TC	vIG	port for (UDP) messages from IG to TC	50303
PORT_SCP	SensorManager DynamicsManager ModuleManager scpGenerator	SCP-Port	48179
SERVER_SCP	SensorManager DynamicsManager ModuleManager scpGenerator	name of the computer hosting the SCP server	127.0.0.1
PORT_TC_2_SOUND	taskControl sound	port for data from TC to sound	13107
PORT_SOUND_2_TC	taskControl sound	port for data from sound to TC	13108
VI_MAX_MSG_COUNT	taskControl moduleManager	maximum number of messages that may be printed to the output stream	-1 (disabled)

Special Message Formats

The chapter describes the structure of the messages which are exchanged between the components.

- All binary data structures must be interpreted with 4-byte-alignment.
- The Binary messages according to internal standard are no longer disclosed.

The various special message formats are:

- Simulation Control Protocol (SCP)
- Runtime Data Bus (TC > any)
- Shared memory for VIL

Simulation Control Protocol (SCP)

The Simulation Control Protocol (SCP) can be used to control the simulation by 3rd-party components under non-real-time conditions. SCP is a text-based interface whose contents can be adapted easily to changing requirements without modifying the underlying mechanisms.

SCP uses XML syntax with the same delimiters and hierarchy structures. This also provides a means to pack command sequences into files and have them executed, e.g., by using the **scpGenerator** tool.

The SCP instruction set is composed of command areas, commands, and arguments. Each command area can contain one or more commands. Each command can hold an arbitrary number of arguments. Special focus is put on providing a flat command structure that is easy to understand even by non-expert users.

SCP Instruction Set

The complete list of commands has been excluded from this document and is now available at

`Doc/SCP_HTML/index.html`

By this, the documentation can be updated at a higher frequency, and, most of all, it is much easier to parse the list of available commands.

SCP Syntax Examples

The following examples show the use of the SCP syntax:

Example 1

In this example we define, switch, de-activate a textured symbol as 3D object attached to a player

1. Define and activate **niceSymbol**
    a. Switch the corresponding overlay to state 2
    b. Attach symbol to player own with the defined position in player's coordinate system
    c. Specify the symbol's real-world size

```
<Symbol name="niceSymbol">
  <Overlay id="0" state="2"/>
  <PosPlayer player="Own" dx="20.0" dy="0.0" dz="1.0"/>
  <RectSize width="6.0" height="4.0"/>
</Symbol>
```

2. Modify **niceSymbol** by switching corresponding overlay to state 1.

```
<Symbol name="niceSymbol">
       <Overlay id="0" state="1"/>
</Symbol>
```

3. Deactivate **niceSymbol**.

```
<Symbol name="niceSymbol">
  <Stop/>
</Symbol>
```

Example 2:

Replay some record files using different camera positions

- Replay a CSV-file, position camera in inertial space, look at player „Ego“

```
<Replay>
    <File path="../Data/RecPlay/" name="K_2_BC_01-003-A.csv"/>
    <Start/>
</Replay>

<Camera name="demoCam">
    <PosInertial x="-165.0" y="2380" z="40.0"/>
    <ViewPlayer player="Ego"/>
    <Set/>
</Camera>
```

- Replay a CSV-file, position camera in inertial space, look into constant direction:

```
<Replay>
    <File path="../Data/RecPlay/" name="K_2_BC_01-005-A.csv"/>
    <Start/>
</Replay>

<Camera name="demoCam">
    <PosInertial x="-165.0" y="2390" z="15.0"/>
    <ViewPos x="-230" y="2360" z="10.0"/>
    <Set/>
</Camera>
```

- Replay a CSV-file, position camera on player „Object 1“, look at player „Ego“:

```
<Replay>
    <File path="../Data/RecPlay/" name="K_2_BC_01-042-A.csv"/>
    <Start/>
</Replay>

<Camera name="demoCam">
    <PosRelative player="Object 1" dx="-8.0" dy="1.0" dz="8.0"/>
    <ViewPlayer player="Ego"/>
    <Set/>
</Camera>
```

Example 3: Turn on sensor symbols in a visual channel

- Show pedestrian symbols upon detection:

```
<Display>
    <SensorSymbols enable="true" mode="default" objectType="pedestrian"/>
</Display>
```

- Turn off pedestrian symbols, show vehicle detection symbols as frames:

```
<Display>
    <SensorSymbols enable="false" objectType="pedestrian"/>
    <SensorSymbols enable="true" mode="default" objectType="vehicle"/>
<Database enable="false"/>
</Display>
```

Packet Format

SCP instructions are transmitted as packets consisting of a header, which is immediately followed by the actual SCP data string of variable length. Header and content are transferred as one packet. The header structure is as follows:

type	size (bytes)	name	unit	value
unsigned short	2	magic number	-	40108

type	size (bytes)	name	unit	value
unsigned short	2	version number of header	-	0x0001
char	64	sender	-	-
char	64	receiver	-	-
int	4	length of the following text block	Byte	-

The sender and receiver of the packet are given in the header as standard text strings. The following code fragment shows the header file of the SCP interface. The corresponding file `scpIcd.hf` is located in `Develop/Framework/inc`.

```
/*****************************************************
*    ICD of the Simulation Control Protocol (SCP)          *
*-------------------------------------------------------    *
*    change log:                                           *
*    25.08.2008: added pragma instructions                 *
*    04.01.2008: created                                  *
*****************************************************/
#pragma pack (push, 4)

#ifndef _SCP_ICD_H
#define _SCP_ICD_H

/* ====== DEFINITIONS ====== */

#define SCP_DEFAULT_PORT 48179    /* default port for SCP communication */
#define SCP_NAME_LENGTH  64       /* length of a name sent via SCP */
#define SCP_MAGIC_NO     40108    /* magic number */
#define SCP_VERSION      0x0001   /* upper byte = major, lower byte = minor */
/**********************/
/*****   MESSAGE   *****/
/**********************/

typedef struct
{
        unsigned short  magicNo;                    /* must be 40108                          */
unsigned short  version;              /* upper byte = major, lower byte = minor    */
char            sender[SCP_NAME_LENGTH];    /* name of the sender as text                     */
char            receiver[SCP_NAME_LENGTH];  /* name of the receiver as text                    */
unsigned int    dataSize;             /* number of data bytes following the header      */
} SCP_MSG_HDR_t;

#endif      /* _SCP_ICD_H */

// end of pragma 4
#pragma pack(pop)
```

Encoding

SCP messages should be encoded using ISO 8859 (e.g., for scripts). Only by this can be guaranteed that special characters like the German Umlaute can be displayed correctly.

Runtime Data Bus (TC > any)

The RDB packages are composed of a package header and a series of pairs consisting of an entry header and an (optional) data block each. The type of the data block is given in the entry header.

To maximize bandwidth, a data block may contain a vector of elements of the identical type so that only one entry header is required to send, e.g., the positions of all players. The number of elements that follow an entry header is given in the entry header itself.

IMPORTANT: All binary data structures must be interpreted with 4-byte-alignment.

Contents ???

- Interface File

Please see `Develop/Framework/inc/viRDBIcd.h`

- Enclosing Entries: Each complete RDB frame (no matter whether it's sent in one package or a sequence of packages) must start with an entry of type
  `RDB_PKG_ID_START_OF_FRAME`
  and must end with an entry of type
  `RDB_PKG_ID_END_OF_FRAME`

Example

The following code fragment shows how to decompose an RDB package. Composing the package is just the other way round. This is just a code fragment and is provides AS IS.

```
void
handleMessage( RDB_MSG_t *msg )
{     if ( !msg )
return;

    if ( !( msg->hdr.dataSize ) )
return;

    size_t remainingBytes = msg->hdr.dataSize;

    RDB_MSG_ENTRY_HDR_t* entryPtr = &msg->entryHdr;

    while ( remainingBytes )
    {
            char* dataPtr = ( char* )entryPtr + entryPtr->headerSize;
remainingBytes -= entryPtr->headerSize;

      unsigned int noElements = entryPtr->elementSize ?
                          entryPtr->dataSize / entryPtr->elementSize : 0;

 handleDataVector( dataPtr, entryPtr->pkgId, noElements, entryPtr->flags,
   msg->hdr.simTime, msg->hdr.frameNo );

            remainingBytes -= entryPtr->dataSize;

        if ( remainingBytes )
        entryPtr = ( RDB_MSG_ENTRY_HDR_t* ) ( dataPtr + entryPtr->dataSize );
    }
}
void
handleDataVector( void* dataVec, unsigned short pkgId, unsigned int noElem,
   unsigned short flags, const double & simTime,
   const unsigned int & simFrame )
{
if ( !dataVec )
    return;
switch ( pkgId )
{
 :
 :
case RDB_PKG_ID_DRIVER_CTRL:
            handleDriverCtrl( ( RDB_DRIVER_CTRL_t* ) dataVec, noElem );
 break;
case RDB_PKG_ID_SYNC:
            handleSync( ( RDB_SYNC_t* ) dataVec, noElem );
break;
case RDB_PKG_ID_OBJECT_STATE:
handleObjectState( simTime, simFrame, ( RDB_OBJECT_STATE_t* ) dataVec,                                                    noEle
   break;
case RDB_PKG_ID_VEHICLE_SYSTEMS:
   handleVehicleSystems( ( RDB_VEHICLE_SYSTEMS_t* ) dataVec, noElem );
   break;
   case RDB_PKG_ID_OBJECT_CFG:
            handleObjectCfg( ( RDB_OBJECT_CFG_t* ) dataVec, noElem, simFrame );
 break;
case RDB_PKG_ID_END_OF_FRAME:
            handleEndOfFrame( simTime, simFrame );
   break;
default:
```

```
  break;
    }
}
```

Shared Memory for VIL

The shared memory for VIL applications contains the interface between the sensor task (interacts with the head tracker and inertial platform), the TaskControl, and the Image Generator.

The documentation is performed utilizing comments in the respective header files, which can be found at

```
Develop/VIL /vilsiShm.h
Develop/VIL /adma.h
Develop/VIL /hybird.h
Develop/VIL /itrace.h
```

File Formats

In this chapter, the configuration and run-time files are described in detail. All files are described under the name of their *producer* (module writing a file) if they are created during run-time or under their consumer's name if they are configuration files or have to be created by the user (as ASCII files).

IMPORTANT: All binary data structures must be interpreted with 4-byte-alignment.

Road Designer

Files and file formats of the Road Designer are described in a separate manual (ROD user manual, 2 parts). The following data (import) formats have been introduced also:

filename:	`<someName>` `.txt`
format:	ASCII

This file describes a road composed of the reference line, elevation information, and lanes. The file is interpreted per line so that each line first contains a keyword and then the respective number of parameters.

Common:

keyword:	//
	comment

Header:

keyword:	`InitRoad`
	lane geometry and road marks
parameter 1:	width driving lane [m]
parameter 2:	width shoulder lane [m]
parameter 3:	number of driving lanes (all in Ego's driving direction) [-]
parameter 4:	width of shoulder lane road mark [m]
parameter 5:	width of driving lane road mark [m]
parameter 6:	length of driving lane road mark [m]
parameter 7:	length of gap between two driving lane road marks [m]
parameter 8:	offset of driving lane road mark from begin of lane [m]
parameter 9:	optional: design speed [m/s]

If parameter 9 is defined, then the super-elevation will be computed automatically.

Body:

The following keywords may occur in arbitrary sequence and number. They describe the road's reference line.

keyword:	Straight
	straight line

	parameter 1:	length [m]
keyword:	CurveC0C1	
	spiral in xy plane	
parameter 1:	length [m]	
parameter 2:	horizontal curvature at begin of element [1/m]	
parameter 3:	first derivative of horizontal curvature [1/m²]	
keyword:	CurveC0C1L0L1	
	spiral in space	
parameter 1:	length [m]	
parameter 2:	horizontal curvature at begin of element [1/m]	
parameter 3:	first derivative of horizontal curvature [1/m²]	
parameter 4:	vertical curvature at begin of element [1/m]	
parameter 5:	the first derivative of vertical curvature [1/m²]	

Example:

```
////////////////////////////////////////////////////////////////////////////////////////
//
//  Roaddata  EXAMPLE
//
////////////////////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////////////////// //
//               laneWidth[m]   boundwidth[m]    Number      RoadStripe-     LaneStripe-  \
LaneStripe-    LaneStripe-    LaneStripe-       Design-
//                                                Lanes        Width[m]        Width[m]   \
length[m]        gap[m]      offset[m]        speed[m/s]
//
InitRoad            3.63            2.0            2            0.25            0.15       \
4.0            8.0            0.0            100.0
//
//
////////////////////////////////////////////////////////////////////////////////////////
//             length [m]
//
Straight       500.0
//                        length[m]   c0[1/m]  c1[1/m^2]   l0[1/m]  l1[1/m^2]
CurveC0C1L0L1        25.0          0.0         0.0             0.0        -0.00004
CurveC0C1L0L1       100.0         0.0         0.0            -0.001    0.0
CurveC0C1L0L1        25.0           0.0         0.0            -0.001    0.00004
```

ScenarioEditor

Configuration Files

- **Basic Configuration**

  file name: `scenarioEditor.xml.ini`
  location: `Data/Setup/Current/Config/ScenarioEditor`
  format: `ASCII / XML`

  This file is described in detail in [3].

- **Vehicle Library**

file name: `<VehicleNAme>.xml`
location: `Data/Projects/Current/Config/Players/Vehicles`
format: `ASCII / XML`

This file is described in detail in [3].

- **Pedestrian Library**

  file name: `characterCfg.xml`
  location: `Data/Projects/Current/Config/Players`
  format: `ASCII / XML`

  This file is described in detail in [3].

- **Driver Database**

  file name: `driverCfg.xml`
  location: `Data/Projects/Current/Config/Players`
  format: `ASCII / XML`

  This file is described in detail in [3].

- **Object Database**

  file name: <ObjectName>.xml
  location: `Data/Projects/Current/Config/Players/Objects`
  format: `ASCII / XML`

  This file is described in detail in [3].

Road Description

file name: `<roadNetworkName>.xodr`
format: `ASCII / OpenDRIVE / XML`

The ScenarioEditor can manage road description in the format OpenDRIVE 1.4. The format specification is publicly available via www.opendrive.org.

Scenario Description

file name: <scenarioName>.xml
format: `ASCII / XML`

This file is described in detail in [3].

VT-GUI (IOS)

Configuration File

On starting the GUI, the configuration file will be loaded from: `Data/Setups/Current/Config/VtGui`

On terminating the GUI, changes will be saved automatically. If the file cannot be found, a new file will be created.

file name: `database.xml`
format: `ASCII / XML`

Project File

The project file contains all information (specific data, cross-references) that belong to a given project.

Dataname: `<projectName>.vpi`
Format: `ASCII / XML`

Example :

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!--Unnamed - 16.07.2010/19:34:24-->
<Testcase author="" created="03.03.2009/21:38:50" modified="16.07.2010/19:34:24"
name="Unnamed" scenario="Current/Scenarios/traffic_demo.xml">
<![CDATA[]]>

  <ViewGroup selIdx="0">
   <View dae="10.000000 0.000000 10.000000" hprIne="0.000000 0.000000 0.000000"
    hprRel="0.000000 0.000000 0.000000" name="VIEW" posPlayerId="2"
    posSensorId="-1" posType="DefEyepoint" useSensorFrustum="false"
    viewPlayerId="2" viewType="Relative" xyzIne="0.000000 0.000000 0.000000"
    xyzRel="0.000000 0.000000 0.000000" xyzTgt="0.000000 0.000000
          0.000000"/>
 </ViewGroup>
 <SceneGroup selIdx="0">
    <Scene envDate="01.06.2008" envFriction="1.000000" envHeadlight="false"
     envRoadState="0" envSkyDome="2" envTime="11:00:00"
     envVisibility="100000.000000" name="SCENE" overlayFile="" sensor1Idx="0"
     sensor2Idx="-1" showDatabase="true" showEgo="false"
showSensorCone1="false" showSensorCone2="false"
    showSensorSymbols1="false" showSensorSymbols2="false"
   showSymbology="false" />
 </SceneGroup>
 <CameraGroup selIdx="0">
 <Camera clipPlanes="1.000000 1500.000000" definition="Frustum" fov="40.000000
  30.000000" iFocalX="400" iFocalY="300" iHeight="600" iPrincipalX="400"
  iPrincipalY="300" iWidth="800" matRow0="0.000000 0.000000 0.000000
  0.000000" matRow1="0.000000 0.000000 0.000000 0.000000"
  matRow2="0.000000 0.000000 0.000000 0.000000" matRow3="0.000000 0.000000
  0.000000 0.000000" name="CAMERA" offsetFrustum="0.000000 0.000000" />
      </CameraGroup>
    <DisplayGroup selIdx="0">
        <Display name="DISPLAY" showVisualId="33" showWinBorder="true"
              viewportX="0.000000 1.000000" viewportY="0.000000 1.000000"
                        winPos="0
              600" winSize="800 600" />
    </DisplayGroup>
    <SensorGroup>
        <Sensor clip="1.000000 90.000000" comPort="48185" comPortName="GSIout"
        comType="TCP" cull="true" debugCamera="false" debugCulling="false"
        debugDetection="true" debugDimension="false" debugPackages="false"
        debugPosition="false" debugRoad="false" filterLight="true"
       filterObstacle="true" filterPedestrian="true" filterTrafficSign="true"
       filterVehicle="true" fov="14.000000 10.000000" nCullObjects="10"
         name="SENSOR"
      offset="0.000000 0.000000" origHpr="0.000000 0.000000
          0.000000" origXyz="0.000000 0.000000 0.000000"
pluginFile="Current/Plugins/SensorManager/libPerfectSensor.so"
posHpr="0.000000 0.000000 0.000000" posXyz="2.100000 0.000000 0.500000"
selected="true" type="video" xmitEgoData="true" />
    </SensorGroup>
</Testcase>
```

Image Generator

The files for the configuration of the image generator are described in chapter Files and File Formats [75]

TaskControl

Data Recording (binary)

All binary data structures must be interpreted with 4-byte-alignment.

file name:	`recordName>.dat`
format:	`binary`

With VTD 2.0, the individual record file format has been discarded. Now, a record file consists of a binary dump of a sequence of RDB and SCP messages. Their magic numbers may identify the message types and maybe parsed according to the message types' format specifications. To parse a record file for analysis, you may use the RDBSniffer. Example:

```
rdbSniffer –f myRecordFile.dat
```

Data Recording (CSV)

The CSV-file is an alternative representation of the binary record file. It does not necessarily contain all details of the binary record file and is typically used for data export only (converting an existing binary record file into ASCII format).

file name:	`<projectName>.csv`
format:	`ASCII`

Conversion of record files into CSV files will be performed by the RDBSniffer tool. Please look at its command-line options for further instructions (see above).

Batch File of the SCP-Generator

The scpGenerator can interpret the batch file. Its syntax is line-based. In each line, there have to be two items:

```
<time> <command>
```

The time may be given in the following manners:

`<n>`	command will be sent in the n^{th} frame of the generator
`+<n>`	command will be sent n frames after the previous command
`<n>s`	command will be sent n seconds after the start of the generator
`+<n>s`	command will be sent n seconds after the previous command

The commands have to be enclosed with double quotation marks (“) and have to comply with the SCP syntax. Lines starting with a # character are interpreted as comments and will not be sent. Commands ranging over various lines in the file have to be linked with the \ character at the end of each line (except for the last one).

Example:

```
# series of SCP commands to be sent at given frame numbers
#
# NOTE: handles only one command per frame!!
#
```

```
"<SimCtrl> <Stop/> </SimCtrl>"
"<SimCtrl> <LoadScenario filename="../Data/Scenarios/default.xml" /> <Start \
 mode="preparation"/> </SimCtrl>"
 +20 "<EgoCtrl> <Speed value="22.0"/> </EgoCtrl>"
 +1 "<Display> <Database enable="true"/>     </Display>"
 +1 "<Display> <SensorSymbols enable="true" sensor="perfect"/> </Display>"
```

If the first line of an SCP file reads `syncToSim`, the scpGenerator will use the simulation time/frames instead of its internal time for timing the commands.

If a line reads the following, execution of an SCP script will be paused until the specified command (case sensitive and whitespace sensitive) arrives.

```
wait "<command to wait for>"
```

Module Plug-Ins

The respective commands give the syntax of the ModuleManager configuration file in the SCP documentation (command classes `<Sensor>` and `<DynamicsPlugin>`)and the examples in the respective chapters.

Tips 'n Tricks

Initialization

Triggering Initialization

The system will init after it has received via SCP

```
<SimCtrl><Init mode="..."/></SimCtrl>
```

This command will be issued either by the GUI upon pressing the INIT button or any other source connected to the SCP communication. If the start command

```
<SimCtrl><Start mode="..."/></SimCtrl>
```

is issued before the system has received an INIT, the initialization will be triggered internally.

11.1.2 Initialization Involving 3rd Party Components

The initialization is finished after all VTD components have signaled to the TC that they are ready to go. External components (i.e., 3rd party components) are ignored unless they explicitly register for a synchronized initialization using the following method.

Before the first INIT command is issued via SCP, the components have to register for synchronized initialization utilizing the SCP command

```
<Query entity="taskControl">
    <Init source="myName"/>
</Query>
```

The TaskControl will acknowledge this request by sending

```
<Reply entity="taskControl">
    <Init source="myName"/>
</Reply>
```

After INIT, the taskControl will wait until the registered component sends an init confirm via

```
<SimCtrl><InitDone source="myName"/></SimCtrl>
```

To avoid blocking the entire simulation by 3rd party components, it is recommended to also define a timeout upon registration for the synchronized initialization using the following syntax:

```
<Query entity="taskControl">
    <Init source="myName" timeout="mySeconds"/>
</Query>
```

With *mySeconds* being the timeout in [s] as a floating-point number.

A registration for synchronized initialization is valid without limit; therefore, components that don't want to continue with the synchronization have to unregister from the mechanism using the following SCP syntax.

```
<Query entity="taskControl">
    <Init source="myName" delete="true"/>
</Query>
```

The TaskControl will acknowledge this request by sending

```
<Reply entity="taskControl">
    <Init source="myName"/>
</Reply>
```

11.2 Frame Synchronization

The simulation may run in free mode or frame-synchronous mode. In free mode, the internal loop consisting of TaskControl and Traffic will run independently of external components (e.g., external vehicle dynamics), only depending on the sync settings in

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

within the section

`<Sync .../>`

Once 3rd party components are connected via RDB, it might be necessary for the simulation to delay the next frame's computation until all connected components are ready for it. Various methods exist to achieve this behavior:

11.2.2 Single Sync Source with Explicit Step Width

The sync may depend on a single source, providing an explicit step width for the simulation. For this, the TaskControl has to be configured in the file

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

Using one of the following parameters:

- `<Sync source="RDB" ... />`
- `<Sync source="SCP" ... />`

In the former case, the synchronization is performed via RDB, in the latter one via SCP. For the sync via RDB, the actual sync source has to send (via RDB) a message of type

`RDB_PKG_ID_TRIGGER`

which contains the delta time for the computation of the next frame and the actual frame number.

For a synchronization via SCP, the actual sync source has to send (via SCP) a message of the following syntax:

```
<SimCtrl><Sync dt="dt in seconds"/></SimCtrl>
```

11.2.3 Multiple Sync Dependencies

Independent of the actual primary sync source, the synchronization may actually depend on various components having finished their current frame before computing the next one. These multiple dependencies may be handled via RDB utilizing a subscriber mechanism.

To subscribe, i.e., register as a component which has to "authorize" the next frame the component must send once the SCP command

```
<Query entity="RDB">
    <Sync source="myName"/>
</Query>
```

The TaskControl will acknowledge this request by sending

```
<Reply entity="RDB">
    <Sync source="myName" mask="intValue" mode="frame"/>
</Reply>
```

After computing each frame, the respective component has to send (via RDB) a message of type

`RDB_PKG_ID_SYNC`

which contains the mask that has been issued with the above <Reply/> command. By this, multiple sync dependencies may be handled.

To unregister from the synchronization mechanism, a component has to send via SCP

```
<Query entity="RDB">
    <Sync source="myName" delete="true"/>
</Query>
```

which will be acknowledged by

```
<Reply entity="RDB">
    <Sync source="myName" mode="free"/>
</Reply>
```