



Taking you to the next level



IPG Documentation

CarMaker®

Programmer's Guide Version 5.1

SimulationSolutions • TestSystems • EngineeringServices

The information in this document is furnished for informational use only, may be revised from time to time, and should not be construed as a commitment by IPG Automotive GmbH. IPG Automotive GmbH assumes no responsibility or liability for any errors or inaccuracies that may appear in this document.

This document contains proprietary and copyrighted information and may not be copied, reproduced, translated, or reduced to any electronic medium without prior consent, in writing, from IPG Automotive GmbH.

© 1999 - 2016 by IPG Automotive GmbH – www.ipg-automotive.com
All rights reserved.

FailSafeTester, IPGCar, IPGControl, IPGDriver, IPGEngine, IPGGraph, IPGKinematics, IPGLock, IPGMotorcycle, IPGMovie, IPGRoad, IPGRoaddata, IPGTire, IPGTrailer, IPGTruck, RealtimeMaker, Xpack4 are trademarks of IPG Automotive GmbH.

CarMaker, TruckMaker, MotorcycleMaker, MESA VERDE are registered trademarks of IPG Automotive GmbH.

All other product names are trademarks of their respective companies.

Contents

1	The CarMaker Environment	14
1.1	CarMaker architectural breakdown	14
1.1.1	CarMaker user interface programs (CIT)	14
1.1.2	CarMaker simulation program (VVE)	14
1.1.3	Data file organisation	15
1.1.4	Program interaction	16
1.1.5	Interaction of HIL systems	16
1.2	Inside the CarMaker simulation program	18
1.2.1	User accessible modules	19
1.3	Rebuilding the CarMaker Simulation Program	21
1.3.1	Preparing your Project Folder	21
1.3.2	Building the new Executable	21
1.3.3	Using the new Executable	26
1.3.4	Rebuilding the model library for CarMaker for Simulink	26
1.3.5	Identifying which model library you are using	26
1.4	Debugging the CarMaker Simulation Program	28
1.4.1	Starting CarMaker in the Debug Mode	28
1.4.2	Basic Functionalities of GDB	29
1.4.3	Debugging CarMaker using Eclipse	29
1.5	CarMaker flow process charts	33
1.5.1	Modified cycle clock generation if oversampling is used	34
1.6	The main cycle explained	37
1.6.1	An pseudo code excerpt from CM_Main.c	38

1.6.2	Basic tasks of the main routine	39
1.6.3	The event loop of the main routine	39
1.7	Using CarMaker/HIL with ETAS Hardware	41
1.7.1	System Overview	41
1.7.2	Realtime System Setup	42
1.7.3	Preparing your project	43
1.7.4	Starting the Simulation	46
1.8	Using CarMaker/HIL with dSPACE Hardware	47
1.8.1	IPG Realtime Services	47
1.8.2	Troubleshooting	48
1.8.3	Extending supplied models by your own ones	49
1.9	Using CarMaker/HIL with National Instruments Hardware	49
1.9.1	System Overview	49
1.9.2	First steps with CarMaker	51
1.9.3	Using CarMaker with NI LabVIEW	52
1.9.4	Using CarMaker with NI VeriStand	54
1.9.5	Troubleshooting	56
2	Logging Module	57
2.1	General information	57
2.2	Recommended use	58
2.2.1	List of functions	58
3	Infofile Module	60
3.1	Infofile format	60
3.2	Access functions	61
3.3	Error handling	61
3.4	C Function List	62
3.4.1	General Functions	62
3.4.2	Read Functions	66
3.4.3	Write Functions	70
3.4.4	Add/Move Functions	71
3.4.5	Helper Functions	73
3.4.6	Error handling	74
3.5	Tcl/Tk Procedure List	74
3.5.1	Library	74
3.5.2	ifile	74
3.5.3	<handle>	75

4 Data Dictionary	79
4.1 Introduction	79
4.2 Defining DataDict Variables	79
4.2.1 General hints	79
4.2.2 Quantity Definition API	80
5 Integrating C-code models	83
5.1 Model Manager	83
5.1.1 Model Registration	86
5.1.2 Exceptions and special cases	87
5.1.3 Using parameter infofiles	88
5.2 Demonstration examples	90
5.2.4 Preparation	90
5.2.5 Running the model	90
6 CarMaker for Simulink	92
6.1 CarMaker for Simulink basics	93
6.1.1 Starting Matlab	93
6.1.2 Creating a new model	94
6.1.3 Starting the CarMaker GUI	95
6.1.4 Running a simulation	95
6.1.5 Switching between several Simulink models	95
6.1.6 Switching between several CarMaker project directories	96
6.1.7 Dealing with the start values of your model	96
6.1.8 Upgrading to a new CarMaker version	96
6.2 The CarMaker interface blockset	98
6.2.1 General information	98
6.2.2 Utility blocks	99
6.2.3 Accessing the CarMaker dictionary	100
6.2.4 Defining CarMaker dictionary variables	102
6.2.5 Accessing CarMaker Infofile Parameters	105
6.2.6 Accessing Sensor Data	106
6.2.7 Accessing C variables	112
6.2.8 Special purpose blocks	113
6.3 CarMaker Subsystem	122
6.4 Demonstration examples	137
6.4.1 ACC	138
6.4.2 AccelCtrl_ACC	139
6.4.3 BodyCtrl	139

6.4.4	HydBrakeCU_ESP	140
6.4.5	PTBatteryCU	141
6.4.6	PTControl	142
6.4.7	PTEngineCU	143
6.4.8	PTMotorCU	144
6.4.9	PTTransmCU	145
6.4.10	SoftABS_Hyd	146
6.4.11	TractCtrl	147
6.4.12	UserBrake	147
6.4.13	UserPowerTrain	148
6.4.14	UserSteer	149
6.4.15	UserSteerTorque	150
6.4.16	UserTire	152
6.4.17	Generic_UserVehicle	153
6.4.18	SingleTrack	154
6.5	Troubleshooting	155
7	CarMaker utilities for Matlab	156
7.1	Importing simulation results with cmread	156
7.2	Accessing CarMaker Infofiles	158
7.3	Sending Tcl Commands to the CarMaker GUI	158
8	Simulink Coder Interface	159
8.1	Starting Matlab	159
8.2	The CarMaker target for Simulink Coder	160
8.2.1	Code generation with the CarMaker target	160
8.2.2	Choosing the right model wrapper	161
8.3	Integration of Simulink models with CarMaker's Model Plug-in	163
8.4	Example: Integrating a user defined model using the CarMaker Model Plug-in	165
8.5	Details on the integration of Simulink models	167
8.5.1	Setting simulation parameters	167
8.5.2	Customizing the wrapper	170
8.5.3	Integrating the model into CarMaker	171
8.5.4	Activating the model	173
8.6	Example: Integration of a Simulink model using the Plain wrapper	173
8.6.1	Creating the Simulink model	173
8.6.2	Setting simulation parameters	174
8.6.3	Generating and compiling the model C-code	174

8.6.4	Running the model	176
8.7	CarMaker's tunable parameter interface	177
8.7.1	Introduction	177
8.7.2	Enabling tunable parameters in a Simulink model	178
8.7.3	Modifying tunable parameters in the model wrapper	179
8.7.4	Parameter values in Infofiles	180
8.7.5	Adding tunable parameters to the CarMaker dictionary	181
8.7.6	Tunable parameter interface functions	182
8.8	Miscellaneous	189
8.8.1	Upgrading to a new CarMaker version	189
8.8.2	CarMaker and Matlab installed on different computers	189
8.8.3	Troubleshooting	191
8.9	Demonstration examples	192
8.9.1	Contents of the examples directory	192
8.9.2	Preparing the examples	192
8.9.3	Rebuilding an example	192
9	Functional Mock-up Interface (FMI)	193
9.1	Integrating an FMU	193
9.2	Configuring the FMU	195
9.2.1	Selecting the Model Class	195
9.2.2	Connecting Input and Output Signals	197
9.2.3	Special Options	197
9.3	Simulating with an FMU	199
9.3.4	FMU Parameterization	199
9.4	Example FMUs	199
9.5	FMU Debugging Assists	200
9.5.5	Turning on debug logging	200
9.5.6	Handling voluminous logging output	200
10	MIO – M-Module Input/Output	203
10.1	Supported M-Modules	203
10.2	Programming M-Module I/O	205
10.2.1	MIO Initialization	205
10.2.2	M-Module Configuration	206
10.2.3	Error Handling	206
10.2.4	Custom MIO Drivers	208
10.3	Administrative Functions	213
10.3.1	Initialization and M-Module Configuration	213

10.3.2	MIO and M-Module Information	214
10.3.3	Error Handling	216
10.4	M-Module Function Description	217
10.4.1	M27: Binary Output (16 channels)	217
10.4.2	M28: Binary Output (16 channels)	219
10.4.3	M31: Binary Inputs (16 channels)	221
10.4.4	M32: Binary Inputs (16 channels)	223
10.4.5	M35 / M35N: Analog Inputs (16/8 channels)	225
10.4.6	M36N: 16 Bits Analog Inputs (16/8 channels)	230
10.4.7	M43: Relay Outputs (8 channels)	236
10.4.8	M51: Quadruple CAN Interface	238
10.4.9	M62 / M62N: Analog Outputs (16 channels)	252
10.4.10	M66: Binary Inputs/Outputs (32 channels)	255
10.4.11	M77: Quadruple RS232/423 - RS422/485 UART	259
10.4.12	M81: Binary Outputs (16 channels)	264
10.4.13	M400: Wheelspeed Generator, 6 Channels	266
10.4.14	M401: Frequency Generator and Frequency Meter (3 Units) ..	273
10.4.15	M402: Engine Signal Generator, 8 Channels	285
10.4.16	M403: Engine Signal Detector, 8 Channels	300
10.4.17	M402 & M403: synchronized Use of Several M402 / M403 ..	315
10.4.18	M404: Waveform Generator, 8 Channels	317
10.4.19	M405: LIN Interface, 12 Channels	321
10.4.20	M406: PSI5 Interface, 8 Slaves + 1 Master	331
10.4.21	M407: SPI Interface, 2 Slaves or 1 Master	340
10.4.22	M408: PWM Meter and SENT Receiver, 20 Channels	354
10.4.23	M409: Power Supply Control, 2 Units	365
10.4.24	M410: Quadruple CAN/CANFD Interface	374
10.4.25	M412: Parksensor Simulation Board	385
10.4.26	M413: 5x 4:1 Multiplexer / De-Multiplexer	393
10.4.27	M440: Resistor Cascade, 6 Channels	396
10.4.28	M441: PWM Out and SENT Transmitter, 12 / 16 Channels ..	399
10.5	Discontinued M-Modules	415
10.5.1	M34: Analog Inputs (16/8 channels)	415
10.5.2	M72: Motion Counter / Multipurpose Module	417
11	USB IO – USB Input/Output Interfaces	426
11.1	PowerUTA	426
11.1.1	Connector assignments	427
11.1.2	Functional description	427

12 FlexRay	430
12.1 Overview	431
12.1.1 Requirements	431
12.1.2 Basic information	431
12.1.3 Features	432
12.2 FlexConfig	434
12.2.1 CarMaker/HIL Plug-in	434
12.2.2 Export for Gateway Mode	440
12.3 Rest Bus Simulation in CarMaker/HIL	442
12.3.1 Integration of FlexRay in a CarMaker Project	442
12.3.2 Initialization of FlexRay Framework	442
12.3.3 Rest Bus Simulation	444
12.3.4 Customization	445
12.3.5 Data flow in Gateway Mode	447
12.4 Functional description	451
12.4.1 FlexCard Function Interface	451
12.4.2 RBS Function Interface	458
12.5 Configuration Files	466
12.5.1 FlexRayParameters	466
12.5.2 FlexCardParameters	478
12.5.3 RBSPatterns	481
13 SOME/IP	489
13.1 Overview	490
13.1.1 Basic information	490
13.1.2 Features	490
13.2 Rest Bus Simulation in CarMaker/HIL	491
13.2.1 Integration of SOME/IP in a CarMaker Project	491
13.2.2 Initialization of SOME/IP Framework	491
13.2.3 Rest Bus Simulation	493
13.2.4 Customization	494
13.3 Functional description	497
13.3.1 SOME/IP RBS Function Interface	497
13.4 Configuration Files	504
13.4.1 SvcParameters	504
13.4.2 SIPParameters	510
14 CCP / XCP	514
14.1 Overview	515

14.1.1	Basic information	515
14.1.2	Features	518
14.1.3	Limitations	520
14.2	Architectural Overview	521
14.2.1	Overview of the CCP protocol integration into CarMaker	521
14.2.2	Overview of the XCP protocol integration into CarMaker	522
14.3	Integration in a CarMaker Project	523
14.3.1	(Re-)Building the CarMaker Executable	523
14.3.2	Tuning the CarMaker GUI	529
14.3.3	Configuring the Communication Module	530
14.3.4	Mapping Measurement Quantities to the CarMaker Model	531
14.4	Configuration Parameters	533
14.4.1	CCPParameters	533
14.4.2	XCPParameters	535
14.4.3	Limiting the number of generated Data Dictionary variables	541
14.4.4	CCP / XCP (De-)Activation via Onboard Diagnostics	542
14.4.5	ASAP2 variable Mappings	543
14.5	The CCP and XCP Configuration Dialog	544
14.5.1	Structure of the XCP Configuration Dialog	545
14.6	ScriptControl Interface	551
14.6.1	CCP Commands	551
14.6.2	XCP Commands	558
14.7	References	572
15	EtherCAT	573
15.1	Overview	574
15.1.1	Basic Information	574
15.1.2	Features	575
15.1.3	Limitations	576
15.2	EtherCAT Implementation in CarMaker/HIL	577
15.2.1	Hardware Integration of EtherCAT in Xpack4	577
15.2.2	Integration of EtherCAT in a CarMaker Project	577
15.2.3	Initialization of the EtherCAT Framework	578
15.2.4	Online Communication	580
15.3	Functional Description	582
15.3.1	EtherCAT Function Interface	582
15.4	Configuration Files	587
15.4.1	CIFXParameters	587
15.4.2	ECATParameters	589

15.4.3 EtherCAT Slave Information	591
15.5 Header Files	592
15.5.1 EtherCAT_Slave.h	592
16 CANiogen – CANdb Import Tool	594
16.1 Overview	595
16.1.1 Basic information	595
16.1.2 Features	597
16.2 Using CANiogen	599
16.2.1 The CANiogen Command Line	599
16.2.2 Importing Electronic Control Units (ECU)	603
16.2.3 Receiving CAN messages and signals of special interest ..	605
16.2.4 Sending arbitrary CAN messages	605
16.2.5 Suppressing of I/O variables in the Data Dictionary	606
16.2.6 Customizing CANiogen	607
16.2.7 Optimizing the output of CANiogen	608
16.2.8 Disabling range checking of Signal values	608
16.2.9 Naming of generated files, I/O variables and functions	608
16.2.10 Naming of generated Data Dictionary entries	609
16.2.11 Extended features	610
16.3 CANiogen's output (C-Code generator Mode)	612
16.3.1 Output files	612
16.3.2 The header file IO_CAN.h	612
16.3.3 The list of generated I/O variables in IO_VarList.txt	621
16.3.4 The C module IO_CAN.c	624
16.3.5 The header file IO_CAN_User.h	630
16.3.6 The C module IO_CAN_User.c	631
16.4 CANiogen's output (J1939 Mode)	634
16.4.1 Output files	634
16.4.2 The J1939 Parameters file	634
16.4.3 The J1939 Mappings file	637
16.5 Integration into CarMaker/HIL (C-Code Mode)	639
16.5.1 Modifications to IO.c	639
16.5.2 Modifications to User.c	643
16.5.3 Modifications to the Makefile	644
16.6 Integration into CarMaker/HIL (J1939 Mode)	649
16.6.1 Modifications to the Makefile	650
16.6.2 Modifications to IO.h	651

16.6.3	Modifications to IO.c	651
16.7	Using databases with CAN FD messages	652
16.8	Version History	653
17	ScriptControl	663
17.1	Introduction	663
17.2	Using the ScriptControl Window	664
17.3	ScriptControl by Example	665
17.3.1	Example 1 – “Hello World!”	665
17.3.2	Example 2 – Starting a TestRun	665
17.3.3	Example 3 – Subscribing to Quantities	666
17.3.4	More Examples	667
17.3.5	Tcl/Tk Documentation Links	669
17.4	ScriptControl Command Reference	670
17.4.1	Loading and Running a Simulation	670
17.4.2	Subscribing to Quantities	674
17.4.3	Clock Timers	676
17.4.4	Logging	678
17.4.5	Infofile Parameter Access	681
17.4.6	KeyValue Parameter Substitution	683
17.4.7	NamedValue Infofile Parameter Substitution	686
17.4.8	Managing the Scratchpad	689
17.4.9	QuantAudit – Pin-point Quantity Monitoring	694
17.4.10	Data Storage	697
17.4.11	Direct Variable Access (DVA)	703
17.4.12	Executing Matlab Commands	704
17.4.13	Application commands	706
17.4.14	PopupCtrl Commands	710
17.4.15	Vehicle Operator / Power Control (KL15)	712
17.4.16	FailSafeTester Commands	714
17.4.17	IPGDriver Commands	719
17.4.18	IPGMovie Commands	721
17.4.19	Concerto Commands	731
17.4.20	Test Manager commands	734
17.4.21	Report commands	741
17.4.22	INCA commands	744
17.4.23	tcom command family	755
17.4.24	Miscellaneous Commands	758

18 Remote GUI Control	762
18.1 TCP Command	762
18.1.1 Starting the TCP command server	762
18.1.2 Protocol specification	762
18.1.3 Testing the TCP command server with telnet	764
18.2 Tcl send (Linux only)	764
18.3 DDE (MS Windows only)	764
18.4 CarMaker for Simulink	765
19 FailSafeTester	766
19.1 FST Global Variables	766
19.2 Configuring the Communication via CAN	766
19.3 FailSafeTester C-Functions	766
A Replacing the Vehicle Model	770
A.1 User Vehicle as a C-Code extension	770
A.1.1 Tires	771
A.1.2 Brake	771
A.1.3 Powertrain	771
A.1.4 The MyCar Example	771
A.2 User Vehicle as a Plug-in Model in Simulink	772
A.2.1 Tires	772
A.2.2 Preprocessing	772
A.2.3 The UserVehicle_RTW Example	772
A.2.4 The SingleTrack_RTW Example	773
A.3 User Vehicle in CarMaker for Simulink	773
A.3.1 Tires	773
A.3.2 The SingleTrack Example	773
A.4 Vehicle Interface Signals	774
B Generic Plug-in Models	778
B.1 Introduction	778
B.2 Handling of a Generic Plugin model	778
B.3 General Parameters	779

Chapter 1

The CarMaker Environment

1.1 CarMaker architectural breakdown

Let's first have a look at CarMaker and some of its basic architectural concepts. We will take the perspective of a user whose job it is to build TestRuns, run simulations with the TestRuns he created, and to postprocess the results of these simulations. What he sees when he's working with CarMaker will be the basis for a first glance at the CarMaker architecture. We will need the concepts explained here later, when we describe the vehicle module, its interface and its connection to the outside world in greater detail.

1.1.1 CarMaker user interface programs (CIT)

The first thing the user will notice, is that CarMaker consists of several individual programs. The one program that he almost certainly knows he has to start in order to use CarMaker, is the CarMaker GUI. The CarMaker GUI's main tasks are to let the user create TestRuns, edit vehicle parameters and start and stop simulations of TestRuns.

When simulating a TestRun, the user certainly wants to have some kind of feedback of what is going on during the simulation. From within the CarMaker GUI he starts the 3D animation tool IPGMovie, because he wants to see the vehicle driving on the given course. He may start the online data visualization tool IPGControl to be able to inspect the time dependent behaviour of certain physical quantities of vehicle during the simulation. Or he might want to start Instruments to visualize the vehicle behaviour using the instruments and lights he knows from a real dashboard.

IPGMovie, IPGControl and Instruments are all implemented as individual programs, that can be started from the CarMaker GUI (they may as well be started outside the CarMaker GUI), and which run independently of each other. Together they comprise the graphical user interface of CarMaker. You find them on the left hand side of Figure 1.1.

1.1.2 CarMaker simulation program (VVE)

Since the tasks of the programs you read about in the previous section have nothing to do with any calculation of physical quantities for the simulation, this leads us to the next important concept of CarMaker. It is the concept of a separately running CarMaker simulation program. Its task is not to provide any kind of elaborate user interface, but to perform the actual

simulation of a TestRun and – in case of CarMaker/HIL – to interface with external hardware being part of the control loop, e.g. a real controller unit. The CarMaker simulation program and possibly connected hardware can be found in the middle and on the right of Figure 1.1.

1.1.3 Data file organisation

Working with CarMaker always takes place in a so called CarMaker project directory that keeps the user's files needed by CarMaker well organized in several subdirectories. A look at Figure 1.1 shows some typical subdirectories that can be found in a project directory. Each subdirectory serves a special purpose; the following paragraphs give you an impression how file data is used and shared by the individual CarMaker programs.

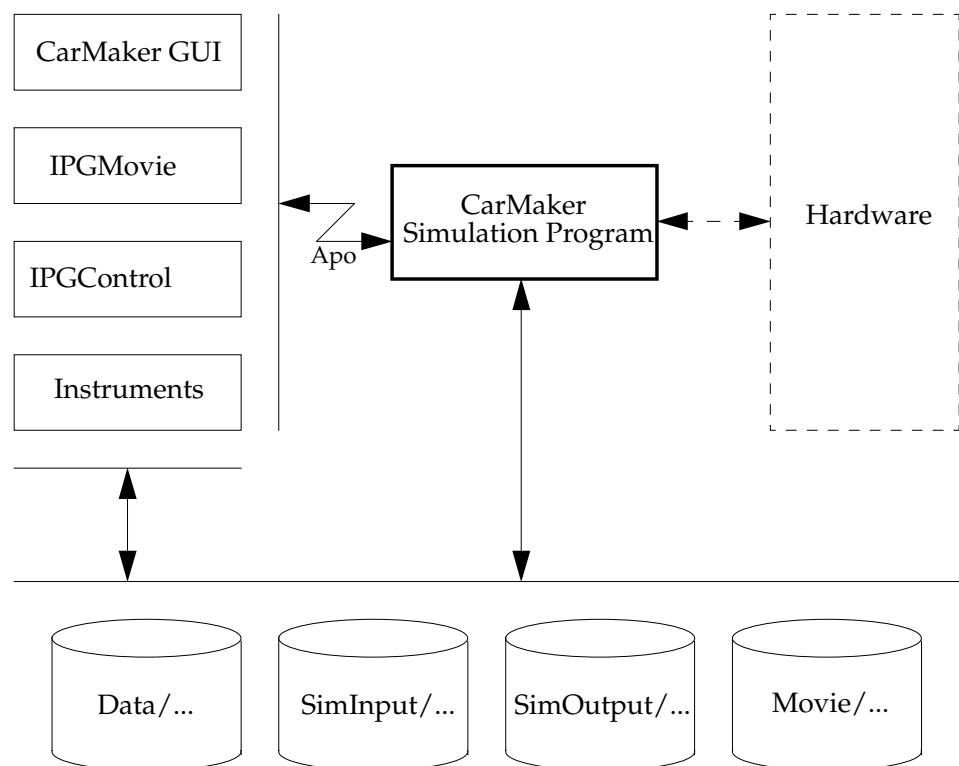


Figure 1.1: Basic CarMaker Architecture

As described before, the user creates TestRuns and edits vehicle parameters using the CarMaker GUI. The CarMaker GUI stores TestRun data in files in the Data subdirectory.

Each time a simulation is started, the CarMaker simulation program reads information about the current TestRun from exactly these files in the Data subdirectory. Files needed in addition to that might be taken from the SimInput subdirectory. During the simulation, a log file and simulation results are written to files in the SimOutput subdirectory.

Also at the start of a simulation, the CarMaker GUI is responsible for providing 3D road geometry data to IPGMovie. This becomes necessary if a TestRun does not use pre-existing real-world digitized road data but a course constructed of individual road segments. IPGMovie then reads road geometry data and 3D vehicle geometry data from the Movie subdirectory.

At a later time, for the purpose of postprocessing, IPGMovie or other software like, as an example, Matlab can read the simulation results in SimOutput written by the CarMaker simulation program.

1.1.4 Program interaction

Having the user interface tools and the simulation program run absolutely independent of each other would not make much sense – there needs to be some kind of coordination between them. For the purpose of program interaction, the CarMaker simulation program plays a central role.

The user interface tools connect to the simulation program. They may send commands to them, like the CarMaker GUI with its “Start” or “Stop” commands. These are sent each time the user clicks on the Start or Stop button. The interface tools may also request some kind of service, e.g. ask for a list of available quantities and request regular transmission of quantity values. This is what IPGControl does in order to display them graphically during the running simulation.

The CarMaker simulation program, in turn, must provide these services. It must react to commands sent by the CarMaker interface tools. It must register quantities that tools might want to receive on a regular basis, e.g. the quantity that contains the current simulation time, or physical quantities calculated by the vehicle module inside the simulation program. Also, at simulation start, the simulation program sends a special message to IPGMovie telling it about the current geometry configuration of the vehicle.

Communication between the CarMaker programs is done using standard network communication mechanisms. A special CarMaker module, the Apo library, implements communication services for the CarMaker programs and defines the “language” being used between them. The lightning titled “Apo” in Figure 1.1 should illustrate these facts. For the curious: Apo is an abbreviation of “Applications online”.

1.1.5 Interaction of HIL systems

CarMaker HIL systems use different computers to distribute the applications. Figure 1.3 shows a realtime computer which hosts the CarMaker/HIL application. All models and services have to meet realtime conditions. Therefore the realtime operating system XENO is used on this machine. The realtime system is diskless and shares the disk with the user workstation over network file system.

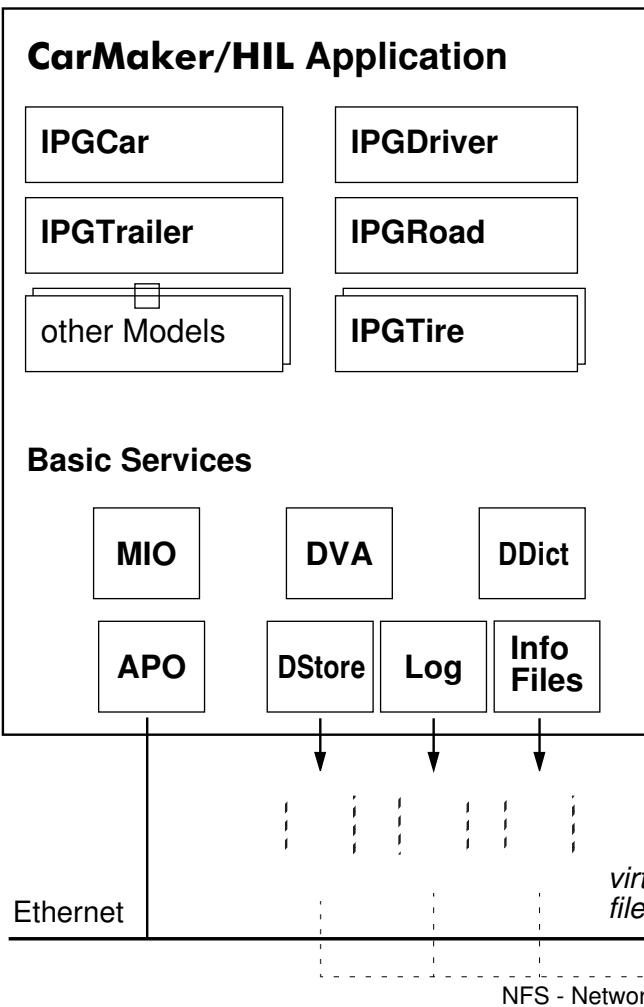
The user workstation hosts all user interface programs and the software development kit to generate custom CarMaker/HIL applications for the realtime computer.

All network communication is done over ethernet by the use of the following protocols:

- PXE, TFTP, DHCP/BOOTP: boot realtime computer over network, determine network address
- telnet: remote terminal
- NFS: virtual file system
- APO: CarMaker communication stack (developed by IPG), based on TCP/IP (UDP and TCP sockets)

Realtime System

XENO
diskless



User Workstation

Linux or Windows

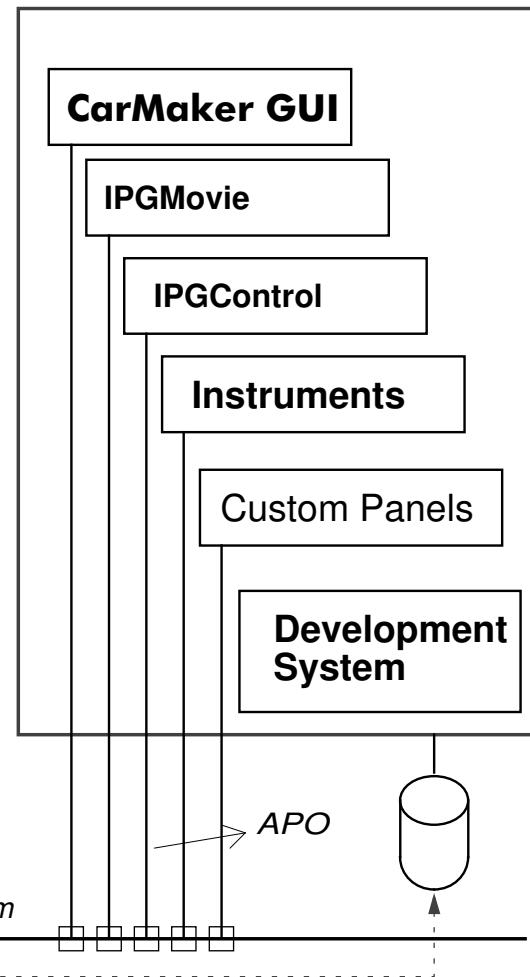


Figure 1.3: Program interaction for HIL systems

1.2 Inside the CarMaker simulation program

As we know from the previous section, the central component within the CarMaker architecture is the CarMaker simulation program. We will now take a closer look at this program to see which modules it consists of and find out about the role that each of these modules plays within the simulation program. Emphasis will be on the vehicle module; this section should give you an impression how the vehicle module is connected to the surrounding CarMaker environment.

Figure 1.5 shows the basic building blocks of the CarMaker simulation program. The files and libraries are more or less the same ones that will be linked together when you rebuild the simulation program. What we can see is that there are four major groups of modules:

- User accessible C code modules (at the top)
- The CarMaker library `libcarmaker.a` (left block in the middle)
- The vehicle library `libcar.a` (right block in the middle)
- Special purpose libraries (at the bottom)

We will examine each group more closely in a moment.

The figure also shows the call hierarchy inside the program: each module calls only functions of other modules of the same level or on a lower level.

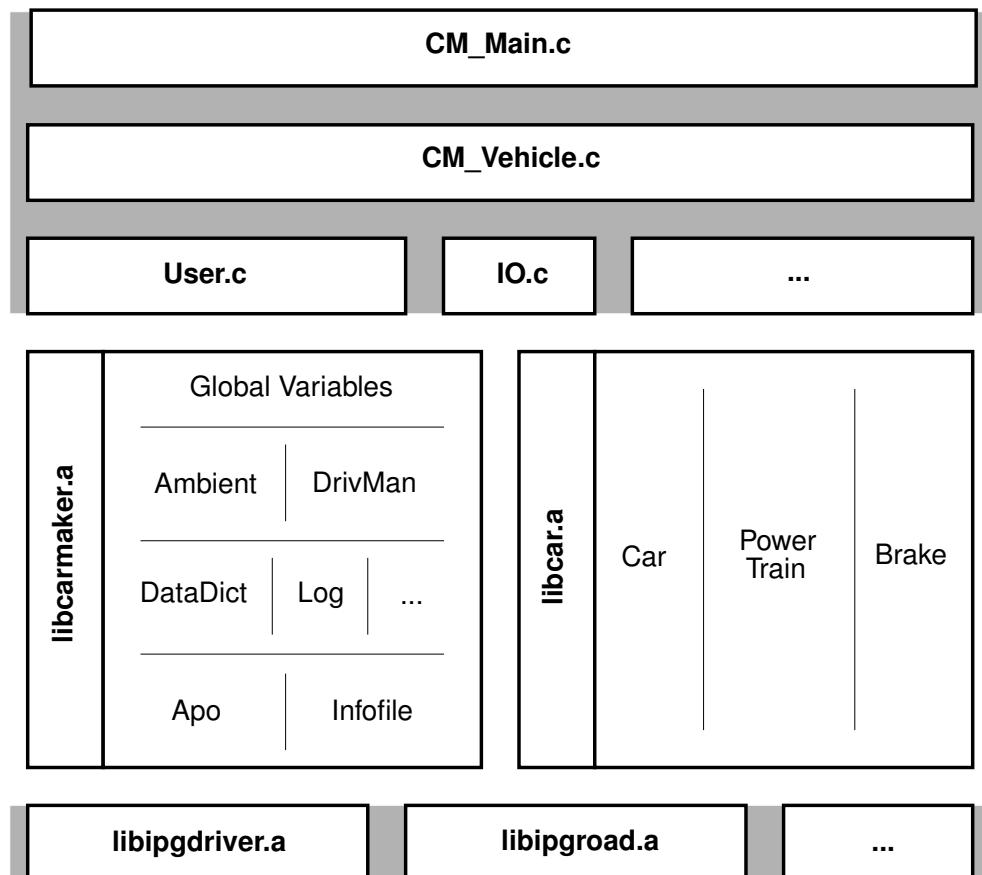


Figure 1.5: Modules of the CarMaker simulation program

1.2.1 User accessible modules

Here we find all the files the user has direct access to, because they are located in the project directories src subdirectory. The user may add its own modules by modifying the *Makefile* also found here.

CM_Main.c

This is the main module of the simulation program. It initializes the application and runs the application's main loop. Individual simulations are started and controlled in this module. All calls to functions in User.c, IO.c are done within this module, as well as almost all calls to functions of the vehicle module. Usually there is no need for the user to modify this module.

CM_Vehicle.c

This is the vehicle module of the simulation program. It represents the highlevel interface to the vehicle library. Normally there is no need for the user to modify this module.

User.c

This and the following module IO.c are the modules the user may tailor to the specific needs of his application. He is provided with a predefined set of functions that are called at different stages of the simulation and at different points during each simulation step. The provided functions have more or less empty bodies, they do nothing. The idea is that the user fills the bodies with his own code, adding application specific functionality to the CarMaker simulation program. Possible tasks the user might handle in User.c are the calculation of additional physical quantities, e.g. to be used IO.c (which should be kept clean of such calculations), or the integration of additional model code.

IO.c

This module is intended solely for the task of accessing application specific HIL hardware components.

The CarMaker library: libcarmaker.a

This library contains nearly all service modules of the simulation program. The code in CM_Main.c relies heavily on this library to control execution of the program and of the simulation. Initialization, data administration, file I/O, hardware I/O, communications and real-time services are handled by (not necessarily public) modules of this library. The library provides a lot of utility functions available to application specific code written by the user.

Global variables

Internal communication between modules inside the CarMaker simulation program is partly done using global struct variables. Three of them are central to the simulation: SimCore, Ambient and DrivMan. SimCore belongs to an internal module and contains data pertaining to control and overall functioning of the simulation program, i.e. most of its struct members do not represent physical quantities during a simulation. User code should therefore treat this variable as 'read only'. To get an idea what the Ambient and DrivMan structs might contain, read the description of the concerning modules of the same name below.

Environment

The Environment module provides information about ambient conditions of the CarMaker virtual simulation environment, e.g. temperature and wind. Road data is also available here. This information is kept in the Environment struct, which is mostly used by the vehicle module.

DrivMan

The DrivMan module is responsible of letting the virtual driver perform the maneuvers defined in TestRun. This is one of the most important modules inside the simulation program since its actions directly control the vehicle. The vehicle module should read the DrivMan struct to find out which direction the virtual driver wants the vehicle to go. The virtual driver in turn calls the vehicle module to find out about the current state of the vehicle, i.e. its position, velocity, etc.

DataDict

The DataDict module organizes all user accessible quantities into a central data dictionary. The dictionary contents are shared with the Apo module, providing online quantity access to CarMaker interface tools like IPGControl. The second task of the DataDict module is to record and write simulation results. The vehicle modules registers quantities, that might be of interest to others, in the data dictionary. Availability of certain physical quantities of the vehicle is also prerequisite to animation of the vehicle with IPGMovie.

Log

During a simulation, whenever something happens that the user should know about (e.g. most often some kind of error condition), information about the event should be recorded in the simulation's log file. The information is stored persistently so as to be available for later evaluation. The CarMaker GUI takes care of notifying the user about important events that appear in the log file. The log module has a set of functions that allow for writing formatted messages of different log levels into the logfile of the current simulation. The vehicle module uses this functionality to inform the user, if e.g. the vehicle during initialization encounters a critical variable's value to be completely out of range.

Apo

This is the communications module within CarMaker. It contains all the functions that a CarMaker application might need if it wants to communicate with other user interface programs over a network using Apo services. Most of the communication tasks using Apo are already handled inside libcarmaker.a. The user might want to add some Apo code to User.c, e.g. providing services to a user's interface tool written in the Tcl language. Normally, there's no need for the vehicle model to make use of the functionality offered by the Apo module.

Infofile

The infofile module contains functions that are used to access information stored in infofiles, e.g. CarMaker TestRun data and vehicle parameters in the *Data* subdirectory. The vehicle module uses this service to read vehicle parameters from file during initialization.

1.3 Rebuilding the CarMaker Simulation Program

Beside the CarMaker for Simulink approach after which the functionality of CarMaker can be extended using Simulink blocks, the C-code interface which is available for all CarMaker versions including CarMaker / HIL can be used to add functionality to the program.

- Plant models: used for components like the brake.
- Control models: used for control systems like ECUs.
- General purposes.

A special application which uses the C-code interface is the Model Manager module. For further information about this, have a look at [section 5.2 'Model Manager'](#).

1.3.1 Preparing your Project Folder

Extensions realized with C-code need the rebuilding of the CarMaker simulation program. To build a new CarMaker executable, the project directory needs to be updated, first. To update a project directory, select *File > Project Folder > Create / Update Project* in the CarMaker GUI. It must be ensured, that the *Sources / Build Environment* box in the *Included Features* area is checked.

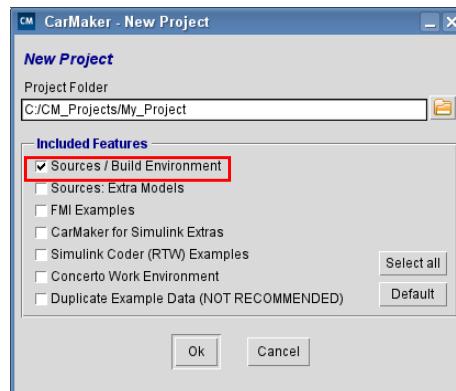


Figure 1.7: Creating a new project directory

This option creates a subfolder *src* in your project directory which contains, beside others, the following files:

- CM_Main.c
- CM_Vehicle.c
- IO.c
- IO.h
- Makefile

1.3.2 Building the new Executable

Using GCC

CarMaker is prepared to be rebuild using the GNU tool chain. The GNU tool chain is a blanket term for a collection of programming tools produced by the GNU Project (<http://gcc.gnu.org>). Its key component is the GNU Compiler Collection (GCC) which is a compiler system produced by the GNU Project supporting various programming languages, C and C++ are the relevant ones.

On Linux systems, the GNU tool chain is part of the system itself, on Windows it can be used via MinGW/MSYS.

MinGW, a contraction of "Minimalist GNU for Windows", is a minimalistic development environment for native Microsoft Windows and cross-hosted applications including:

- A port of the GNU Compiler Collection (GCC), including C and C++ compilers
- GNU Binutils for Windows (assembler, linker, archive manager, make tool)

MSYS, a contraction of "Minimal SYStem", is a Bourne Shell command line interpreter system. Offered as an alternative to Microsoft's cmd.exe, this provides a general purpose command line environment, which is particularly suited to use with MinGW. It includes a small selection of Unix tools (ls, ps, pwd, cd, etc.).

The MSYS version provided by IPG is a modified one and should be used instead of the version available on the Net. During the installation of CarMaker MSYS is installed on your machine automatically.

To start MSYS just use the shortcut Start/Programs/IPG/MSYS-2015.



Figure 1.8: Starting MSYS

The tools can be called up over the shell-console. To invoke the compiler you have to change directory to "src" within your project folder and call "make" to generate a new CarMaker executable.

```
[VMBRUCE-hk] 1) cd C:/CM_Projects/My_Project/src
/c/CM_Projects/My_Project/src
[VMBRUCE-src] 2) make
CC      CM_Main.o
CC      CM_Vehicle.o
CC      User.o
MK      app_tmp.c
CC      app_tmp.o
LD      CarMaker.win32.exe
[VMBRUCE-src] 3) ■
```

Figure 1.9: Executing makefile using MSYS

On Linux systems:

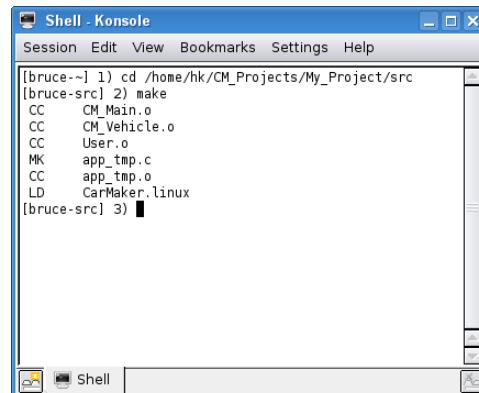


Figure 1.10: Executing makefile using the shell console



In case a 64-bit system is used, it is possible to compile a 64-bit executable with CarMaker version 5.0 onwards. Older CarMaker releases are 32-bit applications only.

However, with the default settings, only 32-bit executables are compiled. To create a 64-bit executable, the Makefile needs to be edited. The Makefile is located in the src folder of your CarMaker project directory and can be edited with any text editor (just check, that the editor does not convert tabs into spaces!). In the Makefile, the include path to the "MakeDefs" file needs to be changed:

- for 32-bit executable (default):


```
include /fs/opt/ipg/hil/<version>/include/MakeDefs
```
- for 64-bit executable (requires manual change):


```
include /fs/opt/ipg/hil/<version>/include/MakeDefs64
```

Using Microsoft Visual Studio

Alternatively, CarMaker can be rebuilt using Microsoft Visual Studio. In your *src* folder, you can find a configuration file *CarMaker.vcproj* which can be directly loaded in Microsoft Visual Studio. Depending on the used version, a conversion will be automatically done. After the conversion you are able to build the CarMaker executable through the standard way.

Please find a list of supported Microsoft Visual Studio versions in the Release Notes.

In case a 64-bit system is used, it is possible to compile a 64-bit executable with CarMaker version 5.0 onwards. Older CarMaker releases are 32-bit applications only.

However, with the default settings, only 32-bit executables are compiled. To create a 64-bit executable, simply change the architecture setting in the Visual Studio environment.

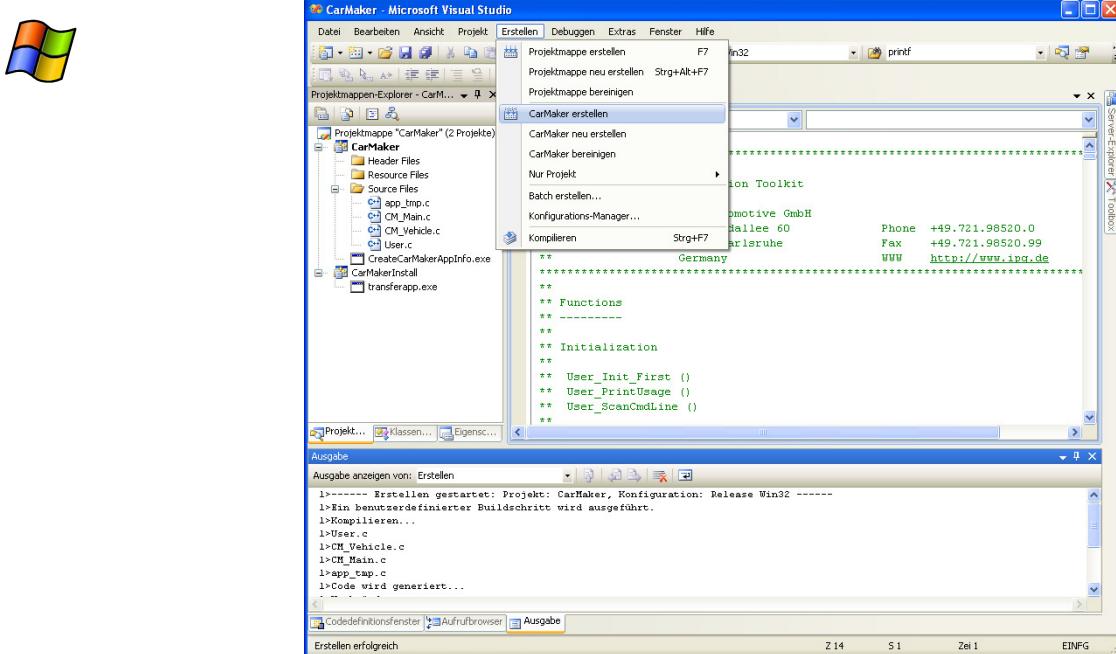


Figure 1.11: Using Microsoft Visual Studio

Using Eclipse

Starting with CarMaker version 4.5.4, the CarMaker executable can also be rebuilt using Eclipse IDE for C/C++. All necessary configuration files reside under the *src* folder namely as *makefile.defs* and *makefile.targets* along with *.settings* folder. These files are in concurrence to the Makefile already present in the same folder. Further information about the Eclipse version supported by your CarMaker version can be found in the Release Notes. Please note that you might have to update your Java version according to the Eclipse standards.

Eclipse can be started directly, alternatively also from CarMaker main GUI (File > Eclipse Workspace). The advantage of starting Eclipse from CarMaker is that the current workspace gets automatically loaded with files from the CarMaker *src* folder.



Please note, that the Eclipse Workspace option in the CarMaker main GUI is only accessible if an Eclipse installation is available on the machine. Eclipse IDE for C/C++ can be downloaded from <https://eclipse.org/>.

Further, the CarMaker project needs to comprise the build environment in terms of a *src* folder. In case this *src* folder is missing in your CarMaker project directory, you need to update the project (see section 1.3.1 'Preparing your Project Folder').

Before attempting to build the CarMaker executable please ensure that all environment settings of the respective Eclipse project are set correctly. For this right -click onto the project's root item in the *Project Explorer* window, choose *Properties* and change to the tab *C/C++ Build* > *Environment*. Especially Windows users should take care of the correct path to the local MSYS installation, specified in the environment variable *MSYS*. Once this is done, the application can be built the traditional way (*Project* > *Build Project*).



In case a 64-bit system is used, it is possible to compile a 64-bit executable with CarMaker version 5.0 onwards. Older CarMaker releases are 32-bit applications only.

However, with default settings, only 32-bit executables are compiled. To create a 64-bit executable, the “makefile.defs” needs to be edited. This Makefile is located in the src folder of your CarMaker project directory and can be edited with any text editor (just check, that the editor does not convert tabs into spaces!). In the Makefile, the include path to the “MakeDefs” file needs to be changed:

- for 32-bit executable (default):
include \$(CM_DIR)/include/MakeDefs
- for 64-bit executable (requires manual change):
include \$(CM_DIR)/include/MakeDefs64

Additionally on Windows systems it is essential to modify the *PATH* environment variable to allow a 64-bit build. For this open the project’s environment settings (see above) and modify the matching part of the *PATH* entry (leave everything other unchanged):

- for 32-bit executable (default):
...; \${MSYS}/mingw/bin; ...
- for 64-bit executable (requires manual change):
...; \${MSYS}/mingw64/bin; ...

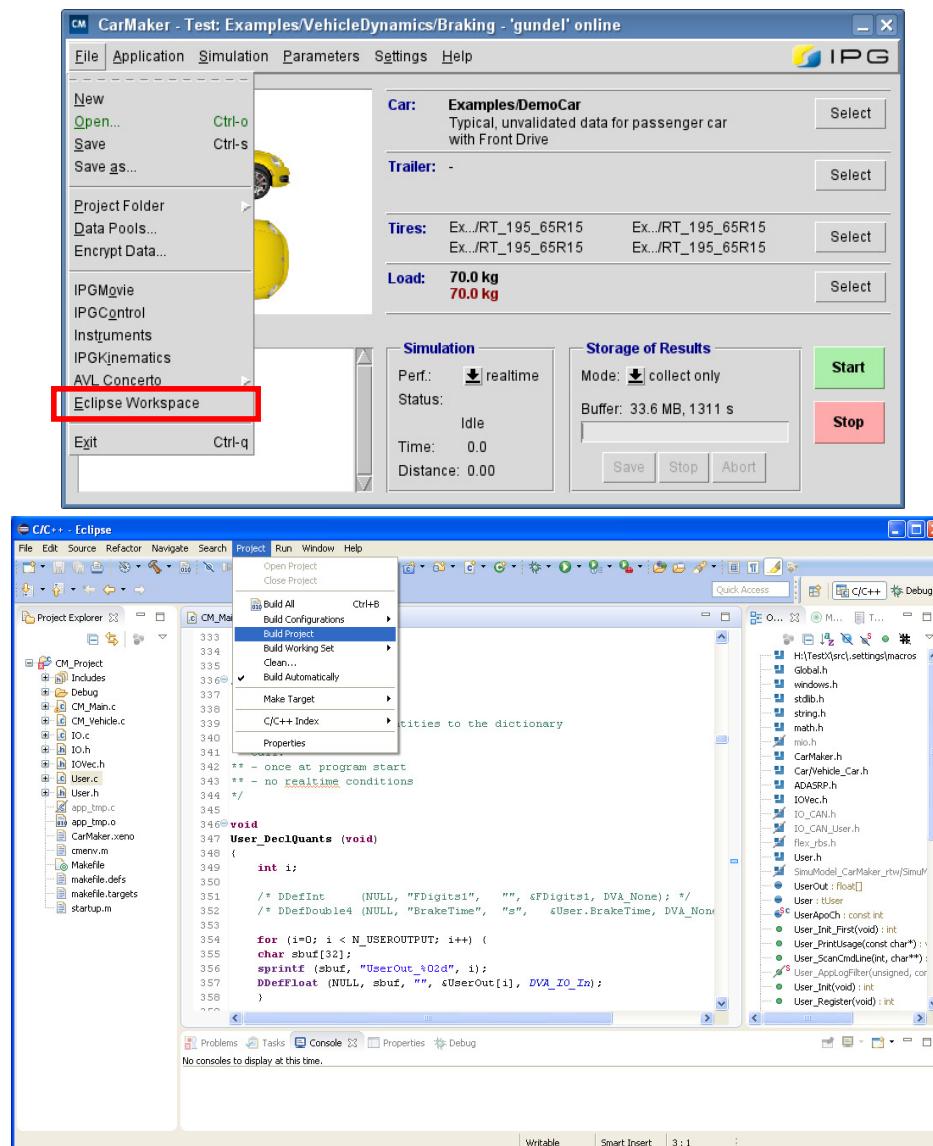


Figure 1.12: Using Eclipse IDE for C/C++

1.3.3 Using the new Executable

To use the compiled CarMaker executable, you have to select it within the CarMaker GUI by clicking on *Application > Configuration / Status*. The executable can be selected in the succeeding dialog.

1.3.4 Rebuilding the model library for CarMaker for Simulink

The part of CarMaker for Simulink, that does the actual numerical calculations for each time step, is called the model library. The model library is implemented as a separate module that can be extended and rebuilt by the user. Some applications might require rebuilding of the standard model library including user extensions, e.g. if C-code based model extensions should be made accessible in the CarMaker for Simulink environment, too.

A prebuilt model library for each supported Matlab version can be found in the CarMaker for Simulink installation directory. This is also the default library that is normally used as long as no custom library was built. Matlab's search path is used to determine which version of the model library is "the right one". See below for details.

The `src_cm4sl` directory contains the user accessible source code of the model library as well as a Makefile which controls the build process. Rebuilding the model library is a matter of invoking the `make` command in the `src_cm4sl` directory of your CarMaker project folder. The `make` command can be executed in an ordinary command shell under Linux. On Windows systems, the Microsoft Visual C studio, Ecplise or a cross-compiler such as MSYS can be used. Please find further information in [section 1.3.2 'Building the new Executable' on page 21](#). With executing the `make` command a new `libcarmaker4sl.mexglx` will be produced.



Please check the Matlab version (`MATSUPP_MATVER`) and installation path (`MAT_HOME`) stated in the *Makefile* before compiling. You might also have to delete the comment signs (#-character at the beginning of the line). You can edit the file with a simple text editor.

If a `libcarmaker4sl.*` is located in the `src_cm4sl` folder of the CarMaker project directory, it will be used in favor of the standard library from the CarMaker installation directory by your CarMaker for Simulink application. Regardless, the library currently used can be checked as described in the following.

1.3.5 Identifying which model library you are using

The CarMaker for Simulink model library can actually be invoked as a Matlab command. This can be used in two ways.

First, the model library is located using the Matlab search path. To find out about the location of the model library in use, at the Matlab prompt try Matlab's `which` command:

```
>> which libcarmaker4sl
/space/myproject/esp/src_cm4sl/libcarmaker4sl.mexglx
```

If you did not recompile the model library, the output will probably point to the library in the CarMaker for Simulink installation directory.

The search path itself is set with the `cmenv.m` script of your project directory. The script extends the search path to search your project's `src` directory first and CarMaker for Simulink's installation directory next. That is, a model library in your `src` directory is always preferred to the CarMaker for Simulink's default model library. You may inspect the current search path with Matlab's `path` command. Depending on how you organize your project source code it might make sense to edit `cmenv.m` to have a different configuration of the Matlab search path.

Second, when called directly, the model library gives you information about itself, e.g. when and how it was created. At the Matlab prompt, type the following command and check the output:

```
>> libcarmaker4sl

Application.Version      = Car_Generic X.X
Application.BuildVersion = 8
Application.CompileTime   = YYYY-MM-DD HH:MM:SS
Application.CompileBy     = fh
Application.CompileSystem = dagobert.ipg.de
Application.CompileFlags:
    -O3 -DNDEBUG -DLINUX -D_REENTRANT -Wall -Wimplicit
    -Wmissing-prototypes -D_GNU_SOURCE
    -DCM_SINGLE_THREADS -DCM4SL -fPIC -pthread
Application.Libs:
    libcar4sl.a
    libcarmaker4sl.a
    libipgroad.a
    libipgdriver.a
```

1.4 Debugging the CarMaker Simulation Program

During the development of user c-code extensions for CarMaker, there may occur situations where the CarMaker simulation program runs into a fundamental error (e.g. segmentation violation fault). To identify which part of the user code leads to this error, it may be useful to start the simulation program in a debugging mode. Thus it is possible to execute the program step by step and to observe at which step the problem comes up.

The CarMaker simulation program can be debugged using the GNU Project Debugger GDB. This chapter shows you how to prepare your CarMaker simulation program for debugging and gives an introduction to the basic functionalities of GDB.

1.4.1 Starting CarMaker in the Debug Mode

Before starting the actual debugging, the debug mode of the CarMaker simulation program needs to be activated. The *Makefile* already contains the compiler flag for activating the debug mode, but it is commented by default.

Listing 1.1: Activate debug mode in the Makefile

```
1: OPT_CFLAGS = -g -O1
```

Uncomment this line by removing the hash # and rebuild the CarMaker simulation program as described in [section 1.3 'Rebuilding the CarMaker Simulation Program'](#). Then start the new application with GDB for switching into debug mode:



On Windows systems:

Open Msys, change directory to "src" within your project folder and execute the new CarMaker.win32.exe with GDB.

```
[VMMAJA-vb] 1) cd C:/CM_Projects/My_Project/src
/c/CM_Projects/My_Project/src
[VMMAJA-src] 2) gdb CarMaker.win32.exe
GNU gdb 5.1.1 (mingw experimental)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or redistribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "mingw32"...
(gdb) [REDACTED]
```

Figure 1.13: Starting GDB using MSYS



On Linux Systems:

Open the shell, change directory to "src" within your project folder and execute the new CarMaker.linux with GDB.

```
[maja-~] 1) cd /home/vb/CM_Projects/My_Project/src
[maja-src] 2) gdb CarMaker.linux
GNU gdb 6.6.50.20070726-cvs
Copyright (C) 2007 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or redistribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i586-suse-linux"...
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) [REDACTED]
```

Figure 1.14: Starting GDB using the Linux shell

On Xpack4/Xeno systems:

First, you need to establish a connection with the realtime system. For this, open a command line and connect with telnet (Windows) or ssh (Linux). Once the login was successful, change directory to "src" within your project folder and execute the new CarMaker.xeno with GDB by typing "gdb CarMaker.xeno".

The following steps are equal for all systems. On Windows systems, GDB is operated via the Msys command window. On Linux or Xeno systems, GDB is operated via the command window that is connected with the realtime system.

After switching on the debug mode, run the CarMaker simulation program by typing "run". In case the application requires options during start, this can be appended with run command e.g.

```
run -io demoapp -constdt
```

Then start the CarMaker GUI and connect to the simulation program by selecting Application > Connect. Now you can select a TestRun and start a simulation the way you are used to.

1.4.2 Basic Functionalities of GDB

Now that the CarMaker simulation program is running in debug mode, some GDB functionalities such as setting break points can be used to identify the part of the C-code that causes the error. This chapter provides only a short introduction to GDB. After running the CarMaker simulation program in debug mode (as described in [section 1.4.1 'Starting CarMaker in the Debug Mode'](#)), the GDB commands can be set in the Msys command window on Windows systems or shell command window on Linux or Xeno systems. For further and more detailed information about GDB and its commands, please refer to GDB documentation (e.g. <https://sourceware.org/gdb/current/onlinedocs/gdb/index.html#Top>).

Stack trace The command "where" outputs the stack trace which provides information about the last operations that were executed before program crashed.

Set break points Break points can be used to stop the execution of the program at a certain point of the code to examine the program's state at this moment. Once the program is started press "Control + C" on GDB command line. Break points can be set by one of the following syntax:

```
break <function name>
break <source name>:<line number>
break <source name>:<function name>
```

Example

```
break User_In
break User.c:680
break User.c:User_In
```

After break point is set, the program can be continued using "continue" command. There is also an option of continue to stop only once at this break point and ignore it for the next <n> iterations:

```
continue [ignore-<n>]
```

Stepping The command "step" is used to execute the program line by line, stopping in between. There is also an option to do steps of <n> lines:

```
step <n>
```

1.4.3 Debugging CarMaker using Eclipse

It is also possible to debug the CarMaker executable using the extended debug functionalities provided by the Eclipse C/C++ IDE. This feature is currently supported for Linux 32/64bit, Windows 32/64bit and CarMaker/HIL XENO applications.

The following chapter gives a brief outline of the necessary configuration steps. Before starting your debug session ensure that the CarMaker executable is built with the build configuration *Debug*.

CarMaker/Office

Open the *Debug Configurations* dialog and create a new debug configuration of type *C/C++ Application*. On the *Main* tab insert the path to the executable you have just built. Most likely Windows users additionally have to change the path to the GDB debugger on the *Debugger* tab. Select the program *gdb.exe* in the *mingw/bin* or *mingw64/bin* subfolder of your MSYS installation directory. Now start your debugging session by pressing the *Debug* button. Eclipse will start your executable in the background and change to the debug perspective. After the application has been started and has not been stopped at a breakpoint you may connect to it with the CarMaker GUI by choosing *Connect* and start your simulation.

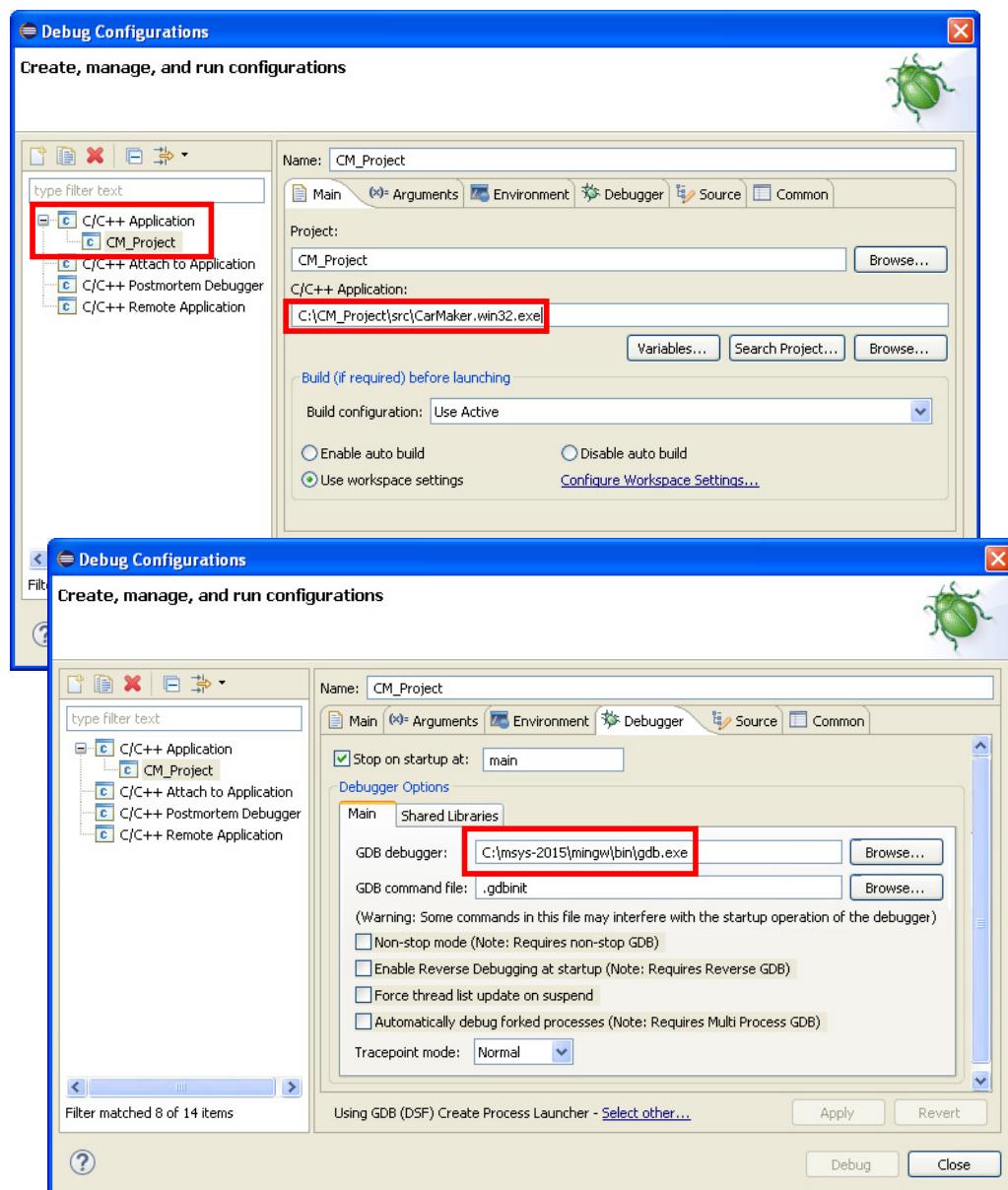


Figure 1.15: CarMaker/Office debug configuration in Eclipse C/C++ IDE

CarMaker/HIL

Debugging a CarMaker/HIL executable differs from the approach shown above since the CarMaker application and the Eclipse IDE run on different systems. To accomplish this task the GDB debugger on the host side is joined by a second program on the target side, the so called GDB server.

The GDB server must be started manually on the realtime system. To do so open a connection to the desired realtime system via telnet or ssh and change to your project directory. Then type:

```
gdbserver --once :2345 src/CarMaker.xeno -constdt
```

This tells GDB server to start the application *CarMaker.xeno* in the *src* directory and listen on TCP-Port 2345 for incoming commands from the GDB debugger which runs on the host. The CarMaker option *-constdt* disables cycle time measurement and prevents cycle time violations during debugging.

After starting the application on the target as shown it is time to create a matching debug configuration in Eclipse. Open the *Debug Configurations* dialog and create a new debug configuration of type *C/C++ Remote Application*. On the *Main* tab insert the path to the executable you have just started. Ensure that *GDB Manual Remote Debugging Launcher* is selected on the bottom of this tab. Windows users additionally have to change the path to the GDB debugger on the *Debugger* tab. Select the program *gdb.exe* in the *mingw/bin* sub-folder of your MSYS installation directory. After that specify the connection settings to the target: Enter the name or IP address of your realtime system and change the TCP port number to the same port you have chosen when starting the application with GDB Server.

After these changes start your debugging session by pressing the *Debug* button. Eclipse will connect to the realtime system you have specified and the GDB debugger and GDB server will talk constantly to each other in the background.

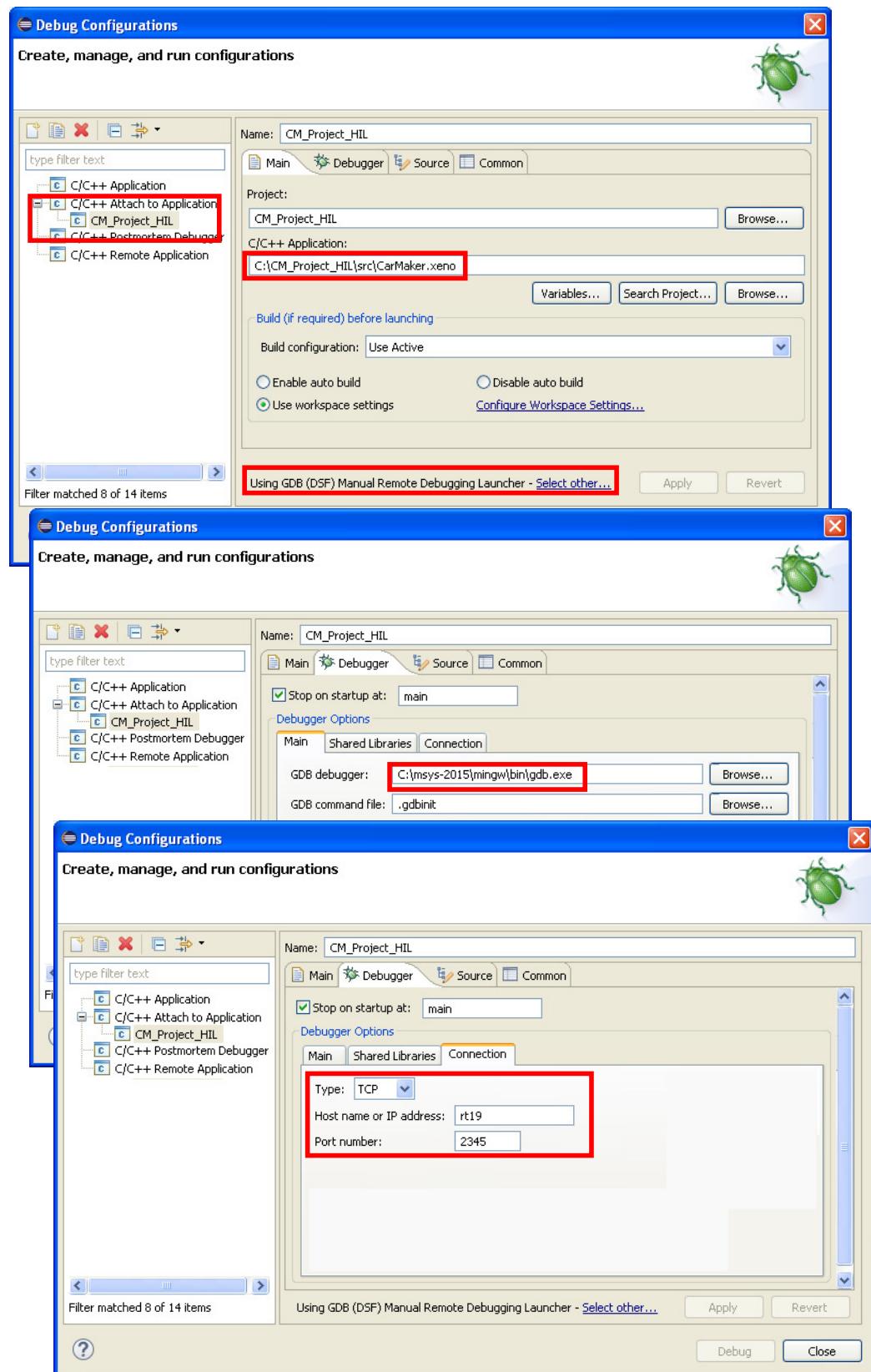


Figure 1.16: CarMaker/HIL debug configuration in Eclipse C/C++ IDE

1.5 CarMaker flow process charts

CarMaker is a multi-threaded application. Hardware I/O (if used) and most calculations are done in the realtime context of the main thread.

Helper Threads

Low Priority

Log

Output to stdout or Log File

DStore

Write Values of selected Variables to Results File

Main Thread

High Priority

Initialisation



IO_In

Read Input Signals from I/O Hardware

Calc

Calculation of all Models.
One Integration Step

IO_Out

Write Output Signals to I/O Hardware

APO

Communication with User Front End Tools.
GUI, IPGMovie, Instruments, ...

Cleanup

Write Output Signals to I/O Hardware

TestRun Start/Stop Thread

Very Low Priority

TestRun Start



Read TestRun

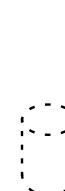
Model Initialisation & Parameterization



Preparation

Calculate Start Conditions

Operation Mode Simulation



TestRun Stop

Finish

Model Finish & Cleanup

Operating Mode Idle

Figure 1.17: CarMaker Flow Process Charts



Even if CarMaker is not running on a realtime computer the principals explained here are still the same.

Main thread restrictions

- Cycle time must always be less than the general sample rate (currently 1ms!)
- No file I/O
- No terminal output (printf...)
- No otherwise blocking operations
- Reading and generation of files is done in separate threads with lower priority.
CarMaker provided services: Log and DStore
- Parameter files are read in a separate “Testrun Start” thread.

1.5.1 Modified cycle clock generation if oversampling is used

Sometimes the cycletime of 1 ms of the main thread is too restrictive and one wants to have parts of the CarMaker application running with a faster sample time. This can be because the acquisition of some input or output signals should be done with a higher frequency than 1 kHz. Only those signals are processed in the oversampling thread. The oversampling rate can only be a multiple of 1ms.

Figure 1.18 shows how the oversampling thread triggers the main thread by sending an event every n-th cycle. This means that the clock timer moved from the main thread to the oversampling thread and triggers the clockless main thread.

Data is exchanged between the threads using data buffers containing the values of each oversample step. It is job of the main thread to process those values, e.g. through multiple calls (n times) of calculation models (with 1/n ms sample time) requiring or providing the data.

Oversampling Thread

Very High Priority

Main Thread

High Priority

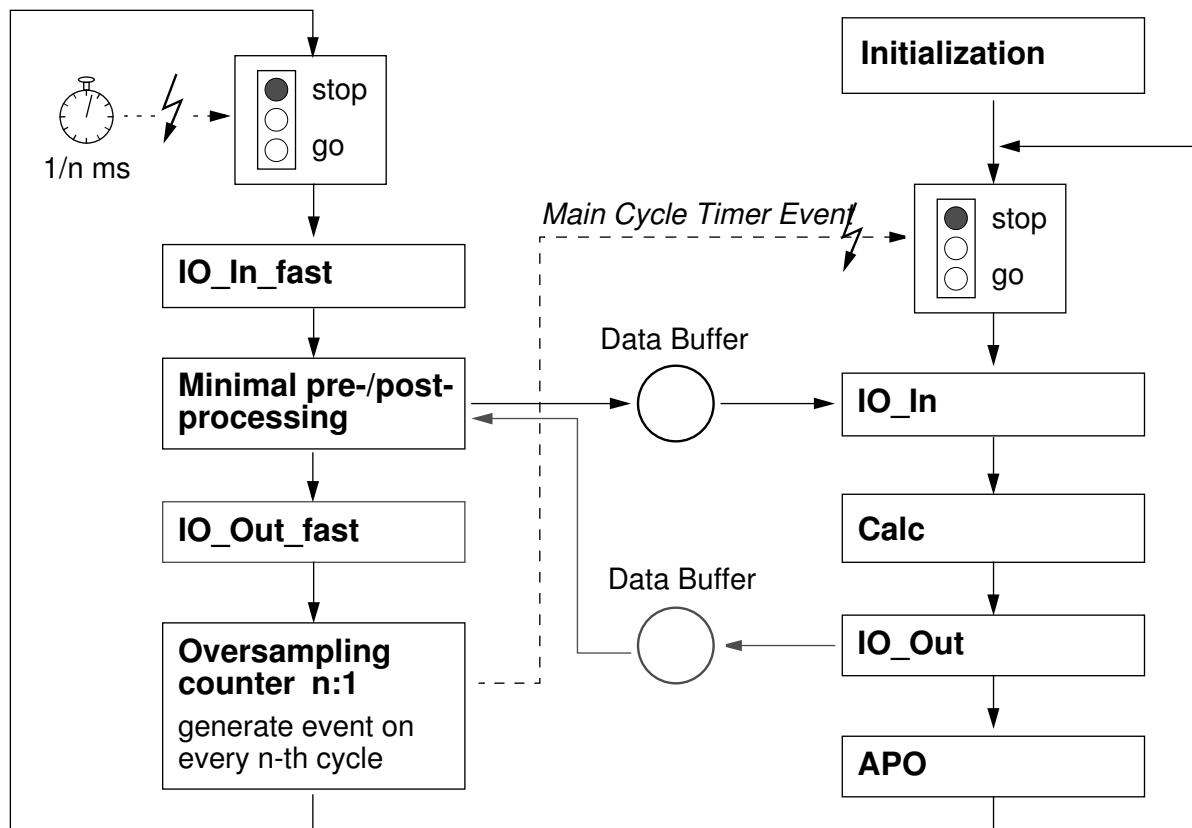


Figure 1.18: Interactions of oversampling thread and main thread

Figure 1.20 depicts the timeline of the oversampling procedure. Important is that the oversampling thread has higher priority than the main thread to ensure favoured execution.

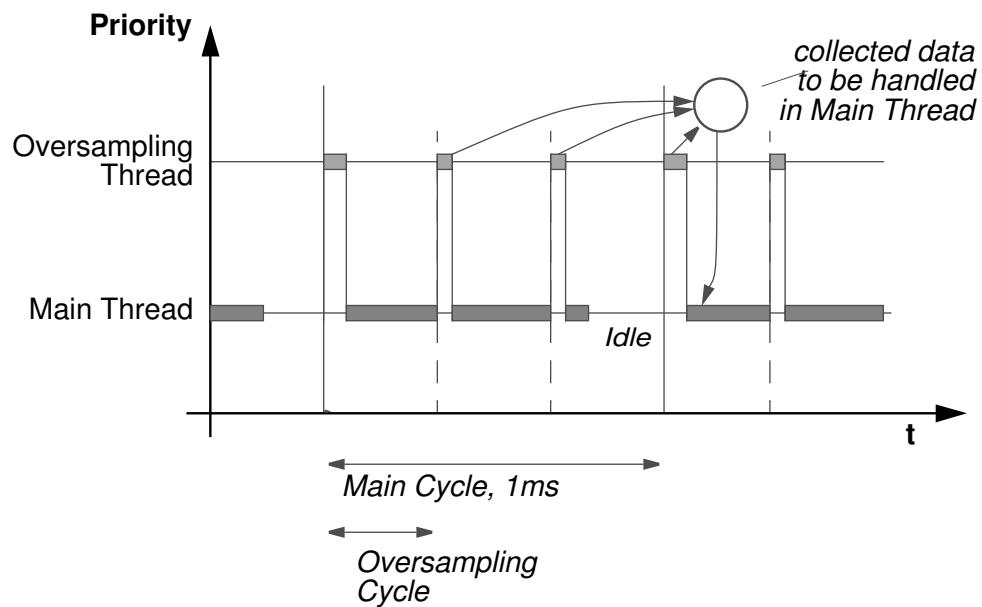


Figure 1.20: Timing Diagram

1.6 The main cycle explained

We now take a look at CM_Main.c, the module that controls execution of the CarMaker simulation program and represents the main cycle of CarMaker. It is important to get an idea of what is going on in the main cycle in order to understand which tasks the vehicle module has to perform, how the interface to the vehicle module is organized and how it works.

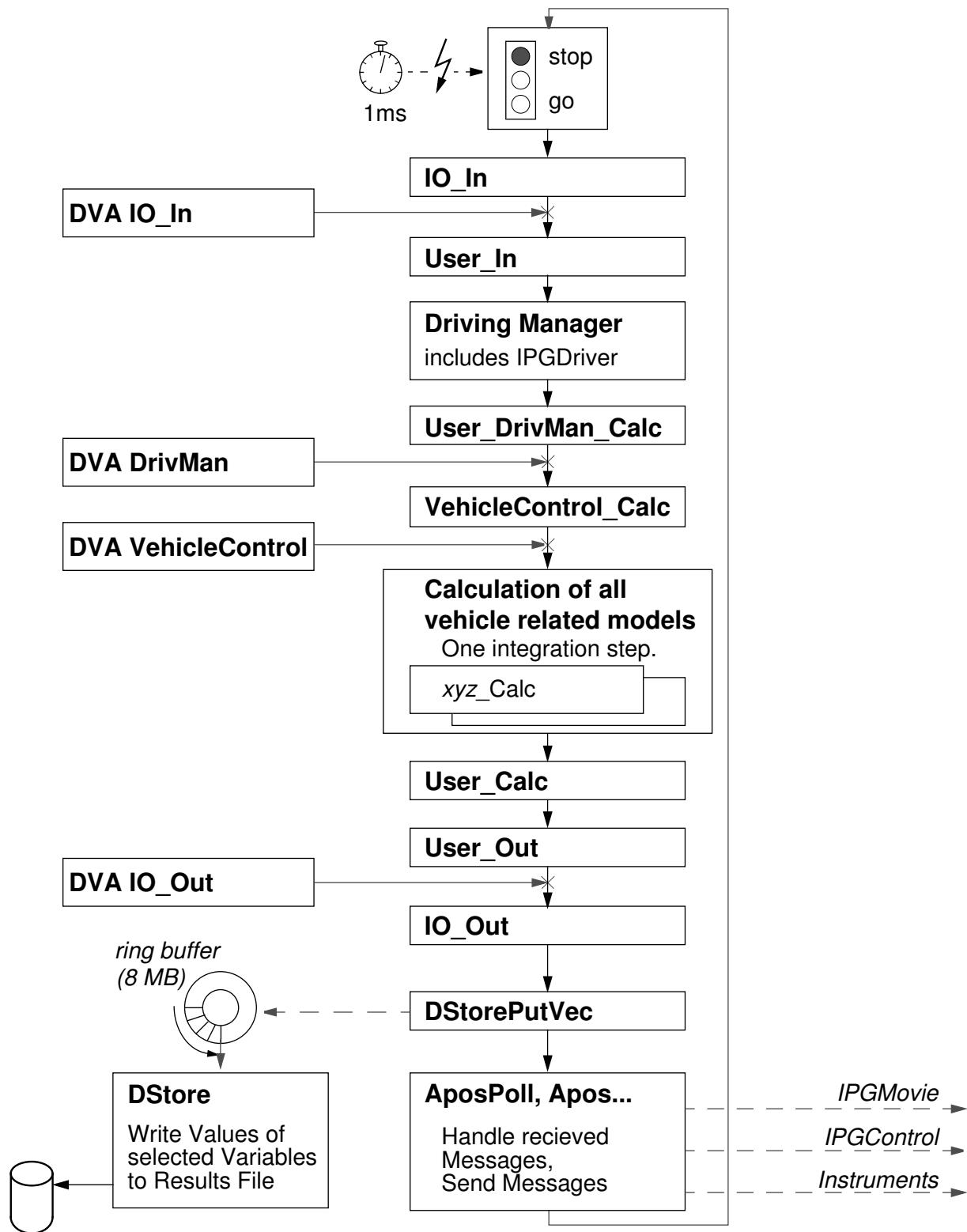


Figure 1.22: Main Cycle – Detailed View

1.6.1 An pseudo code excerpt from CM_Main.c

Note: The CASE statement is not considered to be “fall-through” like in C.

```

/* Things done only once. */
Setup all modules
Register static quantities of all modules in the data dictionary
Export current configuration of the simulation program

SimCore.State = 'Idle'

FOREVER {
    /* Do next 1 ms simulation step */

    Read hardware inputs

    SWITCH SimCore.State {
        CASE 'SimStart':
            Prepare the simulation
            Initialize DriVMan and vehicle module
            Calculate static equilibrium position of the vehicle
            IF everything is prepared THEN
                SimCore.State = 'Simulate'

        CASE 'Simulate':
            Perform DriVMan calculations
            Perform VehicleControl calculations
            Perform vehicle calculations
            IF end of TestRun THEN
                SimCore.State = 'SimStop'

        CASE 'SimStop':
            Run down the simulation
            IF everything is finished THEN
                SimCore.State = 'Idle'

        CASE 'Idle':
            Do nothing special
    }

    Put calculated vehicle quantities into interface variable

    Write hardware outputs

    IF SimCore.State == 'Simulate' THEN
        Store simulation results of current simulation step

    Perform Apo background communication tasks
    Read Apo messages from CarMaker interface tools
    IF message 'Start simulation!' received THEN
        SimCore.State = 'SimStart'
    ELSE IF message 'Stop simulation!' received THEN
        SimCore.State = 'SimStop'
    Send Apo messages to CarMaker interface tools
}

```

The pseudo code listing above gives you a simplified high-level picture of the main actions that go on inside CM_Main.c. A first look at the code reveals two important points about the program:

- The basic tasks of the main routine: Program setup, calculations, hardware I/O, storage of results, Apo communication.
- The event driven nature of the main routine.

We will now inspect each of these points more closely since they are crucial to an understanding of the vehicle module interface.

1.6.2 Basic tasks of the main routine

Program setup is done only once and before entering the event loop. This is the time for all modules to register any quantities in the data dictionary for later access by CarMaker interface tools using Apo services and for the storage of simulation results. Also, information about the current configuration of the simulation program and of its modules (i.e. when it was built, version numbers of libraries and modules, etc.) are gathered and exported to a file later read by the CarMaker GUI.

Calculations are split among the DrivMan and Vehicle Control module and the vehicle module. The term also include the tasks performed e.g. at the beginning and end of a TestRun simulation.

Hardware input/output is done before and after all calculations have been done. First the hardware is read, providing input for the calculations to be done next. After the calculations have been accomplished, the calculated values are output to the hardware. Of course this is only relevant for CarMaker/HIL. In case of CarMaker/Office, the called functions are empty.

Storage of results takes place after the calculations. A vector containing all values of interest of the current time step is put into a buffer, that gets written to a results file asynchronously by a separate background thread.

Apo communication is handled at the end of the event loop. It consists of three tasks. First, Apo must be given a chance to handle its internal communication tasks, e.g. answering an interface tool that likes to connect to the CarMaker simulation program. Next, the program must read and interpret all Apo messages that may have been sent by CarMaker interface tools since the beginning of the current simulation cycle. Third, there might be some messages the simulation program might want to send to connected interface tools itself.

Other tasks like reading input from file and direct variable access (DVA) are not shown in the pseudo code. They do not directly interfere with the tasks of the vehicle module and have been left out for the purpose of clarity.

1.6.3 The event loop of the main routine

The basic principle of the main routine is that it's event driven. In case of the CarMaker simulation program, this is just a short term for the following (abstract) behaviour:

- The program runs in an endless loop, the so called event loop.
- During each cycle the program is always in exactly one of several well-defined states. Each state has a certain, well-defined meaning, and depending on the current state some action is performed.
- During the cycle one or more events may occur, provoking some reaction of the program. One possible reaction is a change of the current state. If no event occurs, the current state does not change.

In theory, the simulation program should only be in one of two states: Either it is simulating a TestRun, or it isn't. In reality, finer control of the sequence of events requires more states to be used. Nevertheless, all states fall into one of the two categories mentioned: simulating or not simulating.

There are several kinds of events that can happen and that may cause a change of state in the simulation program. One possible event is some kind of error condition, e.g. the vehicle leaves the road during simulation or a connected hardware controller unit reports a problem with one of its sensors, so that the current TestRun should be aborted. Other events are of a more harmless nature, e.g. the preparation phase of a simulation is over and the program

should proceed with the actual simulation. A third kind of event are messages sent by CarMaker interface tools that are connected to the simulation program. Think of a user pressing the Start or Stop button of the CarMaker GUI. In this case the CarMaker GUI sends an Apo message to the simulation program, telling it to start or stop the simulation of the current TestRun.

When the program is not simulating a TestRun, it is in state **Idle**. We may call it the default state of the program. Immediately after initialization and between TestRuns the program is in Idle state. This does not imply that the program is doing nothing. Maybe a good circumscription of the actions performed in this state is “simulation of a vehicle standing still”.

The three states pertaining to the simulation of a TestRun are **SimStart**, **Simulate** and **SimStop**.

If a simulation is started, the program first enters state **SimStart**. This state is mainly associated with preparations at the beginning of the simulation. E.g. the vehicle reads its current parameters from file and tries to find its static equilibrium position. Maybe ignition is turned on, the engine is started, and so on.

If all preparations are done (and everything is ok), the current state changes to **Simulate**. Now the program is really performing the simulation of the TestRun. It normally remains in this state until the TestRun ends or is aborted because of some error condition or manual intervention of the user.

To end a simulation, the simulation program changes its state to **SimStop**. The idea of this state is to run down the simulation gracefully and bring the vehicle to a stop. When this has been accomplished, the program automatically changes to state Idle.

1.7 Using CarMaker/HIL with ETAS Hardware

This chapter describes CarMaker running on an ETAS RTPC (Realtime PC) in conjunction with LABCAR OPERATOR.

1.7.1 System Overview

[Figure 1.23](#) shows an overview of the complete system. On the one hand there is the host PC where the graphical user interfaces of all involved software components are running and the user interaction takes place.

On the other hand you will find the so called RTPC, where the models are running in real-time. I/O hardware constitutes the third component of the system. Both the host PC and the I/O hardware board are connected to the RTPC via ethernet.

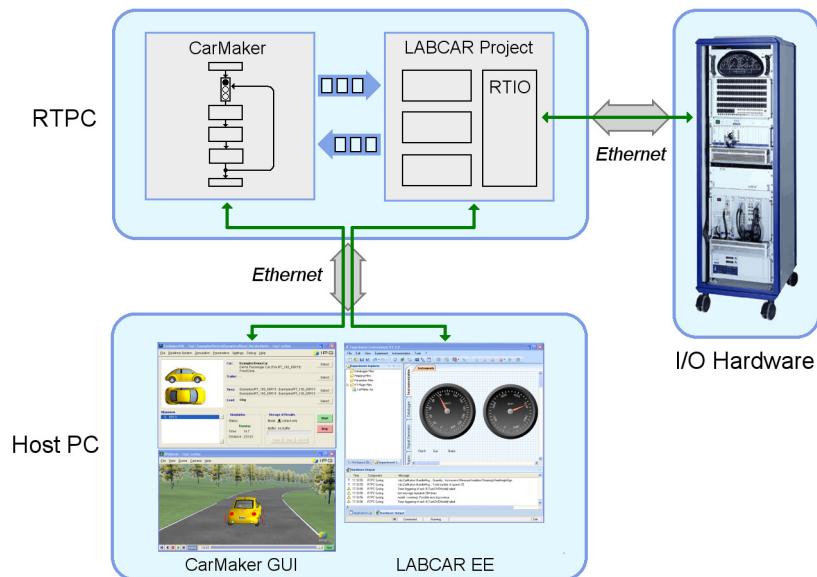


Figure 1.23: ETAS System Overview

Tools to work with

When using CarMaker on the ETAS RTPC the user is confronted with several software tools, each with its distinct purpose: Part of the LABCAR OPERATOR software suite are besides others LABCAR RTIO, LABCAR IP and LABCAR EE.

LABCAR RTIO is used for the configuration of ETAS hardware components. In conjunction with LABCAR IP (Integration Platform) the RTIO Editor is used to integrate software models and attached hardware to a combined project, the so called LABCAR IP project. The third tool LABCAR EE (Experiment Environment) is used for simulation on the RTPC as well as for visualization on the host PC.

For further information and an in-depth explanation of all components please refer to the official ETAS LABCAR documentation.

Ways of communication

When simulating with CarMaker on the ETAS RTPC, both the CarMaker GUI and LABCAR EE are running on the host PC providing the full capabilities and functionalities the user is used to. Both GUIs communicate with their respective simulation backends on the RTPC via the existing ethernet connection: On the RTPC the CarMaker simulation program and the LABCAR IP model are running besides each other. Data is exchanged between them

by using interprocess communication facilities of the operating system. Access to I/O hardware is managed primarily on the ETAS side but capabilities to address all in- and outputs from the CarMaker simulation program are provided. Basically all user accessible quantities defined in CarMaker are accessible from the LABCAR model, and all LABCAR variables including all in- and outports are accessible from CarMaker.

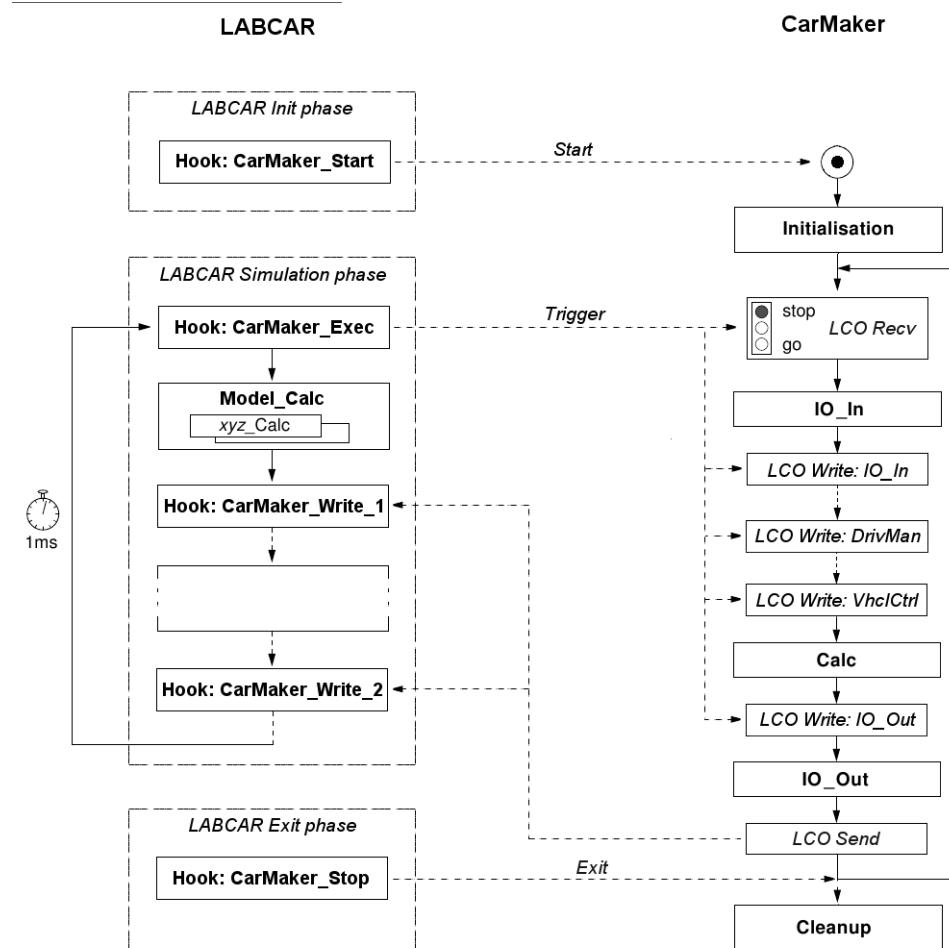


Figure 1.24: Flow Process of CarMaker and ETAS

1.7.2 Realtime System Setup

Perform the Realtime System Setup as described in chapter 3 of the CarMaker Installation Guide.

Here are some additional remarks regarding the Realtime System Setup:

- For integrating ETAS LABCAR RTPC the settings of the NFS server have to be adjusted as follows: In the Realtime System Setup GUI click on the "Information" icon of the NFS Server Network service (shown as a white "I" on blue ground). In the next window click on the button "Expert Settings" for opening the "haneWinNFS Server" dialog. In this dialog stay on the first tab "NFS" and ensure that only NFS Server Protocol Version 2 is activated. Uncheck Version 3 if it has been activated. Confirm your changes by clicking "OK" and close the dialog.
- As "Net Interface" choose the ip address assigned to your network interface which is connected to the RTPC (mostly "192.168.40.xxx", replace "xxx" accordingly).

When adding your ETAS LABCAR RTPC to the list of Realtime systems by clicking on the "+" button, enter the name of your RTPC (usually "rtpc"), the IP address (usually "192.168.40.14") the MAC address of the RTPC and select "XENO / x86" as OS/CPU type. If you do not know the MAC address of your RTPC boot the RTPC, open the Internet Explorer and enter "http://192.168.40.14" in the address field of the Internet Explorer. In the upcoming RTPC web interface select the last link "Power Control" and copy the MAC address from the last line of the page.

When the installation is done, It is useful to expand the shortcut to the CarMaker GUI with the option "-apphost -rtpc". To realize this, open up the properties of the shortcut and insert the options at the end of the target-line.

1.7.3 Preparing your project

Whether you want to start a new project from scratch or to integrate CarMaker into an existing LABCAR project, the basic steps are the same in both cases.

The CarMaker-LABCAR interface strongly depends on the concept of the so called Real-Time Plugins (RT Plugins). A RT Plugin contains user-defined C-Code which is loaded dynamically during runtime and executed at an arbitrary point of your LABCAR model.

Recommended project structure

Place your complete LABCAR project in a subfolder of the "/CM_Projects" directory. We recommend to create the CarMaker project directory within this LABCAR project directory as a direct subfolder and name it "CM_Env". This will ensure that all project specific data is placed within one common folder.

Placing the hooks

To use a RT Plugin, a so called hook has to be placed at a suitable position within the LABCAR process list. This hook defines the calling context of the RT Plugin. For defining a new hook, open the LABCAR Integration Platform (IP), load your project and change to the tab "OS Configuration". To add a hook between two processes, right-click at the appropriate position within the task list and select "Add Hook" (see [Figure 1.25 on page 45](#)).

The name of the hook can be an arbitrary string but has to be unique within the whole LABCAR project. For further description of the underlying concept of hooks and RT Plugins have a look at the official ETAS LABCAR and RTPC documentation.

See the next section for the hooks needed to execute the communication with CarMaker.

Necessary modifications

The CarMaker installation package comes with a ready-to-use RT plugin which is required for the communication between the CarMaker simulation program and the LABCAR model. The RT Plugin is located in the CarMaker installation directory under *Misc-labcar/rt_plugin*.

In this folder you will find the RT-plugin and the source code files "CarMaker_hooks.h", "CarMaker_main.c" and "CarMaker_main.h".

Here is a excerpt of the file:

```
/* path to CarMaker executable relative to CM_Projects directory */
static const char *cm_directory = "Projectfolder/bin";

/* name of CarMaker executable to run */
static const char *cm_executable = "CarMaker.labcar";

/* Name of hook where CarMaker start is performed - must be during Labcar Init phase */
static const char *cm_init_hook = "CarMaker_Start";
```

```

/* Name of hook where CarMaker stop is performed - must be during Labcar Exit phase */
static const char *cm_exit_hook = "CarMaker_Stop";

/* Name of hook where communication with CarMaker is handled, must be called every 0.001 sec */
static const char *cm_exec_hook = "CarMaker_Exec";

/* Names of hooks where Labcar quantities should be written during model simulation */
static const char *cm_write_hooks [MAX_HOOKS] = {
    "CarMaker_Hook_1", "CarMaker_Hook_2", NULL
};

```

You have to make some project specific changes within this file, which are explained in the following steps.

The entry "cm_directory" defines the path to the CarMaker executable relative to the mounted directory "/CM_Projects".

The entry "cm_executable" defines the name of the CarMaker executable to be used for simulation.

Cooperation between CarMaker and LABCAR

A basic co simulation between CarMaker and LABCAR requires three defined hooks. The first one is the so called init hook, where LABCAR starts the CarMaker simulation program and initializes the communication. This hook must be placed within a LABCAR init task to be called during simulation startup. The hook name defined in the entry "cm_init_hook" and the name of the hook in the process list of LABCAR IP must be the same.

Furthermore, a corresponding exit hook for terminating the CarMaker simulation program and closure of communication is required. This hook is typically placed within a LABCAR exit task which is called at the end of each simulation. The entry "cm_exit_hook" defines the name of the exit hook. Again it must be the same one as defined in LABCAR IP.

The basic communication is handled within the third hook defined by "cm_exec_hook", which may be placed anywhere in the LABCAR process list. It must be ensured that it is called every 0,001 seconds, since the outgoing communication from LABCAR triggers the CarMaker simulation cycle which is forced to be exactly 0,001 second. Otherwise, CarMaker will report cycletimes overruns.

Writing from CarMaker to LABCAR

When writing from CarMaker to LABCAR quantities, the time of the write operation may be essential because it may be necessary to overwrite values already calculated by a LABCAR module with other values of CarMaker.

In order to fulfill this task, the user can define an arbitrary number of additional write hooks. The list of potential write hooks is defined by "cm_write_hooks".

A write operation on a LABCAR quantity requires the existence of a corresponding hook where the operation is performed. Within one hook, multiple LABCAR quantities may be overwritten.

Example Assume that a value "v_lco" calculated by LABCAR should be overwritten by another value "v_cm" calculated by CarMaker. If "v_lco" is calculated by process "P_1" and is used by process "P2" of the LABCAR model afterwards, the definition of a write hook between processes "P_1" and "P_2" is essential to replace "v_lco" by "v_cm" in the correct context.

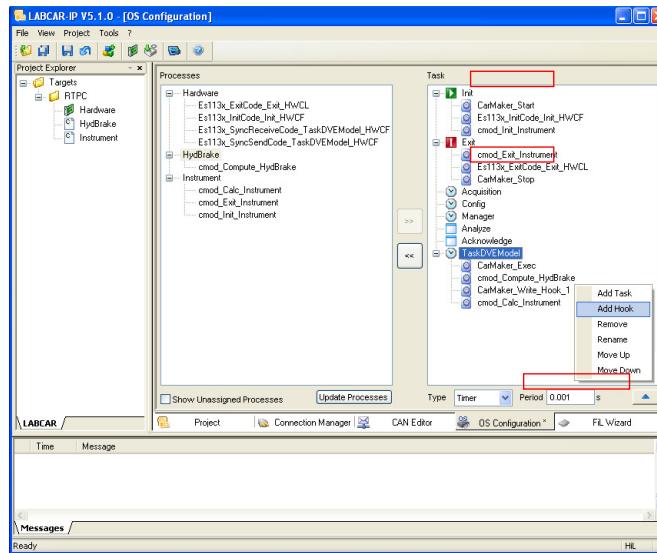


Figure 1.25: Placing hooks in LABCAR Integration Platform (IP)

After the file "CarMaker_hooks.h" is modified according to your needs, start the RT Plugin Builder which comes with your LABCAR installation from the start menu, open the CarMaker RT Plugin and compile it by clicking on *Build*. After that the RT Plugin is ready to use.

Writing from LABCAR to CarMaker

CarMaker is much more complex than LABCAR, only a few key values are available, where quantities can be overwritten. They are called IO-In, DM, VC, IO-Out. For further explanations, see chapter [section 17.4.11 'Direct Variable Access \(DVA\)' on page 703](#).

Defining the static mapping

When preparing your project you have the possibility to define a static mapping between CarMaker and LABCAR quantities. The mapping is defined within a XML file, which is commonly named "cm_lco_mapping.xml" and looks as follows:

```
<Mappings>
  <Mapping Direction="CM2LCO">
    <CM Name="DM.Brake"/>
    <LCO Name="Instrument/MeasureVariables/BrakePedal" Hook="CarMaker_Write_Hook_1"/>
  </Mapping>
  <Mapping Direction="CM2LCO">
    <CM Name="DM.Gas"/>
    <LCO Name="Instrument/MeasureVariables/GasPedal" Hook="CarMaker_Write_Hook_1"/>
  </Mapping>
  <Mapping Direction="CM2LCO">
    <CM Name="Car.v"/>
    <LCO Name="Instrument/MeasureVariables/Speed" Hook="CarMaker_Write_Hook_2"/>
  </Mapping>
  ...
  <Mapping Direction="LC02CM">
    <CM Name="User.SteeringMode" Hook="IO_In"/>
    <LCO Name="Instrument/CalibrationVariables/SteerMode"/>
  </Mapping>
  <Mapping Direction="LC02CM">
    <CM Name="User.SteeringAngle" Hook="DM"/>
    <LCO Name="Instrument/CalibrationVariables/SteerAngle"/>
  </Mapping>
  ...
</Mappings>
```

Besides the fact that the syntax is quite self-explanatory, here are some further notes:

Each <Mapping> block defines exactly one mapping of two related quantities in both worlds. The attribute "Direction" defines in which direction the quantity is exchanged. Writing from CarMaker to a LABCAR quantity is indicated by the key "CM2LCO", from LABCAR to CarMaker by "LCO2CM".

The <CM...> part describes the quantity on CarMaker side, "Name" must be the name of a valid User Accessible Quantity. If the value is transferred from LABCAR to CarMaker, the write place within the CarMaker simulation cycle can be defined by the optional attribute "Hook". Possible Values are "IO_In", "IO_Out", "DM" and "VC". If "Hook" is not specified, CarMaker will choose a suitable position for the write access automatically.

Analogous, the block <LCO...> defines the related quantity within the LABCAR model. "Name" meets the full A2L label as it is shown in the "Workspace Elements" list within the LABCAR Experiment Environment (EE). When writing from CarMaker to LABCAR, the corresponding hook must be defined by the mandatory attribute "Hook" as described further above.

The location of the mapping file is specified by the SimParameter entry "LABCAR.Mapping-File". The path must be relative to the current project directory. If you do not specify "LABCAR.MappingFile" and followed the above advice regarding the common project structure CarMaker will try to find a mapping file named "cm_lco_mapping.xml" in the uppermost LABCAR project directory (one level above the carmaker project directory "CM_Env").

1.7.4 Starting the Simulation

After you have prepared your project as described above, you are ready to start a simulation. For a simulation LABCAR and CarMaker GUIs are required. Open the CarMaker GUI and load your model and TestRun. Start LABCAR EE, load the prepared LABCAR experiment and perform the download of the model to the RTPC by clicking on *Experiment > Download*.

Before starting the simulation, ensure that the correct CarMaker RT Plugin is included in your LABCAR project by changing to the "Experiment Explorer" and checking the active RT Plugin files. If it is still missing, add the CarMaker RT Plugin by right-clicking on the "RT Plugin Files" item and selecting "Add file". Change to the location where the RT Plugin is stored and select the file "CarMaker.rtp".

After the start of the simulation in LABCAR EE, select "Application Connect" to connect the CarMaker GUI to the running CarMaker simulation program on the RTPC and to start the simulation.

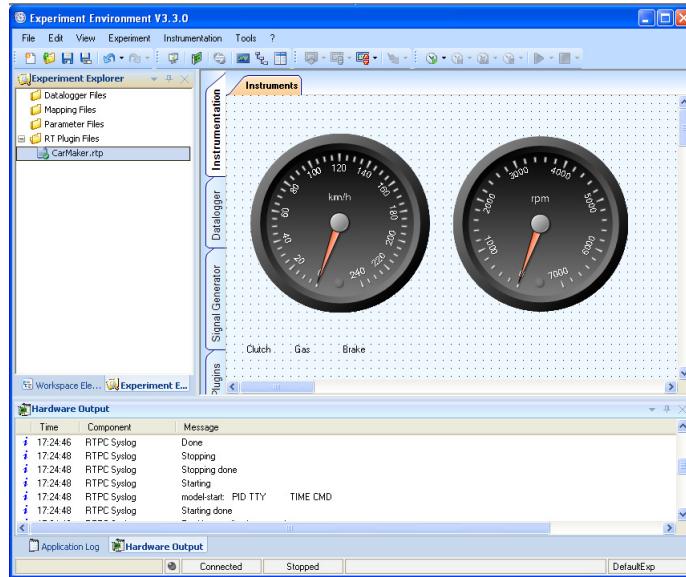


Figure 1.26: LABCAR Experiment Environment (EE) with loaded CarMaker RT Plugin

Addressing LABCAR quantities

Besides the possibility to define a static mapping between CarMaker and LABCAR quantities, it is also possible to address any LABCAR quantity from CarMaker within a user defined Realtime Expression. Just use the LABCAR A2L label of the desired quantity as you would use a CarMaker User Accessible Quantity. The only thing to remind is that a "/" within a LABCAR A2L label must be replaced by a ":".

Example Instrument/MeasureVariables/SteeringWheelAngle
must be replaced by
Instrument.MeasureVariables.SteeringWheelAngle



To copy the full A2L label out of LABCAR EE, right-click on the appropriate item in the "Workspace Elements" view and select "Copy label to Clipboard". To paste the copied A2L label into the chosen CarMaker Realtime Expression entry field, just press **Ctrl+V**.

If the Realtime Expression specifies a write access to the selected LABCAR quantity, the specification of an appropriate hook is essential to define the correct context within the LABCAR model. The hook is simply defined by appending "::" followed by the hook name to the A2L label.

Example Eval Instrument.CalibrationVariable.SteeringMode::CarMaker_Write_Hook_1=1

1.8 Using CarMaker/HIL with dSPACE Hardware

1.8.1 IPG Realtime Services

Native dSPACE models have no possibility to use high-level operating system functions like reading or writing files or the usage of network protocols. To enable the user to proceed as normal while using CarMaker on dSPACE systems, the IPGRT service was created.

IPGRT runs on the host-PC to operate the file-requests of the realtime-PC, the network communication with the CarMaker GUI, IPGControl and so on.

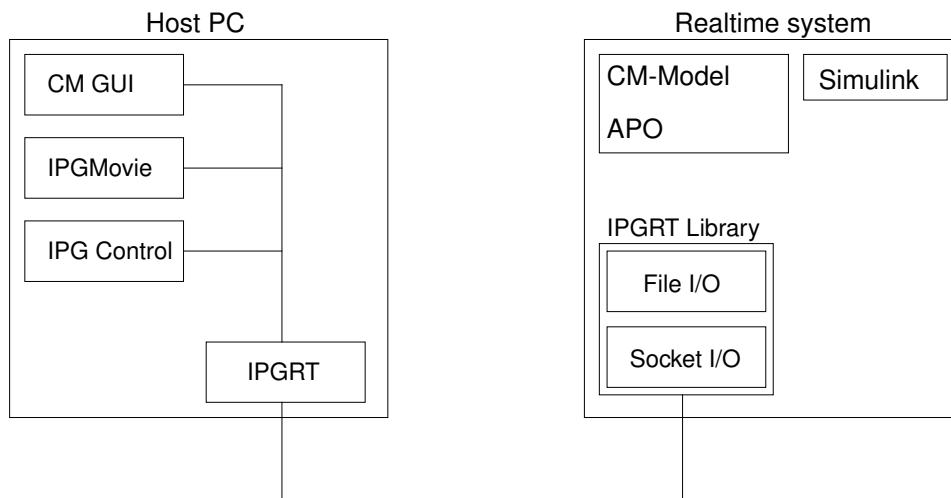


Figure 1.27: Setup of host PC and realtime system

Using CarMaker with Bus Connection

CarMaker is communicating with the realtime system via the bus connection by default. You do not need to change anything, if you want to use this type of communication.

Using CarMaker with network connection

There are some steps required to establish a network connection between CarMaker and the realtime system. Open up your CarMaker installation folder, which you will usually find under `C:\IPG\HIL\<CM-Version>\win32-ds\<Board_No>\bin`. The next step is to rename the `ipgrt.exe` into `ipgrtb.exe`. When this is done, rename the `ipgrtn.exe` into `ipgrt.exe`.

Now, CarMaker is ready to communicate to the realtime system via ethernet.

1.8.2 Troubleshooting

Check IPGRT Log

An easy way to get information about the condition of the realtime system and to call up the descriptions of the last errors which occurred during the simulation, is to select *Simulation > Session log* in the CarMaker GUI. Use the right mouse button to click into the window and select *Check IPGRT Log*. In this dialog, only the last few entries of the debug console will be displayed.

For the majority of users, this option is fully adequate. Another possibility is to use the IPGRT Debug Console, which is described in the following paragraph.

IPGRT Debug Console

This option is suitable for experienced users. The debug console offers the possibility to get access to the IPGRT. Compared to the error messages of Controldesk, the Debug Console is a more detailed, extended opportunity to communicate with IPGRT. It establishes a connection to the IPGRT service via telnet, which enables you to use a lot of commands. Furthermore, the debugging will be much easier.

The commands can be sent directly to the realtime-PC. A list of available commands can be displayed by entering “help”. The debug console is very useful if problems appear, for example a crash of a model.

1.8.3 Extending supplied models by your own ones

Preparing your model

Before the supplied models can be extended, there are a few things attention should be paid to. If not done already, you have to create a project directory with the option “Sources / Build Environment” activated. This project directory needs to be opened with Matlab to load the *generic.mdl* model. If this is done, you have to select the suitable system target file. For further information on this, have a look at section “CarMaker on dSPACE Systems” of the UsersGuide. Before extending your model it is recommended to save it under a different filename which ensures a high recognition value.

Extending your model

To extend your model, you can use any dSPACE blockset, for example for programming I/O. For further information on this, have a look at the relating dSPACE documentation.

Another possibility is to route I/O signals and other signals from your own model-extensions into CarMaker. For further information about this, have a look at [section ‘CarMaker for Simulink’ on page 92](#).

After your model is extended, a new CarMaker executable needs to be generated. After generating the new executable, two files will be generated in the working directory: *<model name>_usr.c* and *<model name>_user.mk*. These files build a further interface for user-defined customizing on C-code level.



The variable CM_SRC_DIR in the file *<model name>_user.mk* defines the path, where project specific source files of CarMaker (*User.c*, *User.h*, *IO.c*, *IO.h*, *CM_Main.c*, *CM_Vehicle.c*) are located.

1.9 Using CarMaker/HIL with National Instruments Hardware

This chapter describes CarMaker running on a National Instruments PXI(e) realtime system in conjunction with NI LabVIEW or NI VeriStand software.

1.9.1 System Overview

There are two approaches to use CarMaker with hardware from National Instruments:

First CarMaker can be integrated into National Instrument's LabVIEW toolkit as part of a LabVIEW project which follows certain conventions. Utilizing CarMaker this way allows the integration of (sub-)models or hardware drivers written in LabVIEW's wide-spread G-code into the CarMaker simulation environment.

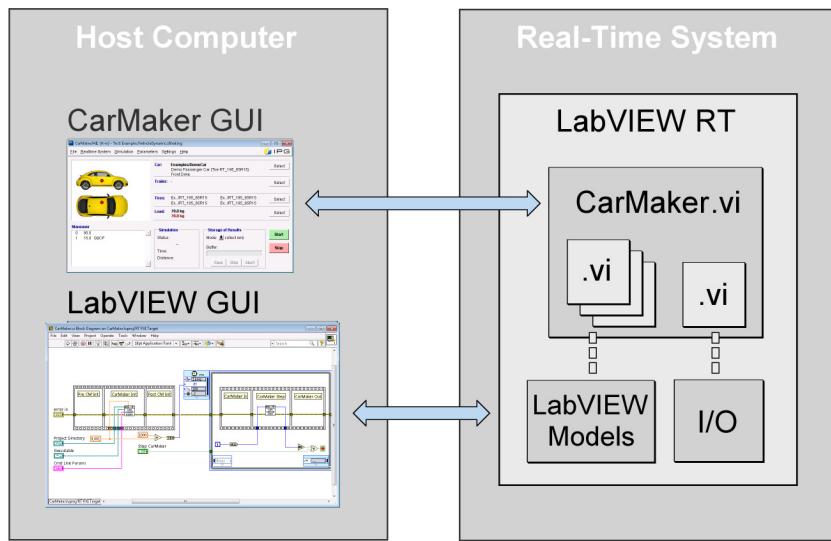


Figure 1.29: CarMaker with NI LabVIEW - System overview

Secondly CarMaker can be part of a VeriStand project by integrating it into the VeriStand system as a so-called Custom Device. This allows fast and easy connection of models from different simulation environments on a higher-level basis with CarMaker as the central integration platform.

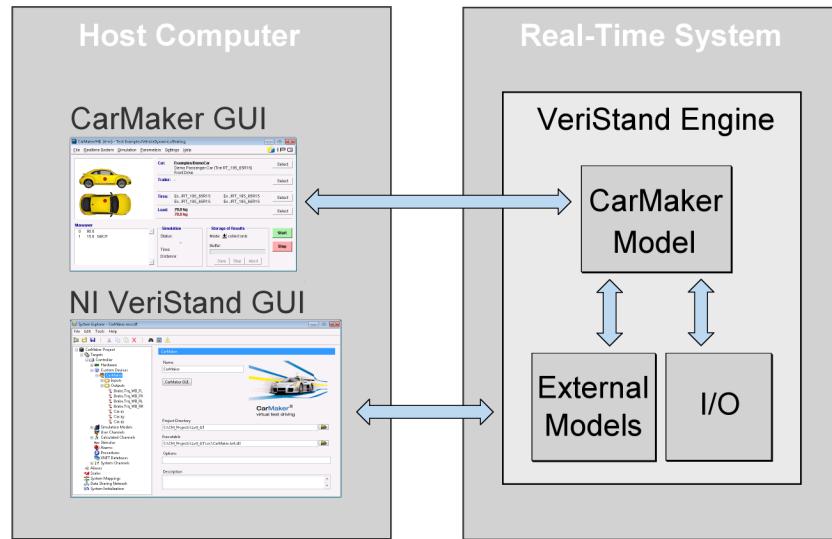


Figure 1.30: CarMaker with NI VeriStand - System overview

Both alternatives have in common that they make use of the National Instruments toolchain to install, initialize and control NI hardware. Data sent or obtained via NI hardware is managed by the dedicated National Instruments drivers and is then handed over to the CarMaker simulation environment.

Installation on the host pc

For the installation and licensing of CarMaker/HIL for NI on the host pc please follow the general instructions given in the CarMaker Installation Guide. Additionally the following National Instruments software is required for full functionality:

- NI MAX (Measurement & Automation Explorer)
- NI LabWindows/CVI 2013
- NI LabVIEW 2014 + NI LabVIEW RT 2014 and/or
- NI VeriStand 2014

Configuring the Real-Time System

Before running CarMaker on a PXI(e) system from National Instruments it is essential to check if all necessary drivers and services are installed on the respective realtime system.

To make use of CarMaker the installation of *NI LabVIEW Real-Time* and *LabWindows/CVI Run-Time Engine* is mandatory. If you want to use CarMaker with NI VeriStand the *NI VeriStand Engine* must also be installed on your system. Keep in mind that the NI VeriStand Engine must be re-installed each time when you switch from VeriStand to LabVIEW and back again. Besides that all necessary drivers for connected hardware have to be installed. Additionally *NI-XNET* is needed for CAN integration on the CarMaker side and if you intend to use IPG hardware in conjunction with your NI realtime system *NI-VISA* drivers are needed as well.

1.9.2 First steps with CarMaker

Create a new project directory by starting the CarMaker GUI and follow the instructions given in chapter 3 of the CarMaker User's Guide. The resulting project directory contains all data and parameter files which are necessary for a later simulation on the real-time system.

It is worth mentioning that the project directory only resides on the host pc and not on the realtime target. In contrary to the general National Instruments approach CarMaker does not copy any files to the hard drive of the realtime system. Instead of that the realtime system directly accesses the files on the host pc by making use of a dedicated internal service provided with the CarMaker toolchain.

Building a new CarMaker executable

If you want to extend the CarMaker simulation program with your own C-code models the CarMaker executable has to be rebuilt before making use of your extensions. On NI systems the CarMaker simulation program is available as Dynamic Link Library (DLL) and can be compiled by using GNU make which is part of the MSYS development environment.

Before starting a build it may be necessary to adapt some settings in the central CarMaker Makefile which is located in the src subfolder of your project directory. If the installation root path of your National Instrument tools differs from "C:/Program Files (x86)/National Instruments" you have to remove the comment before the entry *NI_ROOT* and change the following path as needed. Finally open the MSYS command window, change to the src subfolder of your project directory and type make for starting the build.

1.9.3 Using CarMaker with NI LabVIEW

A LabVIEW project template can be found in the *LabVIEW* subfolder of the CarMaker project directory (if selected during project creation). Change to this folder and open the LabVIEW project by clicking the LabVIEW project file *CarMaker.lvproj*.

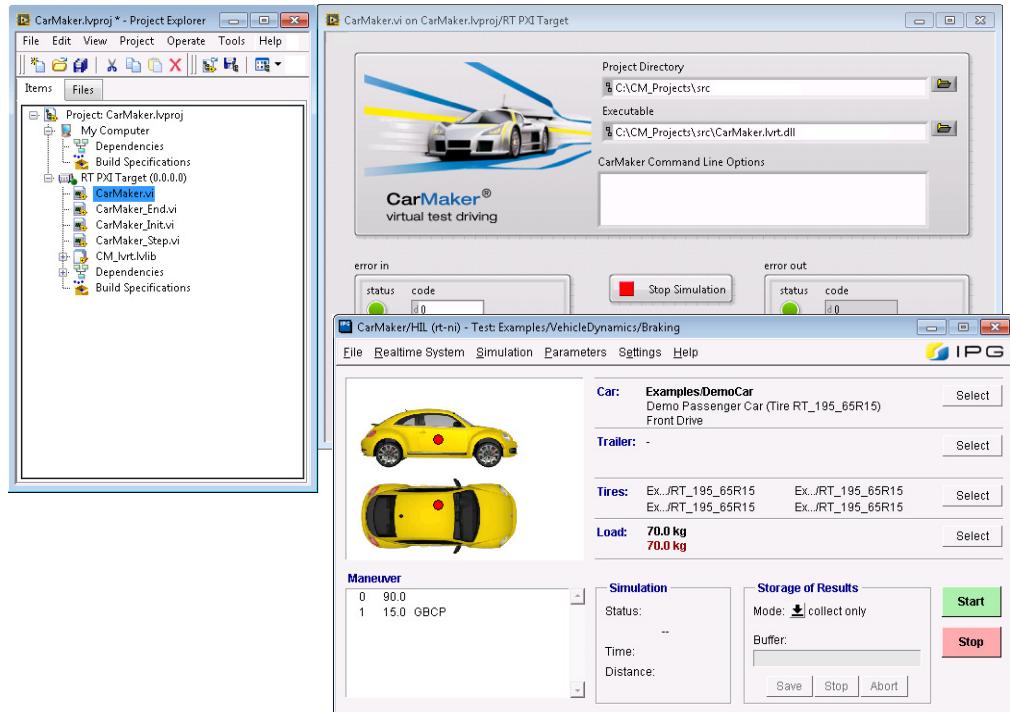


Figure 1.31: CarMaker with NI LabVIEW

Configuring the LabVIEW project

Since the project template is configured only for a generic Real-Time PXI target, you have to modify the IP address in the properties of the target to your needs before start.

The central part of the project is the CarMaker VI named *CarMaker.vi*. All other VIs of the project are only helper VIs which get called by this central VI. Open the CarMaker VI and have a look at the front panel: The entry fields for the path to the CarMaker project directory and to the desired CarMaker executable are the input controls of this VI. Additionally it is possible to define optional arguments which are handed over to the CarMaker simulation program at simulation start.

Extending the LabVIEW project

Change from the Front Panel of the CarMaker VI to the Block Diagram view. The block diagram is separated into three main parts:

The left sequence structure is dedicated to the CarMaker initialization. Besides the CarMaker Init VI where the CarMaker application is started there are two arrays which define the input and output quantities of the CarMaker simulation model. Only those quantities listed in these two arrays can be exchanged between the CarMaker simulation model and the remaining LabVIEW code. Writing to the value field of one of those input controls in Lab-

VIEW leads directly to the modification of the respective CarMaker quantity. Accordingly an internal change of a CarMaker quantity changes the value field of the respective LabVIEW output control.

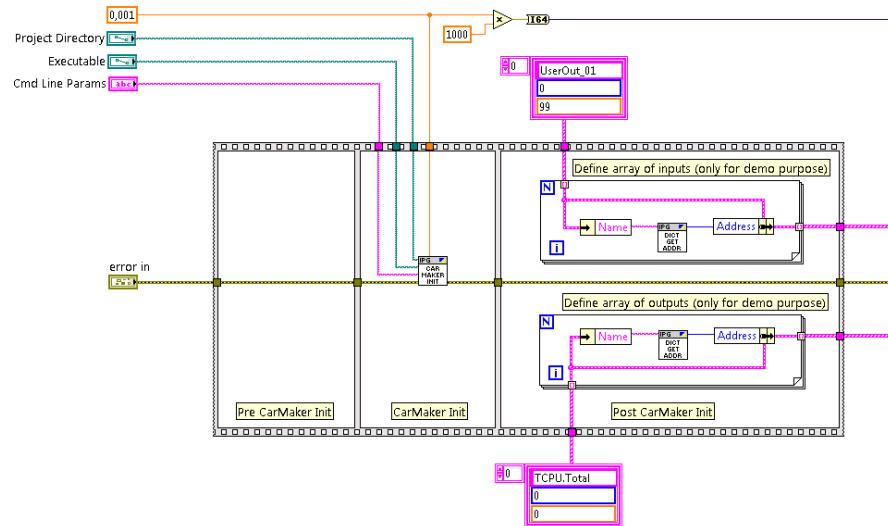


Figure 1.32: LabVIEW project template - Initialization phase

The TimedLoop following the initialization sequence resembles the CarMaker simulation cycle which is timed at 1000 Hz. It is divided into three sub-sequences: The first one acts as a *CarMaker In* block where all internal CarMaker quantities are updated with the current LabView control values. It is followed by the actual CarMaker simulation cycle where the CarMaker simulation model is called for one time step. After that the LabVIEW controls get updated in the *CarMaker Out* block with the calculated CarMaker quantities.

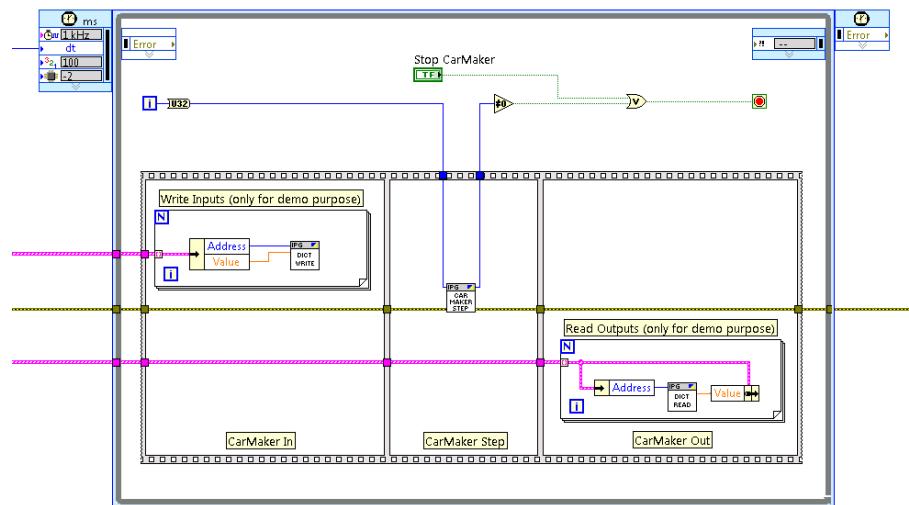


Figure 1.33: LabVIEW project template - Simulation phase

The right sequence structure handles the CarMaker deinitialization and is called when the simulation program has been requested to stop and exit.

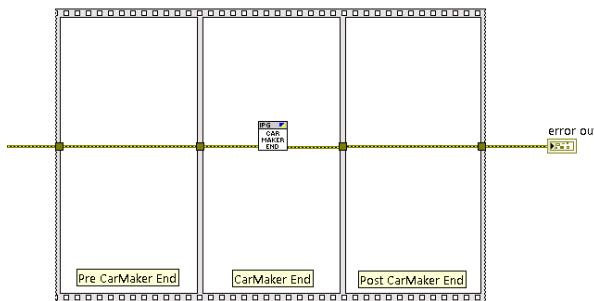


Figure 1.34: LabVIEW project template - Deinitialization phase

When adding your own code to the project template changes to the CarMaker VI should be kept to a necessary minimum. It is best practice to put all your code into a separate VI which provides the necessary input and output controls. Place this VI at the proper location within the CarMaker VI and wire the controls accordingly.

Simulating with CarMaker and LabVIEW

Open the CarMaker GUI, change to the correct project directory and run the central CarMaker VI in LabVIEW. After the download has been completed you can connect the CarMaker GUI to the running simulation program and start simulation. Depending on your project settings and the connected hardware there may be a timeout reported during the download process. In this case do not abort the process but just wait a little bit longer until the download finishes successfully.

1.9.4 Using CarMaker with NI VeriStand

CarMaker can be integrated in NI VeriStand as a so-called VeriStand Custom Device. This provides a high level of flexibility when specifying the interface to the VeriStand Engine compared to simple VeriStand simulation models whose interface is more or less static and pre-defined.

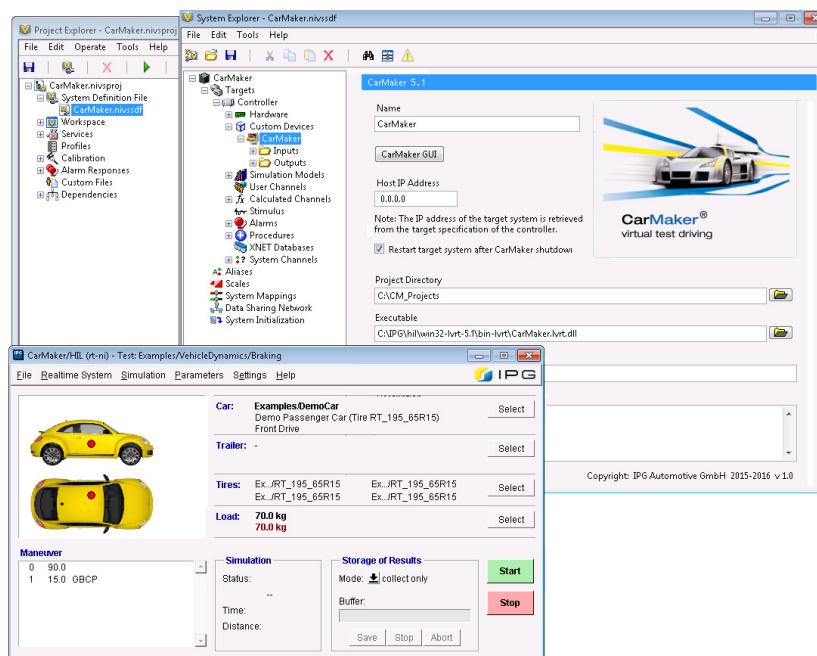


Figure 1.35: CarMaker with NI VeriStand

Adding CarMaker to a VeriStand project

In principle CarMaker can be added either to a new or an already existing NI VeriStand project. Before starting please check if the latest CarMaker Custom Device is installed in the correct place: Change to the system folder "*Users/Public/Public Documents/National Instruments/NI VeriStand 2014/Custom Devices*" and see if a folder named CarMaker and an accompanying XML file exists. If not copy the content of the *misc-lvt* subfolder of your CarMaker installation directory to the specified location and proceed afterwards.

Start NI VeriStand and open the System Definition File of the desired VeriStand project. In the System Explorer select the target controller on which CarMaker should run. Check if "Pharlap" is chosen as target operating system, the IP address of your realtime system is correct and the target rate equals 1000 Hz.

On the left side of the System Explorer window right-click on the Custom Devices item and add CarMaker to the current target. Note that CarMaker can be added only once to each target. As a first step some general settings should be checked: Change to the CarMaker main tab and enter the path to your CarMaker project directory and the desired CarMaker simulation program which should be used for simulation. Additional options which will be handed over to the CarMaker simulation program at start can be specified in the designated text field below.

After having specified the overall settings one can add additional in- and outputs to the CarMaker Custom Device. These channels will be used to transfer data from the CarMaker simulation model to external models, system channels or I/O hardware and back. All CarMaker user accessible quantities can be selected here - either with a double-click from the list of default quantities or by specifying the name of the quantity manually. After adding all necessary in- and outputs one can configure mappings between the selected CarMaker quantities and channels from other model components by making use of the established VeriStand mechanisms.

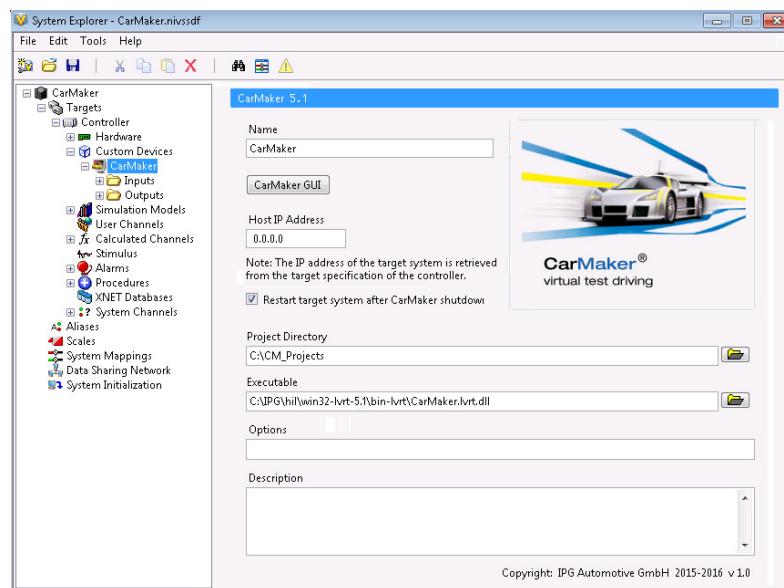


Figure 1.36: The VeriStand System Explorer with CarMaker as Custom Device

Finally if you have finished the configuration, save the System Definition File and close the System Explorer.

Simulating with CarMaker and VeriStand

When deploying the VeriStand project to the realtime system via the NI VeriStand Project Explorer the CarMaker main GUI gets started automatically and changes to the current project directory. If a timeout occurs during the deployment process the timeout value of the VeriStand Gateway can be increased by right-clicking onto the Project item and selecting Properties in the upcoming context menu. After the deployment process has finished one can connect the CarMaker main GUI with the target system and start simulation.

1.9.5 Troubleshooting

Checking DLL compatibility

When extending the CarMaker simulation model with own code it is essential to test if the resulting CarMaker DLL is executable on the realtime system. For this purpose National Instruments offers a small tool named *DLL Checker* on their website which checks if the included code makes use of any Windows API functions which are not supported on the NI realtime platform. If your code extension already comes along as a DLL it is advisable to check the DLL in advance before integrating it into CarMaker. The same applies to FMUs containing code for the win32 target: Those FMUs may or may not be executable on the realtime system depending on their specific content.

Examining download problems

If you encounter any problems during download or simulation it is often helpful to have a look at the console output of the realtime system. If you do not have a monitor connected to your realtime system you can view the console output also by opening a web browser and addressing the web server of the target system by its IP address. When using NI VeriStand the console output can also be viewed in the *Console Viewer* application which is located in the Tools menu of the System Explorer under Workspace Tools. On CarMaker side you can check the IPGRT debug console which is accessible within the CarMaker Application dialog under the HIL System button or alternatively open the CarMaker Session Log, right-click on the log output and select IPGRT Log.

Chapter 2

Logging Module

Inside your CarMaker code, e.g. during a simulation, there may be situations where it would seem useful to leave a short informational note about the current situation or to inform the user about the circumstances of an error, that just occurred. You may consider using `printf()` for this purpose, but this is not recommended under real-time conditions. Also, when the CarMaker simulation program is running on the real-time CPU, there is not necessarily a terminal available where the output of `printf()` could go.

In general it seems useful to keep a history of important or unusual situations and events during the simulation of a TestRun, that does not disappear when the simulation is finished or the user turns off his computer. This is why CarMaker offers a logging facility that keeps a record of events for each simulation in a log file. The log file can then be inspected at a later time or kept as a protocol e.g. of a driving maneuver that the connected hardware controller unit is still unable to handle.

In this section we would like to show you how to use basic functionality of the Log module in your own code. Advanced features exists but are out of scope of this text.

2.1 General information

Each time a CarMaker simulation program is started, it creates a new log file in the `SimOutput` subdirectory of the CarMaker project directory. Log messages are recorded in the log file with a time stamp relative to the start of the current simulation.

Log messages fall into one of the following three categories, that determine the importance of a message:

- Errors

Issuing an error log message causes the current simulation to be aborted, so messages of this category should be reserved for cases where the program code is unable to cope with the current situation.

- Warnings

This category should be used for situations that are unnormal, but not critical. It means that the program code is able to handle the situation. A typical example would be a parameter read during initialization that is out of range and has to be overridden with a more meaningful value.

- Purely informational messages

Use this category to inform about special events or conditions, like e.g. an ABS controller that is deactivated. It is also intended for debugging purposes. E.g. while customizing CarMaker for your real-time application, you may find it useful to tell that a certain point in initialization has been reached, or log the results of some critical computation.

Furthermore, warnings and errors are categorized by the time they are issued:

- The fault happened during initialization
- The fault happened during simulation
- The fault happened neither during initialization nor during simulation, it is considered to be a more general fault

2.2 Recommended use

As a rule of thumb, the number of log file entries should be kept to a minimum so as to increase readability. Entries should be kept short and informational. It is not useful to clutter the log file with lots of messages, especially under real-time conditions. It does not make sense to issue one or more log message in each simulation cycle, i.e. every millisecond; this unnecessarily wastes CPU power and network bandwidth.

2.2.1 List of functions

A look at the available functions should make clear, that they are quite straightforward to use. Use them as you use any function of the printf() family.

Please note the following peculiarity, though: Except for Log(), all functions automatically add a line feed character (\n) to the string to be added to the log file.

```
#include "Log.h"

void LogErrStr    (unsigned ECId, const char *msg);
void LogErrF     (unsigned ECId, const char *format, ...);

void LogWarnStr   (unsigned ECId, const char *msg);
void LogWarnF    (unsigned ECId, const char *format, ...);

void Log (const char *format, ...);    /* no line feed added automatically */

/* Constants to be used for the ECId parameter */
enum {
    EC_Init          /* Initialization fault */
    EC_Sim           /* Simulation fault */
    EC_General       /* General fault */
};
```

Example

```
#include "Log.h"

LogErrF(EC_Init, "Tyre parameter file '%s' not found.", fname);
LogWarnStr(EC_Sim, "Controller unit not responding.");
Log("Braking distance >= 50m\n");
```

Chapter 3

Infofile Module

In the CarMaker environment, configuration and parameter information are stored in ASCII files that use a special format for easy access by program code. Such files are called infofiles and the CarMaker library libcarmaker.a contains the Infofile module functionality used for opening, reading and writing to infofiles and accessing infofile data, all in a keyword oriented manner. For user code this offers a standardized and easy to use method to organize and access data in a file. It also avoids the difficulties of defining appropriate data structures, learning the syntax and checking for errors.

When the simulation of a TestRun starts, almost all TestRun configuration data is contained in infofiles in the Data subdirectory of the current CarMaker project directory. This includes information about the configuration of the virtual driver, the vehicle, tires etc. At the beginning of a simulation the modules that need specific configuration and initialization data must retrieve this data from the appropriate infofile.

This section gives a basic overview and also gives a concise list of the infofile functions in both the C and Tcl/Tk programming environments.

3.1 Infofile format

Normally, in the CarMaker environment the user can edit most data stored in infofiles comfortably using specifically tailored programs like the CarMaker GUI. This does not mean, however, that the data stored is totally inaccessible outside the CarMaker environment. Infofiles are human-readable ASCII files, i.e. they can be viewed and edited with an ordinary ASCII text editor, making them much easier to handle than the binary data files often used by other Software.

Data is stored as key-value pairs. The order in which key-value pairs appear in an infofile is not important. Keys names can (and should) be hierarchically organized using the dot character as a separator. A value is basically a number, a string or a text (i.e. a sequence of strings). This is best illustrated by a short example:

```
Composer = Sergej Prokofjef
Title = Peter and the Wolf, Op. 67
Premiere.Year = 1936
Premiere.Month = May
Movements:
    Introduction
    The Story Begins
    The Bird
```

As you surely have noticed, there is a difference between keys with a numeric or string value and keys with a text value. A text key is always delimited by a colon (':') character and the following lines containing its value (the text lines) must be indented with a single tab character.

Numerous example infofiles can be found in the *Data* subdirectory of your CarMaker project directory. As you will quickly see, infofile syntax is not at all a terribly complicated matter.

3.2 Access functions

Functions for opening and reading an infofile exist. They return a so called infofile handle of type “`struct tlnfos *`”, that is a pointer to the data read from the infofile. This handle can be used to access individual keys and retrieve their value. Normally the CarMaker library automatically opens and reads all important infofiles of the current TestRun and stores the infofile handles in the global variable `SimEnv`. Also, when the CarMaker environment invokes an initialization function like e.g. `Vhcl_NewInit()`, an infofile handle is automatically passed as a parameter. This means that in most cases you should not have to worry about how to read an infofile into memory. For more details see the example code below.

Access functions named `iGetXXX()` retrieve different types of values from keys: Strings, texts, double precision floating point values and signed or unsigned long integer values. Parameters to these functions are an infofile handle and a key name. Non-existence of a key accessed using these functions is considered to be an error.

For the purpose of retrieving values from keys that are only optionally contained in an infofile, a second set of functions named `iGetXXXOpt()` is available. They take an additional parameter to be used as the key’s default value in case the key does not exist. (The `iGetTxtOpt()` function is an exception to this rule; check for a NULL pointer returned by this function.) Non-existence of a key accessed with these functions is not considered to be an error.

Functions for the retrieval of other, higher level data types like tables and vectors also exist. Please refer to the functions description in this section for more information.

3.3 Error handling

In case of an error an appropriate log message is automatically issued by the functions described above. In addition, an internal infofile error counter is maintained by the Infofile module. In case an error occurs (i.e. a non-optional key does not exist or a key’s value could

not be retrieved), the error counter is automatically incremented. This counter provides an efficient way of error checking for your code (see next paragraph for an outline of how to use this feature). The current value of the infofile error counter can be read using the GetInfoErrorCount() function.

Error handling is best implemented as follows: before reading any infofile keys, store the current infofile error counter's value in a variable. Then access all necessary infofile keys (optional or not) in a row. After that, compare the current error counter with the value stored at the beginning. If the two values differ, at least one key's value could not be retrieved and your code should report an error. The example below illustrates this technique.

Example

```
# Infofile read by the example code (taken from Data/Vehicle/DemoCar)

Body.Mass = 1301

PowerTrain.Kind = Front
```

```
#include "infoc.h"
#include "InfoUtils.h"
#include "SimCore.h"      /* for SimCore global variable */

int
Vhcl_NewInit (const struct tInfos *Inf /*vehicle parameters*/)
{
    int count = GetInfoErrorCount();
    double vel, pos, mass;
    char *ptkind;

    mass = iGetDbl(Inf, "Body.mass");
    ptkind = iGetStr(Inf, "PowerTrain.Kind");

    /* TestRun parameters */
    vel = iGetDblOpt(SimCore.TestRun.Inf, "DrivMan.Velocityt0", 0.0);
    pos = iGetDblOpt(SimCore.TestRun.Inf, "Road.CarStartPos", 1.0);

    if (GetInfoErrorCount() != count)
        return -1; /* Error */

    return 0;
}
```

3.4 C Function List

3.4.1 General Functions

InfoNew ()

Prototype

```
tInfos *InfoNew (void);
```

Arguments

none

Description

Creates a new instance of type tInfos, usually called an Infofile handle.

Return Value

The newly created Infofile handle, or a NULL pointer on failure.

InfoDelete ()**Prototype**

```
int InfoDelete (tInfos *inf);
```

Arguments

- inf – the Infofile handle to be deleted

Description

Deletes the specified Infofile handle, i.e. deallocates all data associated with it.

Return Value

0 in case of success, -1 otherwise.

InfoRead ()**Prototype**

```
int InfoRead (tErrorMsg **perrors, tInfos *inf, const char *filename);
```

Arguments

- perrors – pointer to an array that will contain any errors generated during the reading of the infofile data
- inf – Infofile handle that will be used to store the information read
- filename – name of the infofile to be read

Description

Reads the file `filename` and stores the Infofile data read in Infofile handle `inf`.

Return Value

Returns the number of errors that were generated during the read and that are contained in the perrors list. If the value is 0 then no errors were generated. If the value is greater than 0, then there were syntax errors during the read and detailed information can be found in perrors. If the value is less than 0 then there were hard errors (e.g. file not found, permission denied, etc.) and the file could not be read at all.

InfReadMem ()**Prototype**

```
int InfoReadMem (tErrorMsg **perrors, tInfos *inf, char *bufStart, int bufSize);
```

Arguments

- `perrors` – pointer to an array that will contain any errors generated during the reading of the infofile data
- `inf` – Infofile handle that will be used to store the information read
- `bufStart` – buffer with Infofile data to be read (will be modified during the read)
- `bufSize` – number of bytes of Infofile data

Description

Reads from the specified buffer and stores the Infofile data read in Infofile handle `inf`.

Return Value

See `InfoRead()`.

InfoWrite ()

Prototype

```
int InfoWrite (tInfos *inf, const char *filename);
```

Arguments

- `inf` – Infofile handle with the data to be written
- `filename` - name of the file to be written to

Description

Writes the information contained in Infofile handle `inf` to the file specified by `filename`.

Return Value

Returns 0 on success, any other value indicates a failure (see also the description of `InfoErrno` / `InfoSysErrno` / `InfoStrError()`)

InfoListKeys ()

Prototype

```
char **InfoListKeys (tInfos *inf, const char *prefix, tIterKind kind);
typedef enum {
    Keys,
    Subkeys,
    Unread_Keys,
    All_Keys,
} tIterKind;
```

Arguments

- `inf` - `tInfos` handle to the infofile buffer
- `prefix` - specified the key prefix to be matched when searching for keys
- `kind` - the kind of key that will be included in the list. Must be one of the following enum types: `Keys`, `Subkeys`, `Unread_Keys`, or `All_Keys`.

Description

Used to get a list of the keys that match the specified search criteria. Searching is done using the two arguments `prefix` and `kind`. `Prefix` specifies a character string to match. `Kind` specifies the key search type to be used. The following kinds are possible:

- Keys - Once the all keys are identified with the specified prefix, will list the key segments that have a unique top level key segment. For example, if the prefix = AA, kind = Keys and the following list of key is used in a search:

```
AA.XX.11
_____
AA.XX.22
_____
AA.YY.11
_____
AA.YY.22
_____
AA.ZZ.11
_____
AA.ZZ.22
```

... then, the keys that would match the search criteria would be AA.XX, AA.YY, AA.ZZ. The third part of the key will play no role in this case and would be truncated from the key name when shown in the list. The list returned would be:

```
AA.XX, AA.YY, AA.ZZ
```

- Subkeys - The same as Keys except the list will be generated without the prefix part listed and only the matching subkey portion. The same search as described before would then return:
XX, YY, ZZ
- Unread_Keys - Only those keys with the matching prefix that have not been accessed (read or write) will be included in the list.
- All_Keys - All keys with the matching prefix will be returned. The whole key name will be listed. for example, using the same table and prefix as was used before, the returned list would be:

```
AA.XX.11, AA.XX.22, AA.YY.11, AA.YY.22, AA.ZZ.11, AA.ZZ.22
```

Return Value

Returns a NULL terminated list of keys, i.e. an array of char arrays.

InfoKeyKind ()

Prototype

```
tKeyKind InfoKeyKind (tInfos *inf, const char *key);
typedef enum {
    No_Key,
    Empty_Key,
    String_Key,
    Text_Key,
    Prefix_Key
} tKeyKind;
```

Arguments

- inf - tInfos handle to the infofile buffer
- key - name of the key

Description

Returns the type of value assigned to the specified key.

Return Value

The return value is an enum type that can be one of the following:

- No_Key - There is no key with the name specified
- Empty_Key - The key is not assigned a value
- String_Key - The key contains a string of data (i.e. the value is on a single line)
- Text_Key - The key contains text or a table of data (i.e. the value spans more than one line)
- Prefix_Key - The specified key is a prefix and not a complete key.

InfoDelKey ()

Prototype

```
int InfoDelKey (tInfos *inf, const char *key);
```

Arguments

- inf - tInfos handle to the infofile buffer
- key - name of the key

Description

Deletes the keys from the infofile buffer specified with `inf` that have the exact name as `key`.

Return Value

Returns 0 on success, -1 on failure.

3.4.2 Read Functions

InfoGetStr ()

Prototype

```
int InfoGetStr (char **pval ,tInfos *inf, const char *key);
```

Arguments

- pval - pointer to the string that is read from the infofile buffer. The memory associated with this variable should NOT be freed by the user.
- inf - tInfos handle to the infofile buffer
- key - name of the key to read

Description

Gets the string value of `key` located in the infofile buffer that is specified with the handle `inf`.

Return Value

Returns 0 on success, -1 on failure.

InfoGetLong ()

Prototype

```
int InfoGetLong (long *pval, tInfos *inf, const char *key);
```

Arguments

- `pval` - pointer to the value of type long int that is read from the infofile buffer. The memory associated with this variable should NOT be freed by the user.
- `inf` - tInfos handle to the infofile buffer
- `key` - name of the key to read

Description

Gets the long int value of `key` located in the infofile buffer that is specified with the handle `inf`.

Return Value

Returns 0 on success, -1 on failure.

InfoGetDbl ()

Prototype

```
int InfoGetDbl (double *pval, tInfos *inf, const char *key);
```

Arguments

- `pval` - pointer to the value of type double that is read from the infofile buffer. The memory associated with this variable should NOT be freed by the user.
- `inf` - tInfos handle to the infofile buffer
- `key` - name of the key to read

Description

Gets the double value of `key` located in the infofile buffer that is specified with the handle `inf`.

Return Value

Returns 0 on success, -1 on failure.

InfoGetTxt ()

Prototype

```
int InfoGetTxt (char ***pval, tInfos *inf, const char *key);
```

Arguments

- `pval` - pointer to the text string value that is read from the infofile buffer. The memory associated with this variable should NOT be freed by the user.
- `inf` - tInfos handle to the infofile buffer
- `key` - name of the key to read

Description

Gets the text string value of `key` located in the infofile buffer that is specified with the handle `inf`. Used to read key values that span multiple lines.

Return Value

Returns 0 on success, -1 on failure.

InfoGetStrDef ()

Prototype

```
int InfoGetStrDef (char **pval , tInfos *inf, const char *key, const char *def);
```

Arguments

- `pval` - pointer to the string value that is read from the infofile buffer. The memory associated with this variable should NOT be freed by the user.
- `inf` - `tInfos` handle to the infofile buffer
- `key` - name of the key to read
- `def` - default key value

Description

Gets the string value of `key` located in the infofile buffer that is specified with the handle `inf`. If there is no value assigned to it then the default value will be used.

Return Value

Returns 0 on success, 1 if the key does not exist and -1 on failure.

InfoGetLongDef ()

Prototype

```
int InfoGetLongDef(long *pval, tInfos *inf, const char *key, long def);
```

Arguments

- `pval` - pointer to the long int value that is read from the infofile buffer. The memory associated with this variable should NOT be freed by the user.
- `inf` - `tInfos` handle to the infofile buffer
- `key` - name of the key to read
- `def` - default key value

Description

Gets the long int value of `key` located in the infofile buffer that is specified with the handle `inf`. If there is no value assigned to it then the default value will be used.

Return Value

Returns 0 on success, 1 if the key does not exist and -1 on failure.

InfoGetDblDef ()

Prototype

```
int InfoGetDblDef (double *pval ,tInfos *inf, const char *key, double def);
```

Arguments

- `pval` - pointer to the double value that is read from the infofile buffer. The memory associated with this variable should NOT be freed by the user.
- `inf` - `tInfos` handle to the infofile buffer
- `key` - name of the key to read
- `def` - default key value

Description

Gets the double value of `key` located in the infofile buffer that is specified with the handle `inf`. If there is no value assigned to it then the default value will be used.

Return Value

Returns 0 on success, 1 if the key does not exist and -1 on failure.

InfoGetTxtDef ()

Prototype

```
int InfoGetTxtDef (char ***pval, tInfos *inf, const char *key, char **def);
```

Arguments

- `pval` - pointer to the text string value that is read from the infofile buffer. The memory associated with this variable should NOT be freed by the user.
- `inf` - `tInfos` handle to the infofile buffer
- `key` - name of the key to read
- `def` - default key value

Description

Gets the text string value of `key` located in the infofile buffer that is specified with the handle `inf`. Used to read key values that span multiple lines. If there is no value assigned to it then the default value will be used.

Return Value

Returns 0 on success, 1 if the key does not exist and -1 on failure.

3.4.3 Write Functions

InfoSetStr ()

Prototype

```
int InfoSetStr(tInfos *inf, const char *key, const char *val);
```

Arguments

- `inf` - tInfos handle to the infofile buffer
- `key` - name of the key to be set
- `val` - new string value to be written to the key

Description

Sets the value of the key. If the specified key does not exist then one will be created.

Return Value

Returns 0 on success, and -1 on failure.

InfoSetLong ()

Prototype

```
int InfoSetLong(tInfos *inf, const char *key, long val);
```

Arguments

- `inf` - tInfos handle to the infofile buffer
- `key` - name of the key to be set
- `val` - new long int value to be written to the key

Description

Sets the value of the key. If the specified key does not exist then one will be created.

Return Value

Returns 0 on success, and -1 on failure.

InfoSetDbl ()

Prototype

```
int InfoSetDbl(tInfos *inf, const char *key, double val);
```

Arguments

- `inf` - tInfos handle to the infofile buffer
- `key` - name of the key to be set

- `val` - new double value to be written to the key

Description

Sets the value of the key. If the specified key does not exist then one will be created.

Return Value

Returns 0 on success, and -1 on failure.

InfoSetTxt ()

Prototype

```
int InfoSetTxt(tInfos *inf, const char *key, char **val);
```

Arguments

- `inf` - tInfos handle to the infofile buffer
- `key` - name of the key to be set
- `val` - new text string value to be written to the key

Description

Sets the value of the key. If the specified key does not exist then one will be created.

Return Value

Returns 0 on success, and -1 on failure.

3.4.4 Add/Move Functions

InfoMoveKeyBefore ()

Prototype

```
int InfoMoveKeyBefore (tInfos *inf, const char *destkey, const char *key);
```

Arguments

- `inf` - tInfos handle to the infofile buffer
- `destkey` - name of the destination key used as a reference for the move
- `key` - name of the key to be moved

Description

Moves the key specified by `key` directly before the key specified by `destkey`.

Return Value

Returns 0 on success, and -1 on failure.

InfoMoveKeyBehind ()

Prototype

```
int InfoMoveKeyBehind (tInfos *inf, const char *destkey, const char *key);
```

Arguments

- inf - tInfos handle to the infofile buffer
- destkey - name of the destination key used as a reference for the move
- key - name of the key to be moved

Description

Moves the key specified by `key` behind the key specified by `destkey`.

Return Value

Returns 0 on success, and -1 on failure.

InfoAddLineBefore ()

Prototype

```
int InfoAddLineBefore (tInfos *inf, const char *destkey, const char *val);
```

Arguments

- inf - tInfos handle to the infofile buffer
- destkey - name of the destination key used as a reference for the line insertion
- val - character string to be inserted

Description

Used to insert a comment or a blank line before the key specified by `destkey`.

Return Value

Returns 0 on success, and -1 on failure.

InfoAddLineBehind ()

Prototype

```
int InfoAddLineBehind (tInfos *inf, const char *destkey, const char *val);
```

Arguments

- inf - tInfos handle to the infofile buffer
- destkey - name of the destination key used as a reference for the line insertion
- val - character string to be inserted

Description

Used to insert a comment or a blank line behind the key specified by `destkey`.

Return Value

Returns 0 on success, and -1 on failure.

3.4.5 Helper Functions

InfoKeyJoin ()

Prototype

```
char *InfoKeyJoin (char *subkeys[ ]);
```

Arguments

- `subkeys` – NULL-terminated list of subkeys to be concatenated

Description

Concatenates a list of subkeys, delimited by a NULL pointer, into a single key. The subkeys will be separated with a period.

Return Value

Returns the newly formed key. The value is statically allocated and therefore only valid until the next call to this function.

InfoKeySplit ()

Prototype

```
char **InfoKeySplit (const char *key);
```

Arguments

- `key` – key to be split

Description

Splits a key into its subkeys. The subkeys must be separated by a period.

Return Value

Returns the list of subkeys. The value is statically allocated and therefore only valid until the next call to this function.

3.4.6 Error handling

InfoStrError ()

Prototype

```
const char *InfoStrError (void);
```

Arguments

None

Description

Used to get a descriptive string for the last error stored in global variable InfoErrno.

Return Value

The error message string.

3.5 Tcl/Tk Procedure List

3.5.1 Library

Command

```
load <path_to_ifile.so>
```

This is a standard Tcl/Tk command. It should be called first to load the infofile library. Refer to the Tcl/Tk documentation for a detailed description of the `load` command.

3.5.2 ifile

Procedure

```
ifile option <?arg1> <?arg2> ... <?argn>
```

Description

`ifile` performs one of many operations depending on `option`. Also, the arguments that are needed depend on `option`.

The following description will show the functionality of `ifile` for all possible options.

ifile new

```
ifile new <handle>
```

With the option `new` the `ifile` procedure is used to create a new infofile instance. It takes a single argument, called `handle`. `handle` is the name of a procedure that will be created when `ifile new` is invoked, and will be the procedure used to perform most of the operations on the infofile instance (as described in the next section). It is called a `handle` because it is synonymous with a unique identifier, and used to specify the infofile instance that will be created here and managed in subsequent calls (e.g. when reading and writing).

This is synonymous with the C-function call to *InfoNew()*.

ifile delete

ifile delete <handle>

With the option *delete* the infofile instance referred to by *handle* and created with the call to **ifile new** is deleted. All data associated with the specified handle will be deleted.

This is synonymous with the C-function call to *InfoDelete()*.

ifile makekey

ifile makekey <subkey list> <var>

With the option *makekey* the **ifile** procedure is used to concatatanate a list of subkeys into a single key. The subkeys will be delimited with the dot operator (i.e a period). The first argument *subkey list* is the list of subkeys to be put together. The second argument *var* is the variable that will contain the newly formed key.

This is synonymous with the C-function call to *InfoMakeKeys()*.

ifile splitkey

ifile splitkey <key> <var>

With the option *splitkey* the **ifile** procedure is used to split the given key into the subkeys, delimited with the dot operator. The first argument *key* is the key to be split, and the second argument, *var*, will contain the list of subkey that is created upon return.

This is synonymous with the C-function call to *InfoSplitKeys()*.

ifile strerror

ifile strerror

With the option *strerror* the procedure **ifile** returns a text message that contains the last error that occured during an invocation to one of the infofile procedures.

ifile version

ifile version

With the option *version* the procedure **ifile** returns a string containg the version number of the infofile library that is being used.

3.5.3 <handle>

The procedure <handle> is the one created with the call to **ifile new** <handle> as previously described (see [section on page 74](#)). If, for example, the call was as follows:

```
ifile new myhandle
```

then <handle> would be **myhandle** (i.e. whatever name was given for the argument). The myhandle function will then be use for a number of purposes based on the supplied option and arguments. The syntax is:

```
<handle> option <?arg1> <?arg2> ... <?argn>
```

The following description will show the functionality of <handle> for all possible options.

<handle> read

<handle> read <filename> <errorvar>

With the option *read* the procedure **<handle>** adds the contents of a file into the data location that is associated with the given handle. The argument *filename* is the name of the file to be read. The argument *errorvar* is a variable that will be used to store the list of errors, or an empty string if there are no errors.

The procedure returns the number of errors that were generated during the read and that are contained in the error list. If the value is (0) then no errors were generated, if the value is (greater than 0) then there were syntax errors during the read, if the value is (less than 0) then there were access, i.e. hard errors during the read which caused the read to fail.

This procedure is synonymous with the C-function InfoRead().

<handle> readmem

<handle> readmem <string> <errorvar>

With the option *readmem* the procedure **<handle>** adds the contents of a string into the data location that is associated with the given handle. The argument *string* is the string buffer to be read. The argument *errorvar* is a variable that will be used to store the list of errors, or an empty string if there are no errors.

The procedure returns the number of errors that were generated during the read and that are contained in the error list. If the value is (0) then no errors were generated, if the value is (greater than 0) then there were syntax errors during the read, if the value is (less than 0) then there were access, i.e., hard errors, during the read which caused the read to fail.

This procedure is synonymous with the C-function InfoReadMem().

<handle> write

<handle> write <filename>

With the option *write* the procedure **<handle>** writes the infofile data to a file. The argument *filename* is the name of the file to be written to.

<handle> getstr

<handle> getstr <key> <var> <?default-string>

With the option *getstr* the procedure **<handle>** reads the value of the given key and store it in the specified variable. If the key contains data that spans more than one line (i.e. it is a text key) then the string that is returned will contain the lines of data separated by a newline character (\n).

The argument *key* is the name of the key to read, the argument *var* is the variable that the data will be stored, and the argument *default-string* is an optional argument that allows a default string to be specified if the key to be read is not assigned a value.

This procedure is synonymous with InfoGetStr().

<handle> setstr

<handle> setstr <key> <string>

With the option *setstr* the procedure **<handle>** assignes a string to the specified key. This procedure is used to set a single line of data. The argument *key* is the name of the key that will be set and the argument *string* contains the string data.

This procedure is synonymous with InfoSetStr().

<handle> gettxt

<handle> gettxt <key> <var> <?default-list>

With the option `gettxt` the procedure `<handle>` reads the value of the given key and store it in the specified variable. The argument `key` is the name of the key to read, the argument `var` is the variable that the data will be stored, and the argument `default_string` is an optional argument that allows a default list to be specified if the key to be read is not assigned a value.

This procedure is synonymous with `InfoGetTxt()`.

`<handle> settxt`

`<handle> settxt <key> <list>`

With the option `settxt` the procedure `<handle>` assignes a string to the specified key. This procedure is used to set multiple lines of data. The argument `key` is the name of the key that will be set and the argument `list` contains the data list.

This procedure is synonymous with `InfoSetTxt()`.

`<handle> keykind`

`<handle> keykind <key>`

With the option `keykind` the procedure `<handle>` returns the key type/kind of the specified key. Possible key types are:

- `no_key`
- `string_key`
- `text_key`

This procedure is synonymous with `InfoKeyKind()`.

`<handle> delkey`

`<handle> delkey <key>`

With the option `delkey` the procedure `<handle>` deletes the given key and its value.

This procedure is synonymous with `InfoDelKey()`.

`<handle> listkeys`

`<handle> listkeys <prefix> <kind> <var>`

With the option `listkeys` the procedure `<handle>` returns the list of keys that start with `prefix` and are of the given kind. The argument `prefix` is the prefix subkey that is searched for, the argument `kind` is the kind of search to be done and the argument `var` is the variable that will store the results upon return.

See the description of `InfoListKeys()` for more information.

In case of the Tcl function an additional value `matching_keys` for the `kind` argument can be specified, which enables pattern matching using shell wildcard characters like `? [] *` etc.

`<handle> movekeybefore`

`<handle> movekeybefore <destkey> <key>`

With the option `movekeybefore` the procedure `<handle>` moves the specified key to the position directly before the destination key.

This procedure is synonymous with the C-function `InfoMoveKeyBefore()`.

`<handle> movekeybehind`

`<handle> movekeybehind <destkey> <key>`

With the option *movekeybehind* the procedure **<handle>** moves the specified key to the position directly behind the destination key.

This procedure is synonymous with the C-function InfoMoveKeyBehind().

<handle> addlinebefore

<handle> addlinebefore <?destkey> <string>

With the option *addlinebefore* the procedure **<handle>** adds the specified key to the position directly before the destination key. If the destination key is omitted then the line will be added before the very first line.

This procedure is synonymous with the C-function InfoAddLineBefore().

<handle> addlinebehind

<handle> addlinebehind <?destkey> <string>

With the option *addlinebehind* the procedure **<handle>** adds the specified key to the position directly behind the destination key. If the destination key is omitted then the line will be added after the very last line.

This procedure is synonymous with the C-function InfoAddLineBehind().

Chapter 4

Data Dictionary

4.1 Introduction

Inside the CarMaker simulation program the DataDict module stores important variables (quantities) of the program in a data dictionary. This dictionary is the basis for the following functionality in the CarMaker environment:

- Storage of simulation results
Only quantities registered in the data dictionary can be selected for storage in a results file.
- Availability of online data during a simulation
CarMaker user interface tools can request the simulation program to send them the values of selected quantities at regular intervals. Online 3D animation of the vehicle with IPGMovie and data visualization with IPGControl during a running simulation are just two examples of how programs make use of the simulation program's data dictionary.

4.2 Defining DataDict Variables

4.2.1 General hints

The relevant #include file is DataDict.h. It contains all necessary type definitions and function prototypes.

For each basic C type (double, float, int, ...) of a variable there is a corresponding function *DDefXXX()* to register this variable in the data dictionary. Variables whose values are monotonically increasing over time or whose range spans only a limit number of discrete states should be marked accordingly using functions like *DDefAttrib()* and *DDefStates()*. Visualization tools like IPGControl rely heavily on the correct specification of these attributes and make use this information to optimize display of the values of a quantity.

Registering a variable in the dictionary means that you have to specify a name and a unit for the variable, and its address in memory. Name and unit length should be kept to a sensible maximum of around 32 characters, otherwise readability will decrease when display-

ing such long names in graphical dialogs and menus. The name can be arbitrary, it does not have to be the C code name of the variable. The unit should be the physical unit of the quantity in question. If your variable has no correspondence with a physical unit, specify an empty string. Examples of units used in CarMaker are rpm, rad/s², m/s, m/s², rad/s, m, bar. See also *Unit List* in IPGControl's *Help* menu.

It helps, if you organize the names of your quantities in a systematic and consistent manner, group them by common prefixes and use a naming hierarchy with levels separated by a dot character (IPGControl will group if more than three quantities are in the same group).

Cluttering the dictionary with lots of variables of only minor interest is counterproductive. The number of variables in the dictionary should be kept to a meaningful minimum, as memory and processor time are limited resources, especially on real-time systems. An oversized data dictionary may have an impact on real-time performance of the application.

Dictionary variables should be registered only in functions explicitly intended for this purpose, e.g. *User_DeclQuants()*. It is crucial for proper operation of the data dictionary that dictionary variables are not registered at arbitrary times.



After registering a variable, the variable is expected to exist until the program ends. Registering local (automatic) variables of a function is an error, as the variable's memory is allocated on the stack, which will disappear once the function returns. Use global variables or variables allocated on the heap instead; otherwise memory access errors may result.

A quantity may be defined only once. Once it is defined, redefinition is allowed only with exactly the same parameters as before, otherwise an error will result. Exception to the rule: The memory address of the variable is allowed to change, but name, unit, number of states etc. must all stay the same. This is useful for variables that are dynamically (re-)allocated each time a simulation starts. Normally you do not have to worry about this.

4.2.2 Quantity Definition API

DDefDouble(), DDefFloat(), etc.

```
tDDictEntry *DDefDouble      (tDDDefault *df, const char *name, const char *unit,
                             double *var, tDVAPlace place);
tDDictEntry *DDefDouble4     (tDDDefault *df, const char *name, const char *unit,
                             double *var, tDVAPlace place);
tDDictEntry *DDefFloat       (tDDDefault *df, const char *name, const char *unit,
                             float *var, tDVAPlace place);

tDDictEntry *DDefLong        (tDDDefault *df, const char *name, const char *unit,
                             long *var, tDVAPlace place);
tDDictEntry *DDefULong       (tDDDefault *df, const char *name, const char *unit,
                             unsigned long *var, tDVAPlace place);

tDDictEntry *DDefInt         (tDDDefault *df, const char *name, const char *unit,
                             int *var, tDVAPlace place);
tDDictEntry *DDefUInt        (tDDDefault *df, const char *name, const char *unit,
                             unsigned int *var, tDVAPlace place);

tDDictEntry *DDefShort       (tDDDefault *df, const char *name, const char *unit,
                             short *var, tDVAPlace place);
tDDictEntry *DDefUShort      (tDDDefault *df, const char *name, const char *unit,
                             unsigned short *var, tDVAPlace place);

tDDictEntry *DDefChar        (tDDDefault *df, const char *name, const char *unit,
                             char *var, tDVAPlace place);
tDDictEntry *DDefUChar       (tDDDefault *df, const char *name, const char *unit,
```

```
unsigned char *var, tDVAPlace place);
```

Description

Adds a quantity of the corresponding C-type to the data dictionary, or redefines its memory address.

DDefDouble4() is special in that its C-variable is of type *double*, but the quantity's value will be a *float*. In many cases single precision is sufficient for quantities, which are mainly introduced for the purpose of visualization by APO clients like Instruments, IPGMovie or IPG-Control. Defining a quantity with *DDefDouble4()* allows for calculation of the value as double precision but helps saving network bandwidth by providing the result only as single precision.

For a description of the first parameter *df* and some example code see functions *DDefaultCreate()*, *DDefaultDelete()* and *DDefPrefix()* below. Passing *NULL* for *df* is valid and means not to use any previously defined default settings.

Direct Variable Access (DVA): If the quantity is read-only, *place* should be assigned *DVA_None*. Otherwise one of *DVA_IO_In*, *DVA_DM*, *DVA_VC* or *DV_IO_Out* should be assigned as the DVA write access place.

In case of success a handle to the resulting dictionary entry is returned, *NULL* otherwise. The handle may be used e.g. in subsequent calls to functions like *DDefStates()* or *DDefAttrib()*.

DDefStates ()

```
tDDictEntry *DDefStates (tDDictEntry *e, int nstates, int firststate);
```

Description

Defines the number of states and the first state for the integer quantity *e*.

A *NULL* pointer is a valid entry for *e*; the function will simply do nothing in this case.

In case of success *e* is returned, *NULL* otherwise.

Example

```
int GearNo;      // Gear numbers 1-5, neutral gear 0, reverse gear -1
tDDictEntry *e;

e = DDefInt(NULL, "MyModel.GearNo", "", &GearNo, DVA_None);
DDefStates(e, 6, -1);
```

DDefAttrib ()

```
tDDictEntry *DDefAttrib (tDDictEntry *e, unsigned flags);
```

Description

Defines additional attributes for quantity *e*. The *flags* parameter is a bitmask.

Currently only *DDictEntry_Mono* can be specified for *flags*, designating that the quantity is monotonically increasing.

A *NULL* pointer is a valid entry for *e*; the function will simply do nothing in this case.

In case of success *e* is returned, *NULL* otherwise.

Example

```
double Distance;
tDDictEntry *e;

e = DDefDouble(NULL, "MyModel.Dist", "m", &Distance, DVA_None);
DDefAttrib(e, DDictEntry_Mono);
```

DDefaultCreate () DDefaultDelete ()

```
tDDefault *DDefaultCreate (const char *nameprefixfmt, ...);
void DDefaultDelete (tDDefault *df);
```

Description

Creates (and destroys) a set of default settings shared by a number of subsequent *DDefXXX()* calls.

Currently the only and also most often used setting available is a common name prefix for a group of quantities. It can be (re)defined with *DDefPrefix()* (see below), but since common use is to define the prefix only once, a shortcut was introduced, so *DDefaultCreate()* accepts the same parameters as *DDefPrefix()*, saving you one function call. See the description of *DDefPrefix()* for a detailed example.

tDDefault instances are completely independent of each other and can be used in parallel.

A *tDDefault* instance must be freed with *DDefaultDelete()* after use, otherwise a memory leak will result.

DDefPrefix ()

```
void DDefPrefix (tDDefault *df, const char *nameprefixfmt, ...);
```

Description

Defines a name prefix stored in *tDDefault* instance *df*. The name prefix can be created in a *printf()*-like fashion. Passing *NULL* (or "") for *nameprefixfmt* clears the name prefix.

Invoking *DDefPrefix()* multiple times on the same *tDDefault* instance is completely valid and intended use.

Example

```
tDDefault *df = DDefaultCreate("MyModel");
int i;

DDefDict(df, ".Dist", "m", &Distance, DVA_None); // Defines "MyModel.Dist"
DDefDict(df, ".v", "m/s", &Velocity, DVA_None); // Defines "MyModel.v"

// Define "MyModel.FL.ax", "MyModel.FR.ax", etc.
for (i=0; i<4; i++) {
    static char *Tirepos[] = { "FL", "FR", "RL", "RR" };
    DDefPrefix(df, "MyModel.%s", Tirepos[i]);
    DDefDict(df, ".ax", "m/s^2", &MyTire[i].ax, DVA_None);
    DDefDict(df, ".ay", "m/s^2", &MyTire[i].ay, DVA_None);}

DDefaultDelete(df);
```

Chapter 5

Integrating C-code models

5.1 Model Manager

For every subsystem of the car – e.g. steering system, brake system, ... – more than one model can be available in the simulation program. The model to be used in a specific TestRun is selected in the corresponding tab of the vehicle data set which indirectly sets a key like *Model.Kind* in the vehicle parameter infofile, e.g. *Brake.Kind = MyModel*. This action will select the user model *MyModel* instead of one of the internal CarMaker brake models

Table 5.1: Subsystems supported by model manager

Subsystem Type	Model class	Key
Aerodynamics	Aero	Aero.Kind
Brake	Brake	Brake.Kind
Hydraulic Brake	HydBrakeSystem	Brake.System.Kind
Hydraulic Brake Control Unit	HydBrakeControl	Brake.Control.Kind
Air Brake	AirBrakeSystem	Brake.System.Kind (Truck-Maker only)
Air Brake Control Unit	AirBrakeControl	Brake.Control.Kind (Truck-Maker only)
Environment	Environment	Env.Kind
Powertrain	PowerTrain	PowerTrain.Kind
Powertain with OpenXWD	PowerTrainXWD	PowerTrain.Kind
Engine Control Unit	PTEngineCU	PowerTrain.ECU.Kind
Engine Torque	PTEngine	PowerTrain.ET.Kind
E-Motor Control Unit	PTMotorCU	PowerTrain.MCU.Kind
E-Motor	PTMotor	PowerTrain.Motor<n>.Kind
Transmission Control Unit	PTTransmCU	PowerTrain.TCU.Kind
Clutch	PTClutch	PowerTrain.Clutch.Kind

Table 5.1: Subsystems supported by model manager

Subsystem Type	Model class	Key
GearBox	PTGearBox	PowerTrain.GearBox.Kind
Driveline	PTDriveLine	PowerTrain.DriveLine.Kind
Driveline with OpenXWD	PTDriveLineXWD	PowerTrain.DL.Kind
Powertrain Control	PTControl	PowerTrain.Control.Kind
Powertrain Operation State Machine	PTControlOSM	PowerTrain.ControlOSM.Kind
Powertrain Driveline	PTGenCoupling	Power-Train.DL.<pos>.Diff.Cpl.Kind
Battery Control Unit	PTBatteryCU	PowerTrain.BCU.Kind
Battery	PTBattery	PowerTrain.PowerSupply.Batt<LV/HV>.Kind
Power Supply	PTPowerSupply	PowerTrain.PowerSupply.Kind
Steering	Steering	Steering.Kind
Kinematics and Compliance	SuspKnC	Susp<Pos>.<Kin/Com>.<No ³ >.Kind
Kinematics	SuspKnC_Kin	Susp<pos>.Kin.0.Kind
Compliance	SuspKnC_Com	Susp<pos>.Com.0.Kind
Active Suspension Systems	SuspExtFrcs	SuspExtFrcs.Kind
Active Buffer	SuspEF_Buffer	SuspExtFrcs.Buffer.Kind
Active Damper	SuspEF_Damper	SuspExtFrcs.Damper.Kind
Active Spring	SuspEF_Spring	SuspExtFrcs.Spring.Kind
Active Stabilizer	SuspEF_Stabi	SuspExtFrcs.Stabi.Kind
Suspension front	MCycleSuspFront	SuspF.SK.Kind (MotorcycleMaker only)
Suspension rear	MCycleSuspRear	SuspR.SK.Kind (MotorcycleMaker only)
Suspension force element	MCycleSuspFrcEl	SuspR.SK.FrcEl.Kind (MotorcycleMaker only)
Tire ¹	Tire	²
Tire-road contact point excitation ¹	TireCPMod	TireCPMod.Kind
User Driver Model	UserDriver	²
Vehicle ¹	Vehicle	²
Vehicle Control	VehicleControl	VehicleControl.Kind
Vehicle Operator	VhclOperator	DrivMan.VhclOperator.Kind
Free	Generic	²

¹ This model class needs a special treatment. Please refer to [section 5.1.2 'Exceptions and special cases'](#).² Can only be identified with the FName key, see [Table 5.2](#):³ Pos: F, R, M, F2, R2, depending on car, truck and/or trailer, No: number of applied model

A major advantage of this implementation is the possibility to switch the models "on-the-fly" between two runs just by changing vehicle parameters. As only one of the integrated models is calculated at the same time, the maximum simulation performance is assured. Another advantage is the modular design with the additional possibility to extend the

model functionality with user code.

The internal sequence of the model classes calculated in one simulation cycle is as follows:

- Environment module
- Traffic Lights
- DrivMan
 - IPGDriver
 - UserDriver
 - Maneuvers
 - Input From File
- User_DrivMan_Calc
- Traffic
- User_Traffic_Calc
- VehicleControl
- User_VehicleControl_Calc
- Vehicle
 - Car
 - Steering
 - Suspension Kinematics and Compliance
 - Aerodynamics
 - External Suspension Forces
 - Tire
 - Trailer
 - Brake
 - Brake Control Unit
 - Brake System
 - PowerTrain
 - (kind: Generic)
 - Engine Control Unit
 - Engine
 - Motor Control Unit
 - Motor
 - Transmission Control Unit
 - Clutch
 - GearBox
 - DriveLine
 - (kind: Gen4WD)
 - PTGenCoupling Center Differential
 - PTGenCoupling Front Differential
 - PTGenCoupling Rear Differential
 - Retarder (TruckMaker only)
 - Battery Control Unit
 - Power Supply
- Body Sensors
- Driver Assistance Sensors
- Free Space Sensors

- Road Property Sensors
- Traffic Sign Sensors
- Line Sensors
- Collision Detection module
- GNSS module (Satellite module)
- Pylon Detection
- User_Calc

5.1.1 Model Registration

A model needs to be registered in a specific class providing the identification string (*Kind-Str*) and a model specific structure (*MD*). Please see *ModelManager.h* for the current definition of the required structure of each model and its elements.

Listing 5.1: Register function

```

1: int
2: Model_Register (
3:     tModelClass      ModelClass,
4:     const char       *KindStr,
5:     tModelClassDescr *MD
6: );

```

The registration has to be done before the first simulation starts, typically the function will be called once in the *User_Register()* function (see *src/User.c*).

Listing 5.2: Example for a typical registering function in the user model c-code

```

1: int
2: Brake_Register_Simple (void)
3: {
4:     tModelClassDescr m;
5:
6:     memset (&m, 0, sizeof(m));
7:     m.Brake.VersionId = 1;
8:     m.Brake.New =      BrakeSimple_New;
9:     m.Brake.Calc =     BrakeSimple_Calc;
10:    m.Brake.Delete =   BrakeSimple_Delete;
11:    m.Brake.DeclQuants = BrakeSimple_DeclQuants;
12:
13:    return Model_Register(ModelClass_Brake, "Simple", &m);
14: }

```

Listing 5.3: Example for a function call in User.c to register the model in CarMaker

```

1: int
2: User_Register (void)
3: {
4:     Brake_Register_Simple ();
5: }

```

Mandatory Elements

For every model class 3 function pointers are mandatory:

- New - The function will be called at the beginning of every TestRun. Dynamic memory allocation and reading of parameters from file should be placed here.

- Calc - The function will be called in every simulation cycle. It calculates the model outputs.
- Delete - The function will be called before a new model will be initialized. Use it to free dynamically allocated memory.

Optional Elements

You can add additional elements to link the model with internal CarMaker functions:

- VersionId - To identify the model version.
- DeclQuants - The function will be called at the beginning of every TestRun and once when the ..._Register() function is called. Declaration of model specific User Accessible Quantities should be placed here.

Internal Steps

What CarMaker does when starting a simulation:

- Read the corresponding key *Model.Kind* from file.
- Look for a model with the corresponding kind string.
- If the model is found in the internal database, its corresponding New() function is called to initialize the model.
- The functions Calc() and Delete() are stored to be recalled later for calculation of the model and deletion of model parameters.

5.1.2 Exceptions and special cases

To integrate user models of the model manager classes *Vehicle*, *Tire* and *TireCPMod* some special care needs to be taken as described below.

Vehicle

Please refer to [section 'Replacing the Vehicle Model'](#) for a detailed description.

Tire

CarMaker supports two different interfaces for user tire models as described in Reference Manual chapter "Tire" : Standard Tire Interface (STI) and Contact Point Interface (CPI). Both are activated using the infofile approach described in [section 5.1.3 'Using parameter infofiles'](#) with *FileIdent = CarMaker-Tire-<Name>*. The information if a certain user model uses STI or CPI is not set as a parameter, but in the model registration on c-code level. In this function a parameter *m.Tire.is3DTire* is set to 1, if the model is a STI model, or 0, if the model is a CPI model. [Listing 5.4](#) and [Listing 5.5](#) show this difference for two example models.

[Listing 5.4: Example register function for a STI tire model](#)

```

1: int
2: Tire_Register_MyModelSTI (void)
3: {
4:     tModelClassDescr m;
5:
6:     memset (&m, 0, sizeof(m));
7:     m.Tire.VersionId =ThisVersionId_STI;
8:     m.Tire.is3DTire = 1;// set 1 if STI interface is used
9:     m.Tire.New =MyModel_STI_New;
10:    m.Tire.Calc =MyModel_STI_Calc;
11:    m.Tire.Delete =MyModel_Delete;
12:    m.Tire.DeclQuants =MyModel_DeclQuants;
13:
14:    return Model_Register(ModelClass_Tire, ThisModelKind_STI, &m);
15: }
```

Listing 5.5: Example register function for a CPI tire model

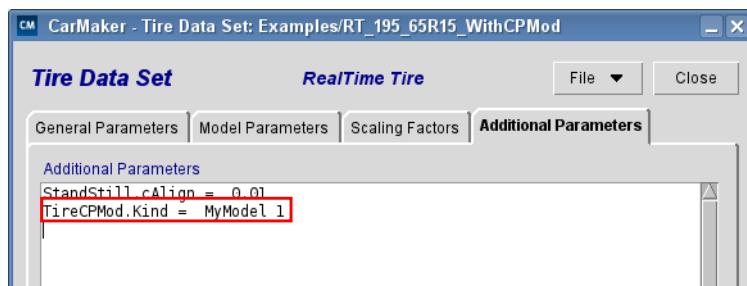
```

1: int
2: Tire_Register_MyModelCPI (void)
3: {
4:     tModelClassDescr m;
5:
6:     memset (&m, 0, sizeof(m));
7:     m.Tire.VersionId =ThisVersionId_CPI;
8:     m.Tire.is3DTire = 0;// set 0 if CPI interface is used
9:     m.Tire.New =MyModel_CPI_New;
10:    m.Tire.Calc =MyModel_CPI_Calc;
11:    m.Tire.Delete =MyModel_Delete;
12:    m.Tire.DeclQuants =MyModel_DeclQuants;
13:
14:    return Model_Register(ModelClass_Tire, ThisModelKind_CPI, &m);
15: }
16:

```

TireCPMod

This model kind can be used to provide an additional excitation to the tire-road contact point for the tire models *RealTime Tire* and *Magic Formula*. This excitation can be e.g. a displacement in vertical direction as it is applied by a testrig or a different friction coefficient for only one tire. They influence the interaction between tire and road. Tire CPMod models are referenced by the tire infofile (and not by the vehicle infofile as most of the other model classes) by the key *TireCPMod.Kind*. This key should be set in the *Additional Parameters* field of the Tire dialog.

Figure 5.1: Selecting a TireCPMod model called *MyModel*

5.1.3 Using parameter infofiles

For the subsystems listed in [Table 5.2](#): it is possible to keep the parameters of a user model in a separate file. These files are written in the infofile format (as described in [section 3.1 'Infofile format'](#)) and contain a key called *FileIdent* that refers to the model that the parameter belong to. E.g. the line *FileIdent = CarMaker-Brake-MyModel* indicates that this infofile belongs to a user brake model called "MyModel".

In this case the model is activated by selecting the infofile under model kind "External File" in the vehicle data set as shown in [Figure 5.2](#) for a brake model. That corresponds to setting the key *<Model>.FName* in the vehicle infofile).

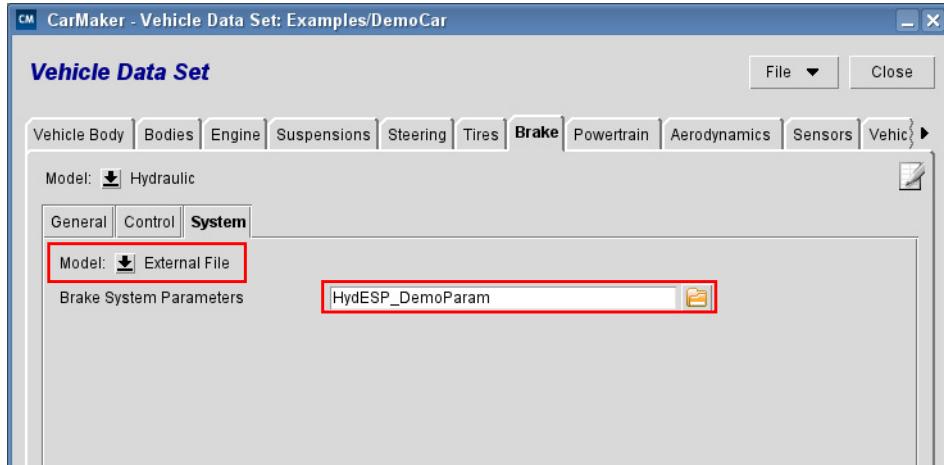


Figure 5.2: Selecting a brake model using parameter infofile

Table 5.2: Subsystems supporting parameters in an external file

Model class	Key	File Ident Value
Aero	Aero.FName	CarMaker-Aero-<Name>
Brake	Brake.FName	CarMaker-Brake-<Name>
Powertrain	PowerTrain.FName	CarMaker-Powertrain-<Name>
PTClutch	PowerTrain.Clutch.FName	CarMaker-PowerTrain.Clutch-<Name>
PTEngine	PowerTrain.ET.FName	CarMaker-PowerTrain.ET-<Name>
PTGearBox	PowerTrain. GearBox.FName	CarMaker-PowerTrain.GearBox-<Name>
Steering	Steering.FName	CarMaker-Steering-<Name>
SuspKnC	Susp<F/R>.<Kin/ Com>.0.FName	CarMaker-SuspKnC-<Name>
SuspExtFrcs	SuspExtFrcs.FName	CarMaker-SuspExtFrcs-<Name>
SuspEF_Buffer	SuspExtFrcs.Buffer.FName	CarMaker-SuspEF_Buffer-<Name>
SuspEF_Damper	SuspExtFrcs.Damper. FName	CarMaker-SuspEF_Damper-<Name>
SuspEF_Spring	SuspExtFrcs.Spring. FName	CarMaker-SuspEF_Spring-<Name>
SuspEF_Stabi	SuspExtFrcs.Stabi. FName	CarMaker-SuspEF_Stabi-<Name>
Tire	Tire.FName	CarMaker-Tire-<Name>
Vehicle	- ¹	CarMaker-Car-<Name>
VehicleControl	VehicleControl.FName	CarMaker-VehicleControl-<Name>

¹ File is the vehicle infofile itself.

5.2 Demonstration examples

The CarMaker package includes several example models in the `src/ExtraModels` folder of your CarMaker project directory. Every example is ready to be included in the build process of CarMaker and comes with a corresponding vehicle data set located in the subdirectory `ExtraModels` of the `Data/Vehicle/Examples` folder, e.g. `DemoCar_MyClutch.c`.

5.2.4 Preparation

Before you can actually use the examples, some preparation is necessary. Let us integrate the model `MyBrake`.

Adding the files to your project directory

Select the option *Sources: Extra Models* when creating or updating a new project. All examples will be copied into the subdirectory `ExtraModels` of your `src` folder. Copy the files corresponding to the model you like to integrate into the `src` folder. For the brake model these are `MyBrake.c` and `MyModels.h`.

Setting up the Makefile

The *Makefile* needs to know which model you like to incorporate. Just extend the `OBJS` macro by adding the `MyBrake` model.

```
OBJS = CM_Main.o CM_Vehicle.o User.o MyBrake.o
```

Register the model

Edit the file `User.c` with your favourite ASCII editor for programming.

Include the corresponding header file `MyModels.h` in the include section.

```
#include "MyModels.h"
```

In `User_Register()` call the registering function.

```
User_Register (void)
{
    Brake_Register_MyModel();
```

Building the simulation program

In the terminal window in the `src` directory, enter the following command to build the simulation program (see also [section 1.3.2 'Building the new Executable'](#)):

```
make
```

5.2.5 Running the model

Executable Selection

To use the model you need a few additional steps:

- Start the CarMaker GUI.
- Tell it to use the `CarMaker.win32.exe` or `CarMaker.linux` executable file just created in the `src` folder.
- Call ***Application / Start & Connect***.
- Open or configure the TestRun you would like to simulate.

Model Selection

You can either

- use the prepared dataset DemoCar_MyBrake (every model comes with its own dataset) or
- parameterize it manually. To do this, open the vehicle editor and select the *Brake* tab. Use the **Add Model** functionality to add the model named *MyModel* as Identifier (or as specified in the second argument of the *Model_Register* function) and an arbitrary description to be presented in the drop-down menu.

Additional model specific parameters can be added using the *Additional Parameters* field in the *Misc* tab. Run the simulation and monitor the brake torque (*Brake.Trq_FL*) with IPGControl. Vary the parameters and observe the influence on the vehicle behavior.

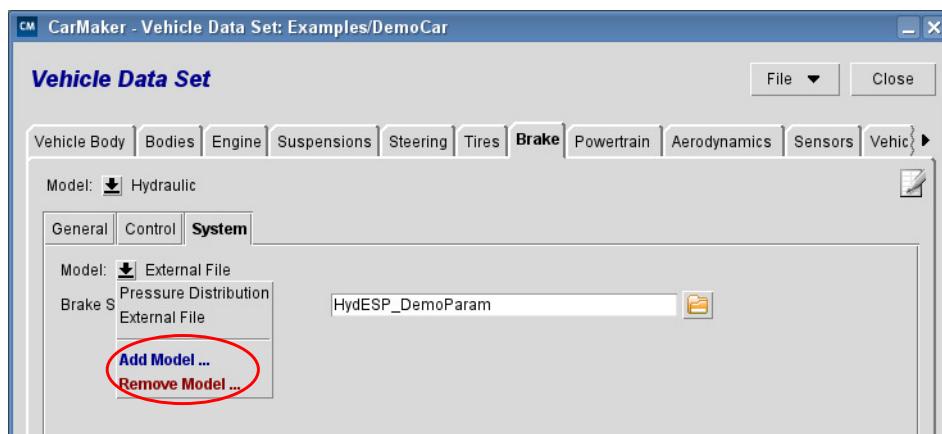


Figure 5.3: Add a model to the selection menu

Chapter 6

CarMaker for Simulink

CarMaker for Simulink is a complete integration of IPG's vehicle dynamics simulation software, CarMaker, into The MathWorks' modeling and simulation environment, Matlab/Simulink. The highly optimized and robust features of CarMaker were added to the Simulink environment using an S-Function implementation and the API functions that are provided by Matlab/Simulink. CarMaker for Simulink is not a loosely coupled co-simulation but a closely linked combination of the two best-in-class applications, resulting in a simulation environment that has both good performance and stability.

Because of this integration, it is now possible to use the power and functionality of CarMaker in the intuitive and full-featured environment of Simulink. Using CarMaker in Simulink is no different than using standard S-Function blocks or built in Simulink blocks. The CarMaker blocks are connected the same way other Simulink blocks are connected, and existing Simulink models can now easily be added to the CarMaker vehicle model with literally a few clicks of the mouse.

Integration does not, however, mean a loss of functionality, as all the features that make CarMaker the premier software in its domain have been included, and can now be used in conjunction with Simulink's rich tool set. The CarMaker GUI can still be used for simulation control and parameter adjustments, as well as defining maneuvers and road configurations. IPGControl can still be used for data analysis and graphing. IPGMovie can still be used to bring the vehicle model to life, with realistic animation and rendering of the multi-body vehicle model in 3-dimensional space.

Access to CarMaker simulation results is possible using the `cmread` utility that can be called from within Matlab. This utility loads the data of any CarMaker simulation results file into the Matlab workspace. After that, the data can be manipulated and viewed, e.g. for post processing purposes, using any of the available Matlab tools.

In short, CarMaker for Simulink is not a stripped down version of the original, but a complete system that can become a part of any Simulink simulation quickly and easily. This chapter of the Programmer's Manual will show you just how easy it can be.

6.1 CarMaker for Simulink basics

6.1.1 Starting Matlab

In your CarMaker project directory, the `src_cm4sl` subdirectory is intended as the default place where to keep the CarMaker for Simulink models. This should also be the place from where to start Matlab.



Hint: If you are using MotorcylceMaker you may have to change the subdirectory to `src_mm4sl` respectively to `src_tm4sl` if you are using TruckMaker.

You will find several example TestRuns when you click on *File > Open*, but you can also generate a completely new TestRun by yourself.

Matlab and CarMaker for Simulink work together by extending Matlab's search path, so that Matlab knows where to find CarMaker for Simulink's blockset and auxiliary commands. The Matlab search path is extended by execution of a small script named `cmenv.m`, which is contained in the `src_cm4sl` subdirectory of a CarMaker project folder. Execution of this script may be done manually, but there is also a way to invoke it automatically each time Matlab is started or a model is loaded.

The golden rule is: Always keep the `cmenv.m` script in the same directory as your Simulink model. Whenever you load a Simulink model which contains the CarMaker subsystem block from the CarMaker for Simulink blockset, `cmenv.m` will be executed automatically. This default behavior should be sufficient for most uses of CarMaker for Simulink.

Alternatively, you may want to make use of Matlab's startup file. I.e. a file called `startup.m` in Matlab's working directory will be executed automatically when Matlab is started. Just make sure that `cmenv` is invoked in your `startup.m` file and you are done. The advantage of this method is that the search path is setup for CarMaker without having to load a Simulink model first. See your Matlab documentation for further details about the Matlab startup options.



Starting Matlab under Unix

In your shell (probably running in an `xterm` or some other console window) change to the `src` subdirectory of your CarMaker project directory, then simply start Matlab:

```
% cd </path/to/your/project_dir>/src_cm4sl  
% matlab
```

Depending on how you set up your working directory (see above), you may have to load a CarMaker for Simulink model or invoke `cmenv.m` manually:

```
>> cmenv
```

The `cmenv.m` script should issue a message like the following in your Matlab console window:

```
CarMaker directory: /opt/ipg/hil/linux-X.X  
addpath /opt/ipg/hil/linux-X.X/Matlab  
addpath /opt/ipg/hil/linux-X.X/Matlab/v7.2-r2006a  
addpath /opt/ipg/hil/linux-X.X/CM4SL  
addpath </path/to/your/project_dir>/src_cm4sl  
Initialize CarMaker for Simulink.  
Done.
```



Starting Matlab using Windows Explorer

Starting Matlab under Windows

In Windows Explorer, simply double click on a Simulink model's `.mdl` file in the `src_cm4sl` subdirectory of your CarMaker project directory. This will automatically start Matlab with `src_cm4sl` as its current working directory.

Starting Matlab using a desktop shortcut

An alternative way of starting Matlab is to create a desktop shortcut to one of your Simulink model's `.mdl` file. In the shortcut's properties dialog, set the working directory to the path the `src_cm4sl` subdirectory of your CarMaker project directory. Matlab may then be started by double-clicking on the desktop shortcut's icon and your model will be loaded automatically.

Depending on how you set up your working directory (see above), you may have to load a CarMaker for Simulink model or invoke `cmenv.m` manually:

```
>> cmenv
```

The `cmenv.m` script should issue a message similar to the following in your Matlab console window:

```
CarMaker directory: c:/ipg/hil/win32-X.X
addpath c:/ipg/hil/win32-X.X/Matlab
addpath c:/ipg/hil/win32-X.X/Matlab/v7.2-r2006a
addpath c:/ipg/hil/win32-X.X/CM4SL
addpath <x:/path/to/your/project_dir>/src_cm4sl
Initialize CarMaker for Simulink.
Done.
```

6.1.2 Creating a new model

You can start building your own model by extending the `generic.mdl` model that is part of every CarMaker project directory which is the recommended approach. The `generic.mdl` represents the plain CarMaker vehicle model without any extensions. However, you may create a new CarMaker for Simulink model from scratch as well.

Creating a new model is only a matter of drag and drop. Open the CarMaker for Simulink blockset (in the Simulink library, under *Blocksets & Toolboxes*, double click on the *CarMaker for Simulink* block). From the blockset window that opens drag and drop at least the following two blocks into your empty model:

- the block labeled *Open GUI*
- the subsystem block labeled *CarMaker*

The *CarMaker Model Configuration* block is optional; you may add it any time later.

Your model should now look like this:

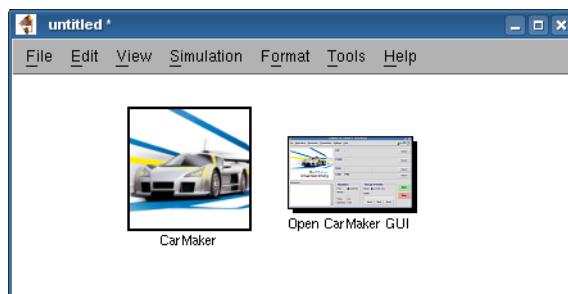


Figure 6.1: A neyle created model

CarMaker for Simulink works with all standard Simulink solvers. No special solver settings are required to run a CarMaker for Simulink model.

Normally, however, CarMaker for Simulink should be given control of the simulation, i.e. a simulation should stop according to the criteria configured in a TestRun. Examples for such criteria: A certain distance that has been driven, an error condition met and, of course, the total time elapsed. This requires the simulation stop time in Simulink to be set to `inf`, i.e. from Simulink's point of view it should run forever.

To set the simulation stop time, select **Simulation Parameters** from the model window's menu bar. The entry field for the simulation stop time is on the Solver tab.

After that, your model is ready for simulation.

6.1.3 Starting the CarMaker GUI

The single tool central to all work with CarMaker for Simulink is the CarMaker GUI. It is used for configuration of test runs and other parts of the CarMaker environment. A running CarMaker GUI is required to perform any simulation with CarMaker for Simulink.

The easiest way to start the CarMaker GUI is to double click on the *CarMaker GUI* block of your Simulink model. If no CarMaker GUI is running, a new one will be started. If a CarMaker GUI is already running, its window will pop up and come to the foreground and if you want to start a GUI with optional starting parameters type into the Matlab Command Window:

```
CMDData.GuiArgs = { '-help', '-foo', '-bar'}
```

If you want to start a GUI out of a different directory type into the Matlab Command Window:

```
CMDData.GuiExe = '/path/to/alternategui/HIL.exe"
```

The CarMaker GUI may as well be started from the Matlab console. This does not require any model to be already loaded. Use the following command:

```
>> CM_Simulink
```

6.1.4 Running a simulation

You may start and stop a simulation the way you normally do with Simulink, i.e. by invoking *Start* or *Stop* from the Simulation menu of your model window.

In CarMaker for Simulink, there is a (possibly easier accessible) alternative: The big green and red *Start* and *Stop* buttons of the CarMaker GUI.

Both ways of controlling a simulation may be mixed freely, e.g. you may start a simulation from the CarMaker GUI, then stop it via the model window and vice versa.

6.1.5 Switching between several Simulink models

In case you have several models loaded in Simulink, the question is: When using the *Start* button of the CarMaker GUI, how does the CarMaker GUI know, which model to start?

In the rare case the CarMaker GUI can not find it out by itself, it will ask you (see dialog shown below). Otherwise the CarMaker GUI will simulate the model that was simulated last.



Figure 6.2: CarMaker asking which model to use

To switch to a different model, either double-click on the model's *CarMaker GUI* button or explicitly invoke a simulation from that model's window. Next time you press the green *Start* button in the CarMaker GUI, the new model will be simulated.

6.1.6 Switching between several CarMaker project directories

During your Matlab session with CarMaker for Simulink you may change Matlab's current working directory to another CarMaker project directory. CarMaker for Simulink will notice the change and ask you what to do. Normally changing to another project directory requires also a project folder change in the GUI. This can be done with a GUI restart or manually with following command :

```
cmcmd('setprojectdir')
```

The command tells the CarMaker GUI to change to Matlab's current working directory, performing the equivalent action of invoking *File > Project Folder > (directory)* manually in the CarMaker GUI. Valuable for test automation purposes.

6.1.7 Dealing with the start values of your model

In the CarMaker GUI use *Simulation / Determine Start Values* to take a snapshot of the vehicle state at an arbitrary time during a TestRun. CarMaker will store the current vehicle state into the *SimOutput/<hostname>/Snapshot.info* file. See the CarMaker Reference Manual for details.

Use the *cmstartcond* command on the Matlab console to read the start values into the Matlab workspace. The workspace variables containing the start values can then be used to parametrize blocks of the Simulink model. This may be done automatically each time a simulation starts using one of Simulink's model callback functions. See the Simulink manual for details.

6.1.8 Upgrading to a new CarMaker version

When upgrading to a new CarMaker version, always be sure to update your *cmenv.m* script as well:

- The *cmenv.m* script is updated automatically when you start a project update from the CarMaker main GUI (*File > Update Project*).

- It can be also replaced manually with its successor, to be found in *Templates/Car/src_cm4sl/cmenv.m* in the installation directory of the new CarMaker for Simulink version.
- Or change the setting of the *cminstdir* variable in the first few lines of your *cmenv.m* script and make it point to the new version.



There might be other files, too, that must be updated. The Release Notes of every CarMaker version will provide you with the necessary information.

In case you are working with a CarMaker model library that you have compiled and linked yourself (i.e. you have made some changes to the C code) it is also required that you rebuild the model library. When you invoke *libcarmaker4sl* from the Matlab command line, the *Application.Version* line shows you (in parentheses) the CarMaker for Simulink version your model library was built for.

6.2 The CarMaker interface blockset

As an introduction to the CarMaker for Simulink interface the additional Simulink blockset for the usage with CarMaker will be presented. It serves as a toolbox with very useful elements that help to create user defined models. The CarMaker interface blockset contains blocks that serve the purpose of directly connecting a Simulink model with CarMaker. The blocks may be used as an alternative or in addition to the Simulink's standard *Inport* and *Outport* blocks at a model's top level.

You may find the blockset in the Simulink library browser, under *Blocksets and Toolboxes*. Double click on *CarMaker4SL*, or simply type

```
>> CarMaker4SL
```

in Matlab's command window.

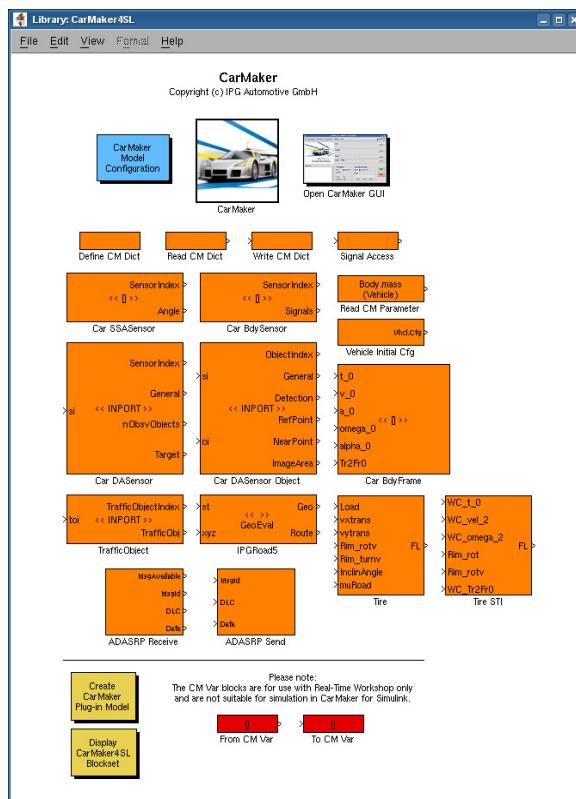


Figure 6.3: Simulink library browser (Unix): CarMaker interface blockset

6.2.1 General information

Block properties

The CarMaker for Simulink blocks of the CarMaker subsystem are *direct feed through* and run with a fixed step size of 1 ms, independently of the rest of the model.

CarMaker for Simulink blocks do not have continuous states in the Simulink sense of the word. They do have internal state variables but these are integrated using CarMaker internal solvers independently of Simulink.

Modeling principles

The CarMaker for Simulink simulation model is made up of a subsystem containing a chain of individual blocks. When Simulink executes CarMaker, all blocks of this CarMaker block chain must be executed in order and exactly once. Algebraic loops involving individual blocks of the CarMaker block chain are not permitted.

Replacing existing CarMaker functionality with Simulink blocks can be done by

- overriding signals and
- disabling unneeded internal CarMaker functionality using special parameters

It can **NOT** be done by

- replacing, removing or reordering CarMaker blocks



For further details please see the demonstration examples and their description in this manual.

Purpose of the Sync_In and Sync_Out ports

The *Sync_In* and *Sync_Out* ports are an important concept in CarMaker for Simulink.

- They guarantee proper order of execution of the CarMaker blocks.
- They let you choose exactly when a CarMaker dictionary variable is accessed with a *Read CM Dict* or *Write CM Dict* block. You may want to read the most up to date value of a variable (and not the value calculated in the previous cycle) or override the value of a variable only after it has been calculated internally by CarMaker.

6.2.2 Utility blocks

Open GUI

The *Open GUI* block is used to connect the Simulink model to a running CarMaker GUI process. If no running GUI can be detected the user will be prompted to open a new CarMaker GUI, to manually reconnect from the undetected CarMaker GUI, or to cancel the attempt to connect to the CarMaker GUI.

Double clicking the *Open GUI* block may also be used to switch between models, i.e. to “activate” a certain model in case several models are loaded in Matlab. The “active” model is the one that will be simulated when you click the *Start* button in the CarMaker GUI.



Open GUI

Figure 6.4: CarMaker *Open GUI* block

CarMaker Model Configuration

This block has been provided as a place to keep some settings specific to a CarMaker for Simulink model. The block is not required in order to run a model.

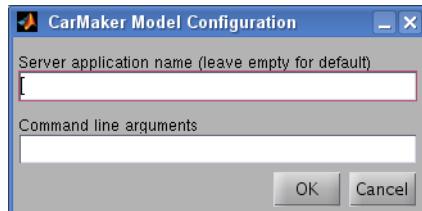


Figure 6.5: CarMaker Model Configuration dialog

Server application name

The server application name is the name under which you can identify your current CarMaker for Simulink session in e.g. IPGControl. Try **File / Connect To Application...** in the IPGControl main window to see the list of all APO applications currently running on hosts of your network. Each line displayed in this list is the server application name of a running APO application.

If your model does not contain a *CarMaker Model Configuration* block or the *Server application name* entry of the block is left empty, something like *CarMaker X.X - Car_Generic* is chosen as the server application name of your current CarMaker for Simulink session.

Almost all example models in the *src_cm4sl* subdirectory of the CarMaker project folder use their own model specific server application name.

Command line arguments

This entry makes it possible to pass special options to CarMaker when running your CarMaker for Simulink model.



Besides the options described in this manual, being able to specify CarMaker command line arguments is mainly intended for internal development and debugging purposes. Enter **-help** and start a simulation to get a generic list of available CarMaker command line options. Please note that not all options displayed have been verified to make much sense in CarMaker for Simulink or even yield meaningful results. Use this feature with care.

6.2.3 Accessing the CarMaker dictionary

The CarMaker interface blockset contains two blocks, *Read CM Dict* and *Write CM Dict*, for read and write access of CarMaker dictionary variables. The variable to be accessed is simply specified by its name.

When a non existent dictionary variable is given as a parameter of a *Read CM Dict* or *Write CM Dict* block, CarMaker will report the following error at the start of a simulation:

Model 'abcde': Missing quantity in data dict: 'xyz'

Read CM Dict

The *Read CM Dict* block reads a variable in the CarMaker dictionary and provides its current value on the block's output port. The variable needs not to be defined with a *Define CM Dict* block in the model; any existing dictionary variable may be read.

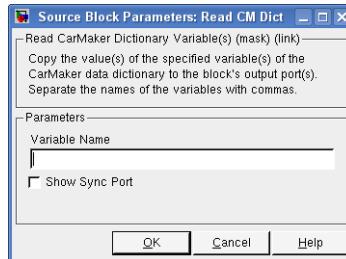


Figure 6.6: *Read CM Dict* block parameters dialog

Enter the name of the variable in the block parameters dialog of the block. In the model, the name will be displayed inside the block's symbol.

Signal Access

The signal access block's main purpose is to check the value of a signal and make it accessible by IPGControl and the DVA write functionality. In general, the input is the same as the output value, as long as no DVA write command was applied on this signal. If the latter is the case, the output delivers the new value of the signal after the DVA write access.

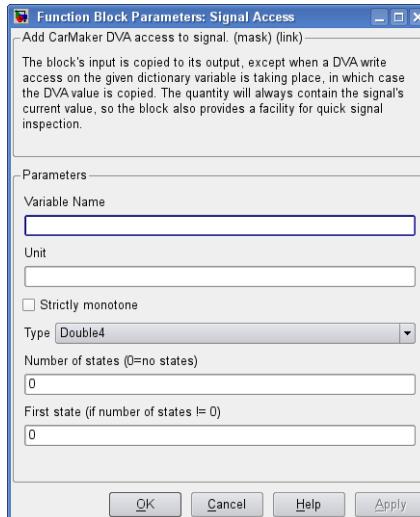


Figure 6.7: *Signal Access* block parameters dialog

Write CM Dict

The *Write CM Dict* block writes the current value at the block's input port to a variable in the CarMaker dictionary. The variable needs not to be defined with a *Define CM Dict* block in the model; any existing dictionary variable may be written to.

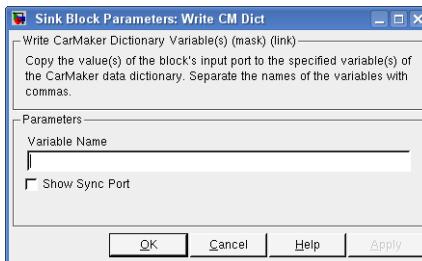


Figure 6.8: *Write CM Dict* block parameters dialog

Enter the name of the variable in the block parameters dialog of the block. In the model, the name will be displayed inside the block's symbol.

6.2.4 Defining CarMaker dictionary variables

A Simulink model can define its own variables in the CarMaker dictionary, because there may be signals that are to be monitored with IPGControl. Also, before a signal can be saved as part of the simulation results file, it must be put into the CarMaker dictionary.

When you define a dictionary variable in Simulink model, it is recommended to prefix its name with the model's name or with a convenient abbreviation. This makes it easier for you to identify the model's variables in the dictionary with tools like the CarMaker GUI or IPG-Control. Example: A dictionary variable *xyz* defined in a Simulink model called *MyModel* should be given the name *MyModel.xyz*.

A convenient shortcut exists for this purpose: The character \$ in a variable's name will be automatically expanded to the model's name. In the example above, instead of *MyModel.xyz* you may also type the shorter form *\$.xyz*.

The shortcut works for all blocks accessing the CarMaker dictionary. It saves you some key-strokes, reduces the probability of typing errors, and proves to be really valuable once the model is saved under a different name.

When defining a dictionary variable (using the *Define CM Dict* block) that is already defined somewhere else, CarMaker will report the following error at the start of a simulation:

```
Model 'abcde': Error defining quantity 'xyz'
```

Define CM Dict

The *Define CM Dict* block defines a variable in the CarMaker dictionary.

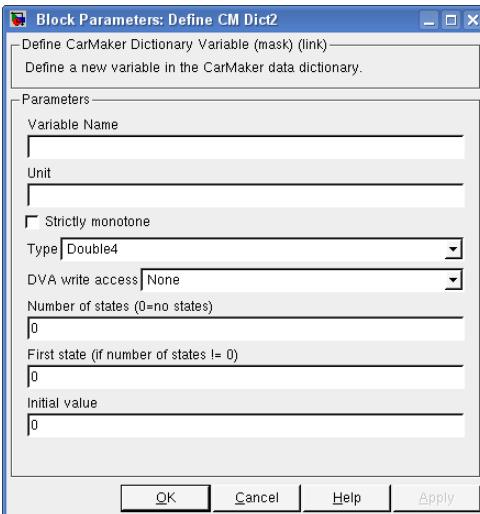


Figure 6.9: *Def CM Dict* block parameters dialog

Enter the name of the variable in the block parameters dialog of the block. In the model, the name will be displayed inside the block's symbol.

The variable may also be given a unit, this is recommended but optional. See section “User Accessible Quantities” of the Reference Manual for units used with CarMaker. Specifying a unit serves as some kind of documentation about the variable, but also allows IPGControl to display it on the same axis with other dictionary variables of the same unit.

If the variable's values are strictly monotonic increasing over time, you should check the **Strictly monotone** checkbox. Again, this is an information that tools like IPGControl need for proper display of the variable's values.

Choose the variable's type according to your needs and the range of values of the variable. You may choose between several types, when in doubt use *Double*.

For each new quantity a DVA write access point needs to be defined. Apart from the standard DVA interface points *IO_In/Out*, *DM* and *VC* you can select *none* to make the new quantity read-only. The option *private* writes a value to the quantity wherever a Write CM Dict block is used. Please find further information on the DVA interface points in [section 1.6 ‘The main cycle explained’ on page 37](#).

For discrete variables (any of the integer types), if the range of values starts at 0 and has an reasonably small upper limit (e.g. an indicator light that is either on or off), it may make sense to specify the number of discrete states of the variable. For the indicator light there would be 2 discrete values (0=off, 1=on). Again this information is provided mainly for IPG-Control which displays variables with a limited number of states in a special, space saving way. Specifying a value of 0 in this field means that no special state info is available. For the *Double* and *Float* type, the value of this field is ignored.

In case of a state variable a first state can be defined, otherwise an initial value can be specified.

Dictionary initialization

When CarMaker for Simulink has been started, but before the first simulation is run, the CarMaker dictionary does not yet know about any additional dictionary variables that will be defined in your model. To make these variables known to CarMaker for Simulink at startup, you may create a file called *startup.dict* in your model directory, that describes their properties.

The file is in plain ASCII and may contain empty lines, comment lines starting with “#”, and lines defining dictionary variables. Each line containing a variable definition consists of 5 columns, separated by tabs and spaces:

- Column 1: The name of the variable.
- Column 2: The unit of the variable. A “-” (minus, hyphen) character means “no unit”. The unit is used only for display purposes in tools like IPGControl.
- Column 3: The type of the variable. Here is a list of allowed types and their corresponding C type:

Dictionary Variable Type	C Data Type
double	double
double 4	double
float	float
llong	long long
ullong	unsigned long long
ulong	unsigned long
long	signed long
uint	unsigned integer
int	signed integer
ushort	unsigned short
short	signed short
uchar	unsigned char
char	signed char

- Column 4: The number of states.
- Column 5: Special properties. Again, this is used only for display purposes. If the variable is monotonic and increasing, specify “M”. Otherwise specify “-”.

Examples:

```
PT.GearBox.GearNo    -      long      0      -
PT.Engine.rotv      rad/s   float     0      -
Car.Distance        m       float     0      M
```



Please note, that the *startup.dict* script is only read once when the CarMaker for Simulink application is started. Changes become active only by restarting the whole CarMaker for Simulink application. The start of a new TestRun or a model reload are not sufficient.

6.2.5 Accessing CarMaker Infofile Parameters

Read CM Parameter

Almost all CarMaker settings related to the simulation environment, the current TestRun, vehicle parameters, model parameters pertaining to brake, powertrain, etc. are kept in so called infofiles, which are specially formatted text files with number of key-value-entries. At the beginning of a TestRun all necessary files are read by CarMaker and the individual modules of the simulation program pick their particular parameters from this collection. With the *Read CM Parameter* block a Simulink model may access numerical CarMaker infofile parameters. This includes “official” CarMaker parameters as well as user-defined ones, as long as they are kept in the standard infofiles read by the simulation program.

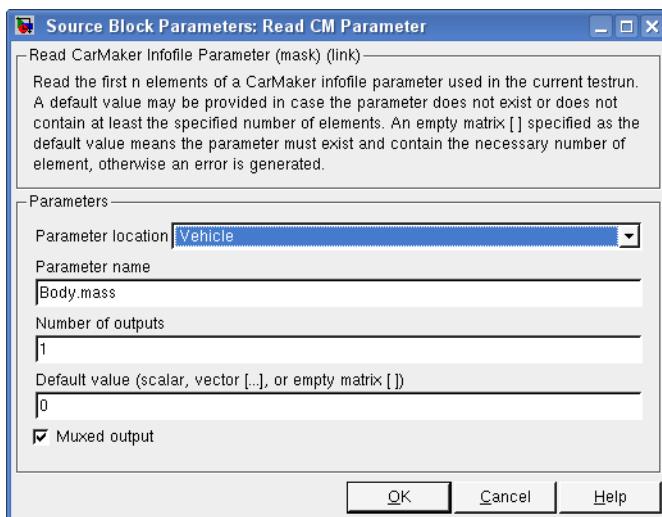


Figure 6.10: *Read CM Parameter* block parameter dialog

Parameter location

The selected location specifies more or less in which infofile the parameter is to be found. More or less, because with many parameters there is a choice whether to keep them in the vehicle parameter file or in a separate file, “external” to the vehicle parameters.

With regard to the *Read CM Parameter* block, even though you might know that in your special case e.g. the brake parameter you need is kept in the vehicle parameter file, you should select *Brake* here as this covers more cases. Specifying *Vehicle* would surely work most of the time, but once you simulate with a vehicle whose brake parameters are kept in an external file, the *Read CM Parameter* block will not be able to find your parameter since it only searches for it among the parameters read from the vehicle infofile (just as you explicitly told it to do).

Parameter name

The exact name of the parameter as stored in the infofile. The CarMaker Reference Manual provides all necessary information about parameter names, units, number of elements etc. understood by CarMaker, but of course any other user-defined parameter is allowed as well. When unsure about the name you may inspect the particular infofile directly using any appropriate text file viewer or an ASCII text editor of choice.

Number of outputs

The *Read CM Parameter* block interprets a numerical infofile parameter as a vector of values, independent of whether the parameter actually contains a scalar, a vector or a matrix. Specifying n as the number of outputs simply means that the block should provide the first n elements of the parameter as its output. If the parameter contains more elements, the remaining ones will simply be ignored. If the parameter contains less elements, the block will report an error unless a default value (see below) is provided.

Default value

The default value provides a means to cope with an optional infofile parameter (i.e. a parameter which might not always be present, in which case the default value is used) or a parameter that may contain less than the expected number of elements (as specified for *Number of outputs*). Three different forms for the *Default value* entry are possible:

- An empty matrix [] means the parameter is expected to exist in the infofile and must contain the expected number of elements. If it does not, an error message will be issued and the TestRun is aborted.
- A vector of values [...] means the parameter is optional or may contain less than the expected number of elements. The corresponding vector elements will be used as the default for all elements missing from the parameter. Please note that the length of the vector must match the specified number of outputs.
- A single scalar value means the parameter is optional or may contain less than the expected number of elements. The scalar value will be used as the default for all elements missing from the parameter.

Muxed output

The setting of this checkbox determines the number of output ports of the block:

- Activating this switch means that the block will provide the parameter's elements multiplexed on a single output port of a width matching the specified number of outputs.
- Deactivating this switch means that the block will provide each parameter element on an individual port, thus having n output ports of width 1, where n is the number of outputs specified.

Import note for CarMaker/HIL on dSPACE platforms:



Right after program startup, i.e. after downloading but before the first TestRun, the *Read CM Parameter* blocks in a CarMaker for Simulink model do not yet have access to infofile parameters. Since the Simulink model is always running (even between TestRuns), it is strongly recommended that a meaningful default value is configured for the blocks, otherwise their output signals will be zero.

6.2.6 Accessing Sensor Data

The sensor blocks in the CarMaker4Simulink library give access to parameters and quantities of all sensors defined in the vehicle model. Of course, the sensor modul's output can also be accessed through *Read CM Dict* blocks. However, the quantity names include the sensor name defined in the vehicle data set. Choosing another vehicle data set, the sensor names might be different. Thus, the quantity names change and with that the *Read CM Dict* blocks need to be modified. Using the sensor blocks for the Simulink blockset, the vehicle sensors can be addressed by their index instead of the name which makes it much easier to define an universal model extension independent on the vehicle data set.

Furthermore, only one block is required to access all data provided by a sensor.

Car SSASensor

This block lets you access the side slip angle measured by a sideslip angle sensor which is defined in the CarMaker vehicle data set on the *Sensors* tab. The sensor is identified by the name given in the CarMaker vehicle data set or by its index. Please note that the index counter starts with zero.



Figure 6.11: SSASensor block parameter dialog and the unmasked layout

The sensor output provides the sensor index and the measured sideslip angle.

Table 6.1: Output signals of Car SSASensor block

Output No.	Name	Unit	Description
0	SensorIndex	-	Index number of the sensor
1	Angle	rad	Measured sideslip angle

Car BdySensor

This block provides all data recorded by a body sensor defined in the vehicle data set in the *Sensors* tab. The sensor is identified by the name specified in the vehicle data set or by its index. Please note that the index counter starts with zero.

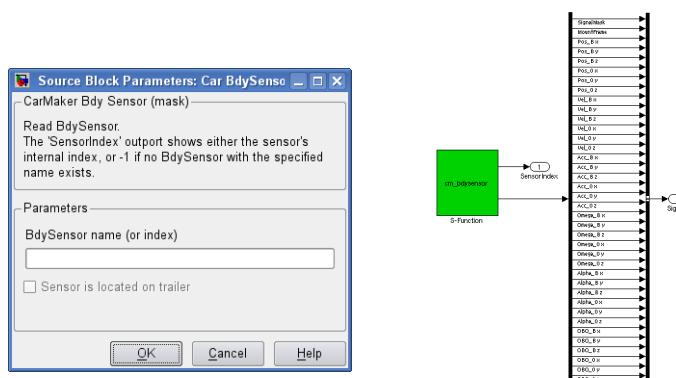


Figure 6.12: BodySensor block parameter dialog and the unmasked layout

The sensor returns the mounting frame as well as it provides access to all body sensor output quantities like position, translational and rotational velocity, translational and rotational acceleration in *Frame0* and the body mounted frame. For more information on a body sensors output signal please refer to the Reference Manual.

Table 6.2: Output signals of Car BdySensor block

Output No.	Name	Unit	Description
0	SignalMask	-	
1	MountFrame	-	Frame sensor is mounted
2 – 4	Pos_B	m	Position (body frame)
5 – 7	Pos_0	m	Position (global frame)
8 – 10	Vel_B	m/s	Velocity (body frame)
11 – 13	Vel_0	m/s	Velocity (global frame)
14 – 16	Acc_B	m/s ²	Acceleration (body frame)
17 – 19	Acc_0	m/s ²	Acceleration (global frame)
20 – 22	Omega_B	rad/s	Angle velocity (body frame)
23 – 25	Omega_0	rad/s	Angle velocity (global frame)
26 – 28	Alpha_B	rad/s ²	Angle acceleration (body frame)
29 – 31	Alpha_0	rad/s ²	Angle acceleration (global frame)
32 – 34	OBO_B	m	Position (same as block parameters)
35 – 37	O1O_0	m	Position (global frame), same as Pos_1

Car DASensor

The DA sensor block delivers information recorded by a camera sensor defined in the vehicle data set. The sensor is identified by the name specified in the vehicle data set or by its index. Please note that the index counter starts with zero.

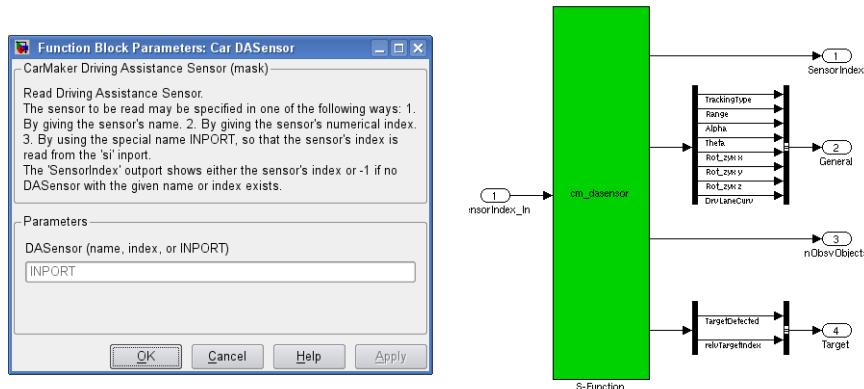


Figure 6.13: DASensor block parameter dialog and the unmasked layout

This block outputs mainly sensor configuration parameters such as the selected tracking type, range, aperture and rotation angle. Furthermore, the curvature of the actual driving lane, the number of observed objects, the target detection status and the index of the relevant target are available.

Table 6.3: Output signals of Car DASensor block

Output No.	Name	Unit	Description
0	SensorIndex	-	Index number of the sensor
1	TrackingType	-	None (0), Lane Tracking (1), Nearest Object (2)
2	Range	m	Longitudinal range of the sensor
3	Alpha	deg	Horizontal aperture of sensor beam
4	Theta	deg	Vertical aperture of sensor beam
5	Rot_zyx x	deg	Rotation around x of sensor frame according to vehicle frame (sensor viewing direction)
6	Rot_zyx y	deg	Rotation around y of sensor frame according to vehicle frame (sensor viewing direction)
7	Rot_zyx z	deg	Rotation around z of sensor frame according to vehicle frame (sensor viewing direction)
8	DrvLaneCurv	1/m	Predicted driving lane curvature of sensor
9	nObsvObjects	-	Number of observed traffic objects
10	TargetDetected	-	Flag if a relevant target is detected
11	relvTargetIndex	-	Index of the relevant target

Car DASensor Object

This block's purpose is to request information from a DA sensor on a specific traffic object. The name or index of the DA sensor specified in the vehicle data set can be assigned to this block. Please note that the index counter starts with zero.

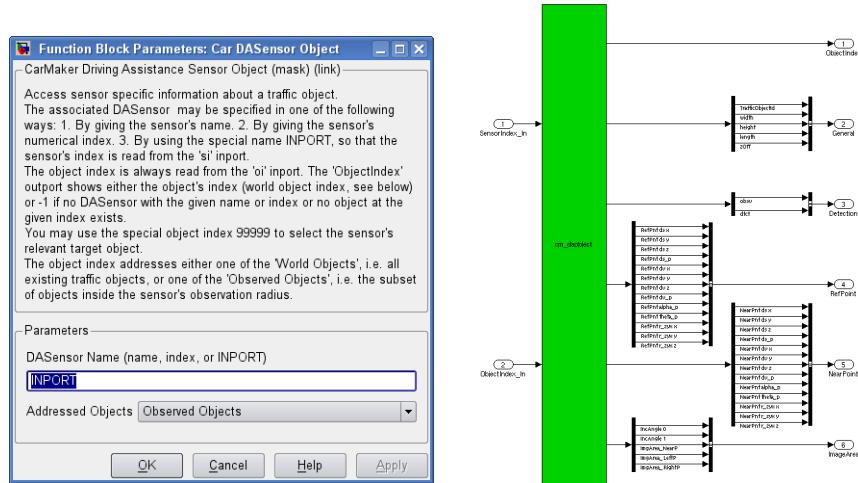


Figure 6.14: DASensor Object block parameter dialog and the unmasked layout

To identify the traffic object the information should be gathered from, the traffic object's index number can be used. By entering the index number 99999, the output information refers to the sensor's relevant target. The addressed object can be of class "World Objects" or be of the class "Observed Objects".

Output of this block is the traffic object's id, dimensions, observation state, location of reference point and nearest point as well as inclination angle and image area.

To access information on a specific traffic object independent of a DA sensor, please use the block [Traffic Object](#).

Table 6.4: Output signals of Car DASensor Object block

Output No.	Name	Unit	Description
0	ObjectIndex	-	Index number of the traffic object
1	TrafficObjectId	-	ID of the traffic object
2	width	m	Width of the traffic object (bounding box)
3	height	m	Height of the traffic object (bounding box)
4	length	m	Length of the traffic object (bounding box)
5	zOff	m	Basic offset of traffic object in z direction
6	obsv	-	Flag if traffic object is in sensor observation area
7	dtct	-	Flag if traffic object is detected
8	RefPnt ds x	m	Distance in x-direction from sensor to the traffic object's reference point
9	RefPnt ds y	m	Distance in y-direction from sensor to the traffic object's reference point
10	RefPnt ds z	m	Distance in z-direction from sensor to the traffic object's reference point
11	RefPnt ds_p	m	Absolute distance from sensor to the traffic object's reference point
12	RefPnt dv x	m/s	Speed in the reference point of the relevant target in x-direction in sensor frame
13	RefPnt dv y	m/s	Speed in the reference point of the relevant target in y-direction in sensor frame
14	RefPnt dv z	m/s	Speed in the reference point of the relevant target in z-direction in sensor frame
15	RefPnt dv_p	m/s	Relative speed between traffic object and sensor
16	RefPnt alpha_p	rad	Bearing azimuth of the traffic object
17	RefPnt theta_p	rad	Bearing elevation of the traffic object
18	RefPnt r_zyx x	rad	Orientation around x of traffic object in the reference point referring to the sensor frame
19	RefPnt r_zyx y	rad	Orientation around y of traffic object in the reference point referring to the sensor frame
20	RefPnt r_zyx z	rad	Orientation around z of traffic object in the reference point referring to the sensor frame
21	Near Pnt ds x	m	Distance in x-direction from sensor to the traffic object's nearest point
22	Near Pnt ds y	m	Distance in y-direction from sensor to the traffic object's nearest point

Table 6.4: Output signals of Car DASensor Object block

Output No.	Name	Unit	Description
23	Near Pnt ds z	m	Distance in z-direction from sensor to the traffic object's nearest point
24	Near Pnt ds_p	m	Absolute distance from sensor to the traffic object's reference point
25	Near Pnt dv x	m/s	Speed in the nearest point of the relevant target in x-direction in sensor frame
26	Near Pnt dv y	m/s	Speed in the nearest point of the relevant target in y-direction in sensor frame
27	Near Pnt dv z	m/s	Speed in the nearest point of the relevant target in z-direction in sensor frame
28	Near Pnt dv_p	m/s	Relative speed between traffic object and sensor
29	Near Pnt alpha_p	rad	Bearing azimuth of the traffic object
30	Near Pnt theta_p	rad	Bearing elevation of the traffic object
31	Near Pnt r_zyx x	rad	Orientation around x of traffic object in the nearest point referring to the sensor frame
32	Near Pnt r_zyx y	rad	Orientation around y of traffic object in the nearest point referring to the sensor frame
33	Near Pnt r_zyx z	rad	Orientation around z of traffic object in the nearest point referring to the sensor frame
34	IncAngle 0	rad	Angle of incidence in the nearest point of the traffic object
35	IncAngle 1	rad	Angle of incidence in the nearest point of the traffic object
36	ImgArea_NearP	-	Projection of the target points on the sensor image area
37	ImgArea_LeftP	-	Projection of the target points on the sensor image area
38	ImgArea_RightP	-	Projection of the target points on the sensor image area

6.2.7 Accessing C variables

The *From CM Var* or *To CM Var* blocks are intended for accessing a C variable that is not defined in the dictionary. You have to make sure that the variable exists and is declared properly in all C code modules where the variable is used.



Please note that these blocksets can only be used in combination with the Matlab Simulink Coder (formerly Realtime Workshop).

Simulink Coder's custom code library blocks might be handy for adding an #include of the proper header file for the referenced variable at the top of your model source file. This setting can be made in the Simulink model under Simulation > Configuration Parameters > Custom code > Header file by adding #include "<Headerfile name>.h". An alternative place for this is the wrapper header file of your model, which gets #include'd automatically by the generated model source code. If a non-existent C variable is specified as a parameter of a *From CM Var* or *To CM Var* block, you will very likely get warnings from the compiler about not knowing the type of the variable, and an error reported by the linker:

```
unresolved external reference: 'xyz'
```

From CM Var

The *From CM Var* block reads a C variable and provides its current value on the block's output port. The variable's name will be put without change into the model's C code.

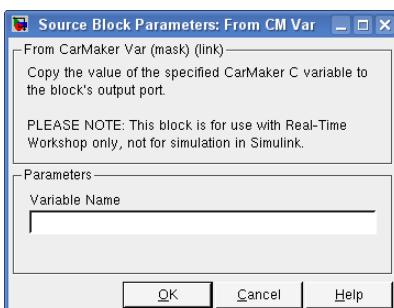


Figure 6.15: *From CM Var* block parameters dialog

Enter the name of the variable in the block parameters dialog of the block. In the model, the name will be displayed inside the block's symbol.

To CM Var

The *To CM Var* block writes the current value at the block's input port to a C variable. The variable's name will be put without change into the model's C code.

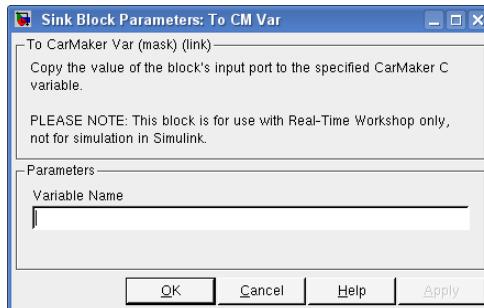


Figure 6.16: *To CM Var* block parameters dialog

Enter the name of the variable in the block parameters dialog of the block. In the model, the name will be displayed inside the block's symbol.

6.2.8 Special purpose blocks

Traffic Object

To collect information on a specific traffic object independent on a DA sensor, the traffic object block can be used. The traffic object is identified by giving the object's name or index number.

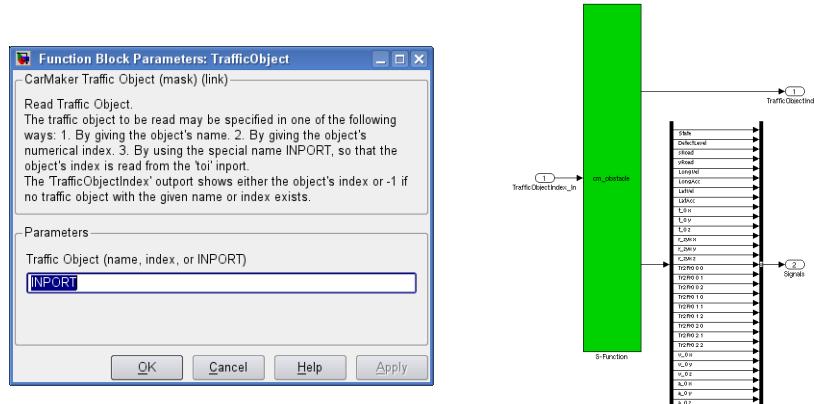


Figure 6.17: *Traffic Object* block parameter dialog and the unmasked layout

The block delivers general information on the selected traffic object such as its state, detection level, road and absolute position, rotation angles, velocity and acceleration.

Table 6.5: Output signals of Traffic Object block

Output No.	Name	Unit	Description
0	TrafficObjectIndex	-	Index of traffic object

Table 6.5: Output signals of Traffic Object block

Output No.	Name	Unit	Description
1	State	-	State of traffic object (integer): 0 = motion deactivated, object hidden 1 = motion active, object visible 2 = fixed visible object (e.g. building) 3 = free motion defined by user
2	DetectLevel	-	Detection level by a sensor of object for IPG Movie visualization (integer): 0 = not detected 1 = detected, but not crucial 2 = detected and crucial 3-7 = not set by the sensor, but could be used for user specific applications with different colors
3	sRoad	m	Traffic object road coordinate
4	yRoad	m	Traffic object's lateral road position
5	LongVel	m/s	Longitudinal velocity of the traffic object in object frame
6	LongAcc	m/s ²	Longitudinal acceleration of the traffic object in object frame
7	LatVel	m/s	Lateral velocity of the traffic object in object frame
8	LatAcc	m/s ²	Lateral acceleration of the traffic object in object frame
9	t_0 x	m	x translation of the traffic object's reference point in global frame
10	t_0 y	m	y translation of the traffic object's reference point in global frame
11	t_0 z	m	z translation of the traffic object's reference point in global frame
12	r_zyx x	rad	Rotation angle around x of traffic object
13	r_zyx y	rad	Rotation angle around y of traffic object
14	r_zyx z	rad	Rotation angle around z of traffic object
15	Tr2Fr0 0 0	-	Transformation matrix from FrTraffic to Fr0 for x coordinate
16	Tr2Fr0 0 1	-	Transformation matrix from FrTraffic to Fr0
17	Tr2Fr0 0 2	-	Transformation matrix from FrTraffic to Fr0
18	Tr2Fr0 1 0	-	Transformation matrix from FrTraffic to Fr0
19	Tr2Fr0 1 1	-	Transformation matrix from FrTraffic to Fr0
20	Tr2Fr0 1 2	-	Transformation matrix from FrTraffic to Fr0
21	Tr2Fr0 2 0	-	Transformation matrix from FrTraffic to Fr0
22	Tr2Fr0 2 1	-	Transformation matrix from FrTraffic to Fr0
23	Tr2Fr0 2 2	-	Transformation matrix from FrTraffic to Fr0
24	v_0 x	m/s	Velocity of the traffic object in global frame
25	v_0 y	m/s	Velocity of the traffic object in global frame
26	v_0 z	m/s	Velocity of the traffic object in global frame

Table 6.5: Output signals of Traffic Object block

Output No.	Name	Unit	Description
27	a_0 x	m/s ²	Acceleration of the traffic object in global frame
28	a_0 y	m/s ²	Acceleration of the traffic object in global frame
29	a_0 z	m/s ²	Acceleration of the traffic object in global frame

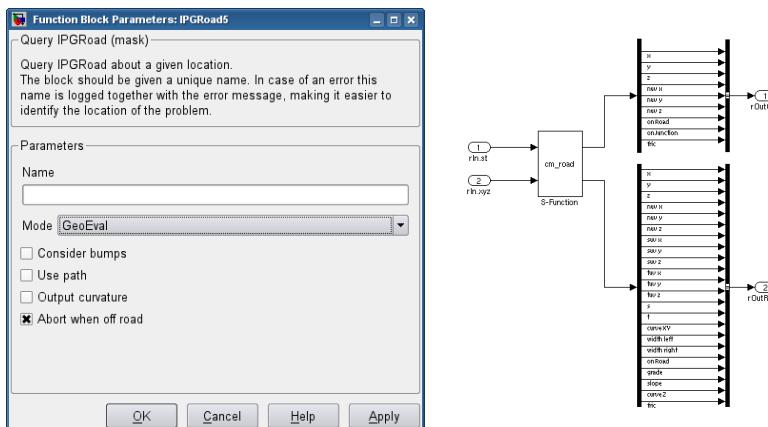
IPGRoad5

The IPGRoad5 block gives the user the possibility to evaluate the road data on a given location by the help of CarMaker's RoadEval function. The road location can be specified in different manners depending on the selected mode::

- *GeoEval*: x-, y-, z-coordinates as input, returns normal vectors, x-, y-, z-coordinates of the middle, flag onRoad, onJunction and friction as shown in the "Geo"-output bus
- *RouteEval ST*: road distance (s-coordinate) and road side (distance to the middle of the road, positive values for left side) as input, output as in "Route" output bus
- *RouteEval XY_S*: x-, y-coordinates of road point at estimated distance (s-coordinate), output as in "Route" output bus
- *RouteEval XY_SZ*: x-, y-coordinates of road point at estimated distance (s-coordinate) and altitude (z) as input, output as in "Route" output bus

Please note that the RoadEval() function only accepts cartesian coordinates, it cannot be used on GPS data.

The modes can be selected in the sensor's parameter field.

Figure 6.18: *IPGRoad* block parameter dialog and the unmasked layout

Apart from the coordinates of the defined position, the absolute track width for both sides, the coordinates and normal vectors, gradient, slope and friction values can be accessed.

Table 6.6: Output signals of IPGRoad block output "Geo"

Output No.	Name	Unit	Description
0	x	m	Road coordinate in cartesian coordinates
1	y	m	Road coordinate in cartesian coordinates
2	z	m	Road coordinate in cartesian coordinates
3	nuv x	m	Road surface normal vector x-direction
4	nuv y	m	Road surface normal vector y-direction
5	nuv z	m	Road surface normal vector z-direction
6	onRoad	-	Flag: is point on road or outside (margin)
7	onJunction	-	Flag: is point on junction or outside (margin)
8	fric	-	Friction at s,t

Table 6.7: Output signals of IPGRoad block output "Route"

Output No.	Name	Unit	Description
0	x	m	Road coordinate in cartesian coordinates
1	y	m	Road coordinate in cartesian coordinates
2	z	m	Road coordinate in cartesian coordinates
3	nuv x	m	Road surface normal vector x-direction
4	nuv y	m	Road surface normal vector y-direction
5	nuv z	m	Road surface normal vector z-direction
6	suv x	m	s-direction (longitudinal) unit vector in x-dir
7	suv y	m	s-direction (longitudinal) unit vector in y-dir
8	suv z	m	s-direction (longitudinal) unit vector in z-dir
9	tuv x	m	t-direction (lateral) unit vector in x-dir
10	tuv y	m	t-direction (lateral) unit vector in y-dir
11	tuv z	m	t-direction (lateral) unit vector in z-dir
12	s	m	Calculated pos. in route coord. s
13	t	m	Calculated pos. in route coord. t
14	curveXY	1/m	Curvature in x,y plane
15	width left	m	Road width left
16	width right	m	Road width right
17	onRoad	m	Flag: is point on road or outside (margin)
18	grade	rad	Longitudinal gradient
19	slope	rad	Lateral gradient
20	curveZ	1/m	Longitudinal z curvature
21	fric	-	Friction at s,t

Vehicle Initial Cfg

The CarMaker for Simulink interface provides the possibility to replace the whole CarMaker vehicle model by a user defined vehicle model. Only in that case, the Vehicle Initial Cfg block can be used. It provides some configuration information determined at simulation start, e.g. the user vehicle's position, roll, yaw and pitch angle as well as rotation speed of engine, gearbox and each wheel.

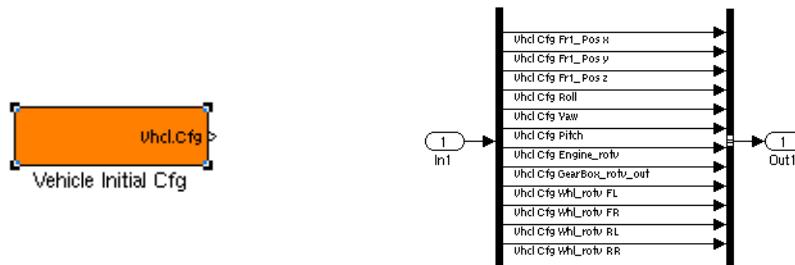


Figure 6.19: *Vehicle Initial Cfg* block parameter dialog and the unmasked layout

Table 6.8: Output signals of Vehicle Initial Cfg block

Output No.	Name	Unit	Description
0	Vhcl Cfg Fr1_Pos x	m	Location of Frame1 origin
1	Vhcl Cfg Fr1_Pos y	m	Location of Frame1 origin
2	Vhcl Cfg Fr1_Pos z	m	Location of Frame1 origin
3	Vhcl Cfg Roll	rad	Initial vehicle roll angle
4	Vhcl Cfg Yaw	rad	Initial vehicle yaw angle
5	Vhcl Cfg Pitch	rad	Initial vehicle pitch angle
6	Vhcl Cfg Engine_rotv	rad/s	Rotational speed of engine output shaft
7	Vhcl Cfg GearBox_rotv_out	rad/s	Rotation speed of gearbox output shaft
8	SimCore Trailer nTrailers	-	Number of trailers
9	Vhcl Cfg Whl_rotv FL	rad/s	Rotation speed of wheel front left
10	Vhcl Cfg Whl_rotv FR	rad/s	Rotation speed of wheel front right
11	Vhcl Cfg Whl_rotv RL	rad/s	Rotation speed of wheel rear left
12	Vhcl Cfg Whl_rotv RR	rad/s	Rotation speed of wheel rear right

ADAS RP Receive

In case the navigation software ADAS RP from NAVTEQ is available, a co-simulation with CarMaker can be established. Route information can be transmitted from the navigation software to CarMaker in realtime during the simulation via CAN. To receive CAN messages from ADAS RP in CarMaker for Simulink, this block can be used.

Please find further information about the coupling to ADAS RP in the User's Guide, appendix D "ADAS RP with CarMaker".



Figure 6.20: *Vehicle Initial Cfg* block parameter dialog and the unmasked layout

Table 6.9: Output signals of ADAS RP Receive block

Output No.	Name	Unit	Description
0	MsgAvailable	-	Flag: Is message available
1	MsgId	-	Message ID
2	DLC	-	Number of data bytes
3	Data	-	Data vector send

ADAS RP Send

Again, this block can only be used in combination with the navigation software ADAS RP from NAVTEQ. To send messages like positioning information from CarMaker to ADAS RP using CAN messages, this block can be used.

Please find further information about the coupling to ADAS RP in the User's Guide, appendix D "ADAS RP with CarMaker".

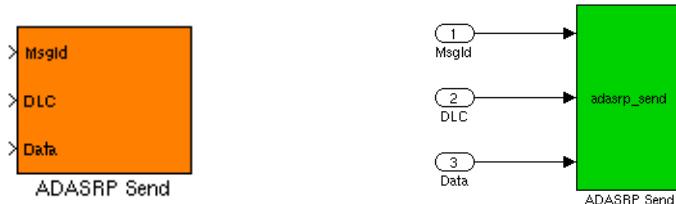


Figure 6.21: *Vehicle Initial Cfg* block parameter dialog and the unmasked layout

Table 6.10: Input signals of ADAS RP Send block

Input No.	Name	Unit	Description
0	MsgId	-	Message ID

Table 6.10: Input signals of ADAS RP Send block

Input No.	Name	Unit	Description
1	DLC	-	Number of data bytes
2	Data	-	Data vector send

Tire

In case a user defined vehicle model is used which does not include an own tire model implementation, CarMaker's RT tire model can be used by integrating this Tire block. One block is required for each wheel - the position can be stated in the block's parameters. Please find more detailed information on the IPG Tire model in the Reference Manual, section "Tire Model with Contact Point Interface (CPI)".

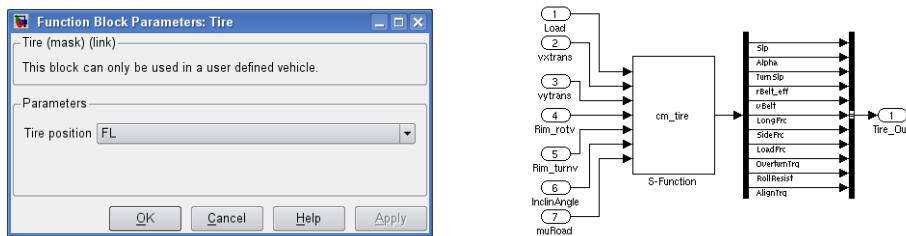
Figure 6.22: *Tire* block parameter dialog and the unmasked layout

Table 6.11: Input signals of Tire block

Input No.	Name	Unit	Description
0	Load	N	Wheel load
1	vxtrans	m/s	Velocity vector in the tire-road contact point
2	vytrans	m/s	Velocity vector in the tire-road contact point
3	Rim_rotv	m/s	Rim rotation speed (FrC)
4	Rim_turnv	m/s	Rim turning speed around vertical axis (FrW)
5	InclinAngle	rad	Inclination angle
6	muRoad	-	Friction value in road contact point

Table 6.12: Output signals of Tire block

Output No.	Name	Unit	Description
0	Slip	-	Longitudinal slip
1	Alpha	rad	Side slip angle
2	TurnSlip	-	Turn slip
3	rBelt_eff	m	Effective rolling radius
4	vBelt	m/s	Velocity of the tire belt in the tire-road contact point
5	LongFrc	N	Longitudinal force in the tire-road contact point
6	SideFrc	N	Lateral force in the tire-road contact point
7	LoadFrc	N	Normal force in the tire-road contact point

Table 6.12: Output signals of Tire block

Output No.	Name	Unit	Description
8	OverturTrq	Nm	Overturning torque in tire contact point
9	RollResist	-	Rolling resistance factor
10	AlignTrq	Nm	Self aligning torque

Tire STI

Again, this blockset is only of interest along with the integration of a user defined vehicle model. If the user model does not provide a whole tire model, the block offers the possibility to access CarMaker's Standard Tire Interface (STI) implementation. One block is required for each wheel - the position can be stated in the block's parameters. Please find more detailed information on the standard tire interface in the Reference Manual, section "Tire Model with Standard Tire Interface (STI)..

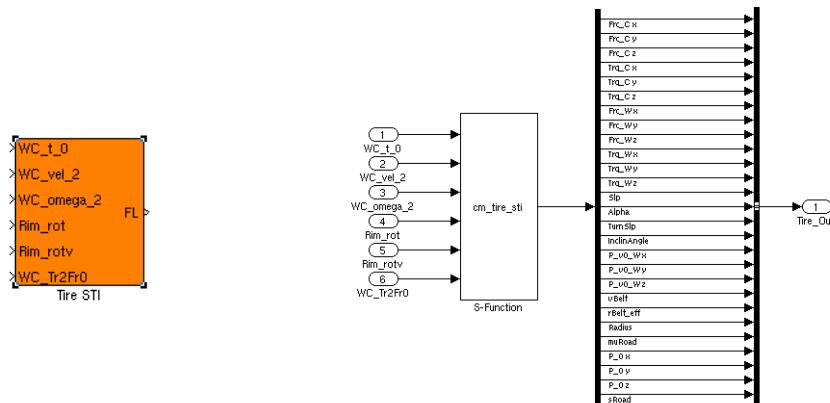


Figure 6.23: Tire STI block parameter dialog and the unmasked layout

Table 6.13: Input signals of Tire STI block

Input No.	Name	Unit	Description
0	WC_t_0	m	Position of the wheel center in global frame
1	WC_vel_2	m/s	Velocity in the wheel center in global frame
2	WC_omega_2	m/s	Wheel rotation speed in the wheel center in global frame
3	Rim_rot	rad	Rim rotation angle
4	Rim_rotv	m/s	Rim rotation speed (FrC)
5	WC_Tr2Fr0	-	Transformation matrix from wheel carrier system FrC (Fr2) to global frame Fr0

Table 6.14: Output signals of Tire STI block

Output No.	Name	Unit	Description
0	Frc_C x	N	Applied tire force to the wheel center
1	Frc_C y	N	Applied tire force to the wheel center

Table 6.14: Output signals of Tire STI block

Output No.	Name	Unit	Description
2	Frc_C z	N	Applied tire force to the wheel center
3	Trq_C x	Nm	Applied tire torque to the wheel center
4	Trq_C y	Nm	Applied tire torque to the wheel center
5	Trq_C z	Nm	Applied tire torque to the wheel center
6	Frc_W x	N	Longitudinal force in the tire-road contact point
7	Frc_W y	N	Lateral force in the tire-road contact point
8	Frc_W z	N	Nominal force in the tire-road contact point
9	Trq_W x	Nm	Tire torque in the tire-road contact point
10	Trq_W y	Nm	Tire torque in the tire-road contact point
11	Trq_W z	Nm	Tire torque in the tire-road contact point
12	Slip	-	Longitudinal slip
13	Alpha	rad	Side slip angle
14	TurnSlip	-	Turn slip
15	InclinAngle	rad	Inclination angle
16	P_v0_W x		Velocity in longitudinal direction in the tire-road contact point
17	P_v0_W y		Velocity in lateral direction in the tire-road contact point
18	P_v0_W z		Velocity in nominal direction in the tire-road contact point
19	vBelt	m/s	Velocity of the tire belt in the tire-road contact point
20	rBelt_eff	m	Effective rolling radius
21	Radius	m	Loaded radius
22	muRoad	-	Friction coefficient between tire and road
23	P_0 x	m	Position of the contact point in the global frame Fr0
24	P_0 y	m	Position of the contact point in the global frame Fr0
25	P_0 z	m	Position of the contact point in the global frame Fr0
26	sRoad	m	Road coordinate

BdyFrame

This block serves for the definition of user defined frames. Vehicle sensors can be mounted on these frames. This block is used primarily for placing CarMaker sensors on the user vehicle model.



Figure 6.24: *BdyFrame* block parameter dialog and the unmasked layout

Table 6.15: Input signals of *BdyFrame* block

Input No.	Name	Unit	Description
0	t_0	m	Position of body frame in global frame
1	v_0	m/s	Velocity of body frame in global frame
2	a_0	m/s^2	Acceleration of body frame in global frame
3	ω_0	rad/s	Angle velocity of body frame in global frame
4	α_0	rad/s^2	Angle acceleration of body frame in global frame
5	Tr2Fr0	-	Transformation matrix from FrTraffic to Fr0

6.3 CarMaker Subsystem

By opening the CarMaker block, several subsystems of the CarMaker environment and the exchanged signals become available. The tables in this chapter show the relationship between a signal name of the CarMaker subsystem, the corresponding C variable, and (if it exists) its name in the CarMaker dictionary. For most signals this information is sufficient to find the place of its exact definition and its default unit in the CarMaker Reference Manual.

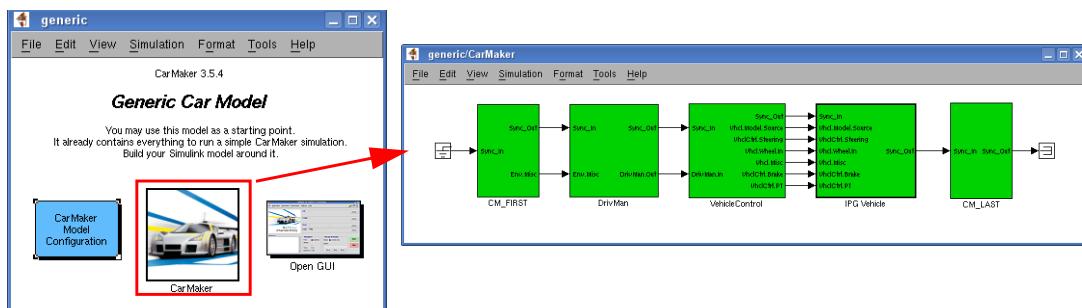


Figure 6.25: *CarMaker* block and its top level subsystems

CM_FIRST

Within this subsystem, the IO_In signals coming from external periphery (e.g. an ECU) can be manipulated as well as some environment parameters.

Environment	C Variable	Dictionary Variable
Env Temperature	Env.Temperature	Env.Temperature
Env AirDensity	Env.AirDensity	Env.AirDensity
Env AirPressure	Env.AirPressure	Env.AirPressure
Env AirHumidity	Env.AirHumidity	Env.AirHumidity
Env SolarRadiation	Env.SolarRadiation	Env.SolarRadiation
Env Time	Env.Time	Env.Time
Env WindVel_tot x	Env.WindVel_tot[0]	Env.WindVel_tot.x
Env WindVel_tot y	Env.WindVel_tot[1]	Env.WindVel_tot.y
Env WindVel_tot z	Env.WindVel_tot[2]	Env.WindVel_tot.z

DrivMan

The subsystem DrivMan provides all signals calculated in the IPGDriver and Maneuver module. If the driver's input should be replaced, this is the correct insertion point.

DrivMan	C Variable	Dictionary Variable
DrivMan SST	DrivMan.SST	DM.SST
DrivMan Key	DrivMan.Key	DM.Key
DrivMan UserSignal<i>	DrivMan.UserSignal[]	DM.UserSignal_<i>
DrivMan SelectorCtrl	DrivMan.SelectorCtrl	DM.SelectorCtrl
DrivMan GearNo	DrivMan.GearNo	DM.GearNo
DrivMan Brake	DrivMan.Brake	DM.Brake
DrivMan BrakePark	DrivMan.BrakePark	DM.BrakePark
DrivMan Clutch	DrivMan.Clutch	DM.Clutch
DrivMan Gas	DrivMan.Gas	DM.Gas
DrivMan Steering Ang	DrivMan.Steering.Ang	DM.Steer.Ang
DrivMan Steering AngVel	DrivMan.Steering .AngVel	DM.Steer.AngVel
DrivMan Steering AngAcc	DrivMan.Steering .AngAcc	DM.Steer.AngAcc
DrivMan Steering Trq	DrivMan.Steering.Trq	DM.Steer.Trq

VehicleControl

The *VehicleControl* interface should be used to insert any kind of assistance systems. Instead of overruling the driver's input by replacing the signals in the *DrivMan* section and such loosing knowledge about the driver's original intensions, the *Vehicle Control* interface interacts before the driver's requirements are passed through to the respective vehicle system.

The *DrivMan/VehicleControl* signals should be overridden using the bus between the *DrivMan* and the *VehicleControl* block - if no changes are made the *Vehicle Control* signals represent the *DrivMan* signals.

VhclCtrl	C Variable	Dictionary Variable
VhclCtrl SST	VehicleControl.SST	VC.SST
VhclCtrl Key	VehicleControl.Key	VC.Key
VhclCtrl UserSignal<i>	VehicleControl.UserSignal[]	VC.UserSignal_<i>
VhclCtrl GearNo	VehicleControl.GearNo	VC.GearNo
VhclCtrl SelectorCtrl	VehicleControl.SelectorCtrl	VC.SelectorCtrl
VhclCtrl Brake	VehicleControl.Brake	VC.Brake
VhclCtrl BrakePark	VehicleControl.BrakePark	VC.BrakePark
VhclCtrl Clutch	VehicleControl.Clutch	VC.Clutch
VhclCtrl Gas	VehicleControl.Gas	VC.Gas
VhclCtrl Steering Ang	VehicleControl.Steering.Ang	VC.Steer.Ang
VhclCtrl Steering AngVel	VehicleControl.Steering .AngVel	VC.Steer.AngVel
VhclCtrl Steering AngAcc	VehicleControl.Steering .AngAcc	VC.Steer.AngAcc
VhclCtrl Steering Trq	VehicleControl.Steering.Trq	VC.Steer.Trq
VhclCtrl Lights LowBeam	VehicleControl.Lights.Low- Beam	VC.Lights.LowBeam
VhclCtrl Lights HighBeam	VehicleControl.Lights.High- Beam	VC.Lights.HighBeam
VhclCtrl Lights Daytime	VehicleControl.Lights.Daytime	VC.Lights.Daytime
VhclCtrl Lights ParkL	VehicleControl.Lights.ParkL	VC.Lights.ParkL
VhclCtrl Lights ParkR	VehicleControl.Lights.ParkR	VC.Lights.ParkR
VhclCtrl Lights IndL	VehicleControl.Lights.IndL	VC.Lights.Ind
VhclCtrl Lights IndR	VehicleControl.Lights.IndR	VC.Lights.IndLR
VhclCtrl Lights FogFrontL	VehicleControl.Lights.Fog- FrontL	VC.Lights.FogFrontL
VhclCtrl Lights FogFrontR	VehicleControl.Lights.Fog- FrontR	VC.Lights.FogFrontR
VhclCtrl Lights FogRear	VehicleControl.Lights.FogRear	VC.Lights.FogRear
VhclCtrl Lights Brake	VehicleControl.Brake	VC.Brake
VhclCtrl Lights Reverse	VehicleControl.Lights.Reverse	VC.Lights.Reverse

As more than one Vehicle Control model can be registered using a CarMaker c-code extension, the *VehicleControlUpd* block is required. The following signal groups were kept in order to have certain signals conveniently available in certain subsystems. They should be considered as read-only, however.

VhclCtrl.Steering

VhclCtrl.Steering	C Variable	Dictionary Variable
VhclCtrl Steering Ang	VehicleControl.Steering.Ang	VC.Steer.Ang
VhclCtrl Steering AngVel	VehicleControl.Steering .AngVel	VC.Steer.AngVel
VhclCtrl Steering AngAcc	VehicleControl .Steering.AngAcc	VC.Steer.AngAcc
VhclCtrl Steering Trq	VehicleControl.Steering.Trq	VC.Steer.Trq
VhclCtrl Steering ByTrq	DrivMan.ActualMan.SteerBy	DM.SteerBy

Vhcl.Wheel.In

VhclCtrl.Wheel.In	C Variable	Dictionary Variable
Vhcl Wheel FL Trq_B2WC	Vehicle.FL.Trq_B2WC	Vhcl.FL.Trq_B2WC
Vhcl Wheel FL Trq_DL2WC	Vehicle.FL.Trq_DL2WC	Vhcl.FL.Trq_DL2WC
Vhcl Wheel FL rotv	Vehicle.FL.rotv	Vhcl.FL.rotv
Vhcl Wheel FR Trq_B2WC	Vehicle.FR.Trq_B2WC	Vhcl.FR.Trq_B2WC
Vhcl Wheel FR Trq_DL2WC	Vehicle.FR.Trq_DL2WC	Vhcl.FR.Trq_DL2WC
Vhcl Wheel FR rotv	Vehicle.FR.rotv	Vhcl.FR.rotv
Vhcl Wheel RL Trq_B2WC	Vehicle.RL.Trq_B2WC	Vhcl.RL.Trq_B2WC
Vhcl Wheel RL Trq_DL2WC	Vehicle.RL.Trq_DL2WC	Vhcl.RL.Trq_DL2WC
Vhcl Wheel RL rotv	Vehicle.RL.rotv	Vhcl.RL.rotv
Vhcl Wheel RR Trq_B2WC	Vehicle.RR.Trq_B2WC	Vhcl.RR.Trq_B2WC
Vhcl Wheel RR Trq_DL2WC	Vehicle.RR.Trq_DL2WC	Vhcl.RR.Trq_DL2WC
Vhcl Wheel RR rotv	Vehicle.RR.rotv	Vhcl.RR.rotv

Vhcl.Misc

Vhcl.Misc	C Variable	Dictionary Variable
Vhcl Trq_DL2Bdy1 x	Vehicle.Trq_DL2Bdy1[0]	-
Vhcl Trq_DL2Bdy1 y	Vehicle.Trq_DL2Bdy1[1]	-
Vhcl Trq_DL2Bdy1 z	Vehicle.Trq_DL2Bdy1[2]	-
Vhcl TrqDL2Bdy1B x	Vehicle.Trq_DL2Bdy1B[0]	-
Vhcl TrqDL2Bdy1B y	Vehicle.Trq_DL2Bdy1B[1]	-
Vhcl TrqDL2Bdy1B z	Vehicle.Trq_DL2Bdy1B[2]	-
Vhcl DL2BdyEng x	Vehicle.Trq_DL2BdyEng[0]	-
Vhcl DL2BdyEng y	Vehicle.Trq_DL2BdyEng[1]	-
Vhcl Wind_vel_0 x	Vehicle.Wind_vel_0[0]	Vhcl.Wind.vx
Vhcl Wind_vel_0 y	Vehicle.Wind_vel_0[1]	Vhcl.Wind.vy
Vhcl Wind_vel_0 z	Vehicle.Wind_vel_0[2]	Vhcl.Wind.vz

VhclCtrl.Brake

VhclCtrl.Brake	C Variable	Dictionary Variable
VhclCtrl Brake	VehicleControl.Brake	VC.Brake
VhclCtrl BrakePark	VehicleControl.BrakePark	VC.BrakePark

VhclCtrl.PT

VhclCtrl.PT	C Variable	Dictionary Variable
VhclCtrl SST	VehicleControl.SST	VC.SST
VhclCtrl Key	VehicleControl.Key	VC.Key
VhclCtrl UserSignal<i>	VehicleControl.UserSignal[]	VC.UserSignal_<i>
VhclCtrl GearNo	VehicleControl.GearNo	VC.GearNo
VhclCtrl SelectorCtrl	VehicleControl.SelectorCtrl	VC.SelectorCtrl
VhclCtrl Clutch	VehicleControl.Clutch	VC.Clutch
VhclCtrl Gas	VehicleControl.Gas	VC.Gas

Vehicle

Coming to the *Vehicle* block, several subsystems are accessible under the mask, representing the subsystems of CarMaker's vehicle model.

Brake

For the brake system, the following input and output interface signals are available:

Brake.IF.In

Brake.IF.In	C Variable	Dictionary Variable
Brake IF T_env	Brake.IF.T_env	Brake.T_env
Brake IF Trq_Reg_max FL	Brake.IF.Trq_Reg_max[0]	-
Brake IF Trq_Reg_max FR	Brake.IF.Trq_Reg_max[1]	-
Brake IF Trq_Reg_max RL	Brake.IF.Trq_Reg_max[2]	-
Brake IF Trq_Reg_max RR	Brake.IF.Trq_Reg_max[3]	-
Brake IF Trq_Reg FL	Brake.IF.Trq_Reg[0]	-
Brake IF Trq_Reg FR	Brake.IF.Trq_Reg[1]	-
Brake IF Trq_Reg RL	Brake.IF.Trq_Reg[2]	-
Brake IF Trq_Reg RR	Brake.IF.Trq_Reg[3]	-

Brake.IF.Out

Brake.IF.Out	C Variable	Dictionary Variable
Brake IF Trq_WB FL	Brake.IF.Trq_WB[0]	Brake.Trq_WB_FL

Brake.IF.Out	C Variable	Dictionary Variable
Brake IF Trq_WB FR	Brake.IF.Trq_WB[1]	Brake.Trq_WB_FR
Brake IF Trq_WB RL	Brake.IF.Trq_WB[2]	Brake.Trq_WB_RL
Brake IF Trq_WB RR	Brake.IF.Trq_WB[3]	Brake.Trq_WB_RR
Brake IF Trq_PB FL	Brake.IF.Trq_PB[0]	Brake.Trq_PB_FL
Brake IF Trq_PB FR	Brake.IF.Trq_PB[1]	Brake.Trq_PB_FR
Brake IF Trq_PB RL	Brake.IF.Trq_PB[2]	Brake.Trq_PB_RL
Brake IF Trq_PB RR	Brake.IF.Trq_PB[3]	Brake.Trq_PB_RR
Brake IF Trq_Reg_trg FL	Brake.IF.Trq_Reg_trg[0]	Brake.Trq_Reg_trg_FL
Brake IF Trq_Reg_trg FR	Brake.IF.Trq_Reg_trg[1]	Brake.Trq_Reg_trg_FR
Brake IF Trq_Reg_trg RL	Brake.IF.Trq_Reg_trg[2]	Brake.Trq_Reg_trg_RL
Brake IF Trq_Reg_trg RR	Brake.IF.Trq_Reg_trg[3]	Brake.Trq_Reg_trg_RR

Vhcl.Wheel.PT

Vhcl.Wheel.PT	C Variable	Dictionary Variable
<preSL> FL Trq_Brake	<preC>.Trq_Brake	-
<preSL> FL Trq_BrakeReg_trg	<preC>.Trq_BrakeReg_trg	-
<preSL> FL Trq_T2W	<preC>.Trq_T2W	-
<preSL> FL WhlBearing	<preC>.Trq_Whlbearing	-
<preSL> FR Trq_Brake	<preC>.Trq_Brake	-
<preSL> FR Trq_BrakeReg_trg	<preC>.Trq_BrakeReg_trg	-
<preSL> FR Trq_T2W	<preC>.Trq_T2W	-
<preSL> FR WhlBearing	<preC>.Trq_Whlbearing	-
<preSL> RL Trq_Brake	<preC>.Trq_Brake	-
<preSL> RL Trq_BrakeReg_trg	<preC>.Trq_BrakeReg_trg	-
<preSL> RL Trq_T2W	<preC>.Trq_T2W	-
<preSL> RL WhlBearing	<preC>.Trq_Whlbearing	-
<preSL> RR Trq_Brake	<preC>.Trq_Brake	-
<preSL> RR Trq_BrakeReg_trg	<preC>.Trq_BrakeReg_trg	-
<preSL> RR Trq_T2W	<preC>.Trq_T2W	-
<preSL> RR WhlBearing	<preC>.Trq_Whlbearing	-

<preSL> := Vhcl Wheel, <preC> := PowerTrain.IF.WheellIn[i]

Powertrain

The powertrain subsystem provides the rotation speed and velocity on all four wheels, engine, gearbox rotation speeds and the engine torque as well as Operation State machine and PT Control signals. Please note: The *PT IF Engine Trq (Engine Control)* signal was

implemented using the EngineControl hook and will always be one cycle late. Using the EngineControl hook in the C level interface will collide with CarMaker for Simulink's *PT Engine Trq* signal. Results will not be as expected.

PowerTrain.Misc	C Variable	Dictionary Variable
PT IF Ignition	<preC>.Ignition	PT.Control.Ignition
PT IF OperationState	<preC>.OperationState	PT.Control.OperationState
PT IF OperationError	<preC>.OperationError	PT.Control.OperationError
PT IF GearNo	<preG>.GearNo	PT.GearBox.GearNo
PT IF Engine_rotv	<preE>.rotv	PT.Engine.rotv
PT IF Engine_Trq (Engine Control)	-	PT.Engine.Trq
PT IF Trq_Supp2Bdy1 x	<pre>.Trq_Supp2Bdy1[0]	PT.Trq_Supp2Bdy1.x
PT IF Trq_Supp2Bdy1 y	<pre>.Trq_Supp2Bdy1[1]	PT.Trq_Supp2Bdy1.y
PT IF Trq_Supp2Bdy1 z	<pre>.Trq_Supp2Bdy1[2]	PT.Trq_Supp2Bdy1.z
PT IF Trq_Supp2BdyEng x	<pre>.Trq_Supp2BdyEng[0]	PT.Trq_Supp2BdyEng.x
PT IF Trq_Supp2BdyEng y	<pre>.Trq_Supp2BdyEng[1]	PT.Trq_Supp2BdyEng.y
PT IF DL_iDiff_mean	<preDL>.iDiff_mean	PT.DL.iDiff_mean
PT IF WFL rot	<preW>.rot	PT.WFL.rot
PT IF WFL rotv	<preW>.rotv	PT.WFL.rotv
PT IF WFL Trq_B2W	<preW>.Trq_B2W	PT.WFL.Trq_B2W
PT IF WFL Trq_Drive	<preW>.TrqDrive	PT.WFL.Trq_Drive
PT IF WFL Trq_Supp2WC	<preW>.Trq_Supp2WC	PT.WFL.Trq_Supp2WC
PT IF WFL Trq_BrakeReg	<preW>.Trq_BrakeReg	PT.WFL.Trq_BrakeReg
PT IF WFL Trq_BrakeReg_max	<preW>.Trq_BrakeReg_max	PT.WFL.Trq_BrakeReg_max
PT IF WFR rot	<preW>.rot	PT.WFR.rot
PT IF WFR rotv	<preW>.rotv	PT.WFR.rotv
PT IF WFR Trq_B2W	<preW>.Trq_B2W	PT.WFR.Trq_B2W
PT IF WFR Trq_Drive	<preW>.TrqDrive	PT.WFR.Trq_Drive
PT IF WFR Trq_Supp2WC	<preW>.Trq_Supp2WC	PT.WFR.Trq_Supp2WC
PT IF WFR Trq_BrakeReg	<preW>.Trq_BrakeReg	PT.WFR.Trq_BrakeReg
PT IF WFR Trq_BrakeReg_max	<preW>.Trq_BrakeReg_max	PT.WFR.Trq_BrakeReg_max
PT IF WRL rot	<preW>.rot	PT.WRR.rot
PT IF WRL rotv	<preW>.rotv	PT.WRR.rotv
PT IF WRL Trq_B2W	<preW>.Trq_B2W	PT.WRR.Trq_B2W
PT IF WRL Trq_Drive	<preW>.TrqDrive	PT.WRR.Trq_Drive
PT IF WRL Trq_Supp2WC	<preW>.Trq_Supp2WC	PT.WRR.Trq_Supp2WC
PT IF WRL Trq_BrakeReg	<preW>.Trq_BrakeReg	PT.WRR.Trq_BrakeReg
PT IF WRL Trq_BrakeReg_max	<preW>.Trq_BrakeReg_max	PT.WRR.Trq_BrakeReg_max
PT IF WRR rot	<preW>.rot	PT.WRR.rot
PT IF WRR rotv	<preW>.rotv	PT.WRR.rotv
PT IF WRR Trq_B2W	<preW>.Trq_B2W	PT.WRR.Trq_B2W
PT IF WRR Trq_Drive	<preW>.TrqDrive	PT.WRR.Trq_Drive
PT IF WRR Trq_Supp2WC	<preW>.Trq_Supp2WC	PT.WRR.Trq_Supp2WC
PT IF WRR Trq_BrakeReg	<preW>.Trq_BrakeReg	PT.WRR.Trq_BrakeReg
PT IF WRR Trq_BrakeReg_max	<preW>.Trq_BrakeReg_max	PT.WRR.Trq_BrakeReg_max

```
<preW> := PowerTrain.IF.WheelOut[pos], <preC> := PowerTrain.ControlIF,
<preG> := PowerTrain.GearBoxIF, <preE> := PowerTrain.EngineIF
<preDL> := PowerTrain.DriveLineIF
```

Steering

The steering interface provides apart from the steering wheel movement signals referring to the steering rack.

Steer.IF	C Variable	Dictionary Variable
Steering IF Ang	Steering.IF.Ang	Steer.WhlAng
Steering IF AngVel	Steering.IF.AngVel	Steer.WhlVel
Steering IF AngAcc	Steering.IF.AngAcc	Steer.WhlAcc
Steering IF Trq	Steering.IF.Trq	Steer.WhlTrq
Steering IF TrqStatic	Steering.IF.TrqStatic	Steer.WhlTrqStatic
Steering IF AssistFrc	Steering.IF.AssistFrc	Steer.AssistFrc
Steering IF AssistTrqCol	Steering.IF.AssistTrqCol	Steer.AssistTrqCol
Steering IF AssistTrqPin	Steering.IF.AssistTrqPin	Steer.AssistTrqPin
Steering IF L q	Steering.IF.L.q	Steer.L.q
Steering IF L qp	Steering.IF.L qp	Steer.L qp
Steering IF L qpp	Steering.IF.L.qpp	Steer.L.qpp
Steering IF L iSteer2q	Steering.IF.iSteer2q	Steer.L.iSteer2q
Steering IF R q	Steering.IF.R.q	Steer.R.q
Steering IF R qp	Steering.IF.R qp	Steer.R qp
Steering IF R qpp	Steering.IF.R.qpp	Steer.R.qpp
Steering IF R iSteer2q	Steering.IF.R.iSteer2q	Steer.R.iSteer2q
Steering IF L Frc	Steering.IF.L.Frc	Steer.L.Frc
Steering IF L Inert	Steering.IF.L.Inert	Steer.L.Inert
Steering IF R Frc	Steering.IF.R.Frc	Steer.R.Frc
Steering IF R Inert	Steering.IF.R.Inert	Steer.R.Inert

Vehicle Forces

This interface includes all external forces and torques acting on the vehicle. Besides wind forces coming from the aerodynamics model, tire and load forces, the user can apply virtual forces on the vehicle body. Please find further information in the Reference Manual.

Car.Virtual

Car.Virtual	C Variable	Dictionary Variable
Car Virtual Frc_1 x	Car.Virtual.Frc_1[0]	Car.Virtual.Frc_1.x
Car Virtual Frc_1 y	Car.Virtual.Frc_1[1]	Car.Virtual.Frc_1.y
Car Virtual Frc_1 z	Car.Virtual.Frc_1[2]	Car.Virtual.Frc_1.z

Car.Virtual	C Variable	Dictionary Variable
Car Virtual Frc_0 x	Car.Virtual.Frc_0[0]	Car.Virtual.Frc_0.x
Car Virtual Frc_0 y	Car.Virtual.Frc_0[1]	Car.Virtual.Frc_0.y
Car Virtual Frc_0 z	Car.Virtual.Frc_0[2]	Car.Virtual.Frc_0.z
Car Virtual Trq_1 x	Car.Virtual.Trq_1[0]	Car.Virtual.Trq_1.x
Car Virtual Trq_1 y	Car.Virtual.Trq_1[1]	Car.Virtual.Trq_1.y
Car Virtual Trq_1 z	Car.Virtual.Trq_1[2]	Car.Virtual.Trq_1.z
Car Virtual Trq_0 x	Car.Virtual.Trq_0[0]	Car.Virtual.Trq_0.x
Car Virtual Trq_0 y	Car.Virtual.Trq_0[1]	Car.Virtual.Trq_0.y
Car Virtual Trq_0 z	Car.Virtual.Trq_0[2]	Car.Virtual.Trq_0.z

Car.Aero

Car.Aero	C Variable	Dictionary Variable
Vehicle sRoadAero	Vehicle.sRoadAero	Vehicle.sRoadAero
Vehicle Wind_vel_0 x	Vehicle.Wind_vel_0[0]	Vhcl.Wind.vx
Vehicle Wind_vel_0 y	Vehicle.Wind_vel_0[1]	Vhcl.Wind.vy
Vehicle Wind_vel_0 z	Vehicle.Wind_vel_0[2]	Vhcl.Wind.vz
car Aero vRes_1 x	(car.Aero. IF.ApproachVel_1[0])	Car.Aero.vres_1.x
car Aero vRes_1 y	(car.Aero. IF.ApproachVel_1[1])	Car.Aero.vres_1.y
car Aero vRes_1 z	(car.Aero. IF.ApproachVel_1[2])	Car.Aero.vres_1.z
car Aero Frc_1 x	(car.Aero.IF.Frc_1[0])	Car.Aero.Frc_1.x
car Aero Frc_1 y	(car.Aero.IF.Frc_1[1])	Car.Aero.Frc_1.y
car Aero Frc_1 z	(car.Aero.IF.Frc_1[2])	Car.Aero.Frc_1.z
car Aero Trq_1 x	(car.Aero.IF.Trq_1[0])	Car.Aero.Trq_1.x
car Aero Trq_1 z	(car.Aero.IF.Trq_1[1])	Car.Aero.Trq_1.y
car Aero Trq_1 y	(car.Aero.IF.Trq_1[2])	Car.Aero.Trq_1.z

Car.Load

Car.Load	C Variable	Dictionary Variable
Car Load_0 mass	Car.Load[0].mass	Car.Load.0.mass
Car Load_0 tx	Car.Load[0].pos[0]	Car.Load.0.tx
Car Load_0 ty	Car.Load[0].pos[1]	Car.Load.0.ty
Car Load_0 tz	Car.Load[0].pos[2]	Car.Load.0.tz
Car Load_1 mass	Car.Load[1].mass	Car.Load.1.mass
Car Load_1 tx	Car.Load[1].pos[0]	Car.Load.1.tx
Car Load_1 ty	Car.Load[1].pos[1]	Car.Load.1.ty
Car Load_1 tz	Car.Load[1].pos[2]	Car.Load.1.tz
Car Load_2 mass	Car.Load[2].mass	Car.Load.2.mass
Car Load_2 tx	Car.Load[2].pos[0]	Car.Load.2.tx
Car Load_2 ty	Car.Load[2].pos[1]	Car.Load.2.ty

Car.Load	C Variable	Dictionary Variable
Car Load_2 tz	Car.Load[2].pos[2]	Car.Load.2.tz
Car Load_3 mass	Car.Load[3].mass	Car.Load.3.mass
Car Load_3 tx	Car.Load[3].pos[0]	Car.Load.3.tx
Car Load_3 ty	Car.Load[3].pos[1]	Car.Load.3.ty
Car Load_3 tz	Car.Load[3].pos[2]	Car.Load.3.tz

Tire

TireXX_In The *Tire_In* signals are always valid, independent of the tire model specified in the currently selected tire parameter file.

TireXX_In	C Variable	Dictionary Variable
Tire_In Load	IF->Frc_W[2] in tFcnCalc_Tire(MP, IF, dt)	Car.FzXX
Tire_In vxtrans	IF->P_v0_W[0] in tFcnCalc_Tire(MP, IF, dt)	Car.vxXX
Tire_In vytrans	IF->P_v0_W[1] in tFcnCalc_Tire(MP, IF, dt)	Car.vyXX
Tire_In Rim_rotv	IF->Rim_rotv in tFcnCalc_Tire(MP, IF, dt)	-
Tire_In Rim_turnv	IF->Rim_turnv in tFcnCalc_Tire(MP, IF, dt)	-
Tire_In Camber	IF->InclinAngle in tFcnCalc_Tire(MP, IF, dt)	Car.InclinAngleXX
Tire_In mu	IF->muRoad in tFcnCalc_Tire(MP, IF, dt)	Car.muRoadXX

TireXX_Out The *Tire_Out* signals will only be used if FileIdent is `CarMaker-Tire-CM4SL` in the currently selected tire parameter file.

TireXX_Out	C Variable	Dictionary Variable
Tire_Out Slp	IF->Slp in tFcnCalc_Tire(MP, IF, dt)	Car.LongSlipXX
Tire_Out Alpha	IF->Alpha in tFcnCalc_Tire(MP, IF, dt)	Car.SlipAngleXX
Tire_Out TurnSlip	IF->TurnSlip in tFcnCalc_Tire(MP, IF, dt)	Car.TurnSlipXX
Tire_Out rBelt_eff	IF->rBelt_eff in tFcnCalc_Tire(MP, IF, dt)	-
Tire_Out vBelt	IF->vBelt in tFcnCalc_Tire(MP, IF, dt)	Vhcl.XX.vBelt
Tire_Out LongFrc	IF->Frc_W[0] in tFcnCalc_Tire(MP, IF, dt)	Car.FxXX
Tire_Out SideFrc	IF->Frc_W[1] in tFcnCalc_Tire(MP, IF, dt)	Car.FyXX

TireXX_Out	C Variable	Dictionary Variable
Tire_Out LoadFrc	IF->Frc_W[2] in tFcnCalc_Tire(MP, IF, dt)	Car.FzXX
Tire_Out OverturnTrq	IF->Trq_W[0] in tFcnCalc_Tire(MP, IF, dt)	Car.TrqOvertXX
Tire_Out RollResist	IF->Trq_W[1] in tFcnCalc_Tire(MP, IF, dt)	Car.TrqRollXX
Tire_Out AlignTrq	IF->Trq_W[2] in tFcnCalc_Tire(MP, IF, dt)	Car.TrqAlignXX

For XX substitute one of FL, FR, RL, RR, according to the tire in question

Forces - External Suspension Forces

External Suspension Forces add a force to the internal force of the selected suspension force element calculated by CarMaker. *External Suspension Forces* can be applied on the spring, damper, buffer and stabilizer. Input signals to this interface are the force element actuation (spring length, damper velocity, buffer length, stabilizer length). Output signals are the superimposed forces.

Using the Suspension Forces hook in the C level interface will collide with the corresponding CarMaker for Simulink signals. Results will not be as expected.

Input Spring

ISpring	C Variable	Dictionary Variable
ISpring FL	ISpring[0] in Car_SuspForcesHook_Calc()	Car.SpringFL.I
ISpring FR	ISpring[1] in Car_SuspForcesHook_Calc()	Car.SpringFR.I
ISpring RL	ISpring[2] in Car_SuspForcesHook_Calc()	Car.SpringRL.I
ISpring RR	ISpring[3] in Car_SuspForcesHook_Calc()	Car.SpringRR.I

Input Damp

vDamp	C Variable	Dictionary Variable
vDamp FL	vDamp[0] in Car_SuspForcesHook_Calc()	Car.DampFL.v
vDamp FR	vDamp[1] in Car_SuspForcesHook_Calc()	Car.DampFR.v
vDamp RL	vDamp[2] in Car_SuspForcesHook_Calc()	Car.DampRL.v
vDamp RR	vDamp[3] in Car_SuspForcesHook_Calc()	Car.DampRR.v

Input Buffer

IBuf	C Variable	Dictionary Variable
IBuf FL	IBuf[0] in Car_SuspForcesHook_Calc()	Car.BufferFL.I
IBuf FR	IBuf[1] in Car_SuspForcesHook_Calc()	Car.BufferFR.I
IBuf RL	IBuf[2] in Car_SuspForcesHook_Calc()	Car.BufferRL.I
IBuf RR	IBuf[3] in Car_SuspForcesHook_Calc()	Car.BufferRR.I

Input Stabi

IStabi	C Variable	Dictionary Variable
IStabi FL	IStabi[0] in Car_SuspForcesHook_Calc()	Car.StabiFL.I
IStabi FR	IStabi[1] in Car_SuspForcesHook_Calc()	Car.StabiFR.I
IStabi RL	IStabi[2] in Car_SuspForcesHook_Calc()	Car.StabiRL.I
IStabi RR	IStabi[3] in Car_SuspForcesHook_Calc()	Car.StabiRR.I

Output Spring

FSpring	C Variable	Dictionary Variable
FSpring FL	FSpring[0] in Car_SuspForcesHook_Calc()	Car.SpringFL.Frc_ext
FSpring FR	FSpring[1] in Car_SuspForcesHook_Calc()	Car.SpringFR.Frc_ext
FSpring RL	FSpring[2] in Car_SuspForcesHook_Calc()	Car.SpringRL.Frc_ext
FSpring RR	FSpring[3] in Car_SuspForcesHook_Calc()	Car.SpringRR.Frc_ext

Output Damp

FDamp	C Variable	Dictionary Variable
FDamp FL	FDamp[0] in Car_SuspForcesHook_Calc()	Car.DampFL.Frc_ext
FDamp FR	FDamp[1] in Car_SuspForcesHook_Calc()	Car.DampFR.Frc_ext
FDamp RL	FDamp[2] in Car_SuspForcesHook_Calc()	Car.DampRL.Frc_ext
FDamp RR	FDamp[3] in Car_SuspForcesHook_Calc()	Car.DampRR.Frc_ext

Output Buffer

FBuffer	C Variable	Dictionary Variable
FBuf FL	FBuffer[0] in Car_SuspForcesHook_Calc()	Car.BufferFL.Frc_ext
FBuf FR	FBuffer[1] in Car_SuspForcesHook_Calc()	Car.BufferFR.Frc_ext
FBuf RL	FBuffer[2] in Car_SuspForcesHook_Calc()	Car.BufferRL.Frc_ext
FBuf RR	FBuffer[3] in Car_SuspForcesHook_Calc()	Car.BufferRR.Frc_ext

Output Stabi

FStabi	C Variable	Dictionary Variable
FStabi FL	FStabi[0] in Car_SuspForcesHook_Calc()	Car.StabiFL.Frc_ext
FStabi FR	FStabi[1] in Car_SuspForcesHook_Calc()	Car.StabiFR.Frc_ext
FStabi RL	FStabi[2] in Car_SuspForcesHook_Calc()	Car.StabiRL.Frc_ext
FStabi RR	FStabi[3] in Car_SuspForcesHook_Calc()	Car.StabiRR.Frc_ext

Kinetics

Trailer.Load

Trailer.Load	C Variable	Dictionary Variable
Trailer Load_0 mass	Trailer.Load[0].mass	Trailer.Load.0.mass
Trailer Load_0 tx	Trailer.Load[0].pos[0]	Trailer.Load.0.tx
Trailer Load_0 ty	Trailer.Load[0].pos[1]	Trailer.Load.0.ty
Trailer Load_0 tz	Trailer.Load[0].pos[2]	Trailer.Load.0.tz
Trailer Load_1 mass	Trailer.Load[1].mass	Trailer.Load.1.mass
Trailer Load_1 tx	Trailer.Load[1].pos[0]	Trailer.Load.1.tx
Trailer Load_1 ty	Trailer.Load[1].pos[1]	Trailer.Load.1.ty
Trailer Load_1 tz	Trailer.Load[1].pos[2]	Trailer.Load.1.tz
Trailer Load_2 mass	Trailer.Load[2].mass	Trailer.Load.2.mass
Trailer Load_2 tx	Trailer.Load[2].pos[0]	Trailer.Load.2.tx
Trailer Load_2 ty	Trailer.Load[2].pos[1]	Trailer.Load.2.ty
Trailer Load_2 tz	Trailer.Load[2].pos[2]	Trailer.Load.2.tz

Car.Hitch

Car.Fr1	C Variable	Dictionary Variable
Car Hitch tx	Car.Hitch.t_0[0]	Car.Hitch.tx
Car Hitch ty	Car.Hitch.t_0[1]	Car.Hitch.ty
Car Hitch tz	Car.Hitch.t_0[2]	Car.Hitch.tz
Car Hitch vx	Car.Hitch.v_0[0]	Car.Hitch.vx
Car Hitch vy	Car.Hitch.v_0[1]	Car.Hitch.vy
Car Hitch vz	Car.Hitch.v_0[2]	Car.Hitch.vz

Car.Trq_T2W

Car.Trq_T2W	C Variable	Dictionary Variable
Car Trq_T2WFL	Car.Tire[0].Trq_T2W	Car.CFL.Trq_T2W
Car Trq_T2WFR	Car.Tire[1].Trq_T2W	Car.CFR.Trq_T2W
Car Trq_T2WRL	Car.Tire[2].Trq_T2W	Car.CRL.Trq_T2W
Car Trq_T2WRR	Car.Tire[3].Trq_T2W	Car.CRR.Trq_T2W

Car.Fr1

Car.Fr1	C Variable	Dictionary Variable
Car Fr1 rx	Car.Fr1.r_zyx[0]	Car.Fr1.rx
Car Fr1 ry	Car.Fr1.r_zyx[1]	Car.Fr1.ry
Car Fr1 rz	Car.Fr1.r_zyx[2]	Car.Fr1.rz
Car Fr1 rvx	Car.Fr1.rv_zyx[0]	-
Car Fr1 rvy	Car.Fr1.rv_zyx[1]	-
Car Fr1 rvz	Car.Fr1.rv_zyx[2]	-

WheelCarrier

WheelCarrier.Misc	C Variable	Dictionary Variable
car Susp FL GenInert1	(car.SuspF.L.GenInert[1])	Car.CFL.GenInert1
car Susp FR GenInert1	(car.SuspF.R.GenInert[1])	Car.CFR.GenInert1
car Susp FL GenFrc1	(car.SuspF.L.GenFrc[1])	Car.CFL.GenFrc1
car Susp FR GenFrc1	(car.SuspF.R.GenFrc[1])	Car.CFR.GenFrc1

Trailer

The trailer subsystem provides an interface to the vehicle model.

Vhcl.Steering

Vhcl.Steering	C Variable	Dictionary Variable
Vhcl Steering Ang	Vehicle.Steering.Ang	Vhcl Steering Trq

Vhcl.Steering	C Variable	Dictionary Variable
Vhcl Steering AngVel	Vehicle.Steering.AngVel	Vhcl.Steer.Vel
Vhcl Steering AngAcc	Vehicle.Steering.AngAcc	Vhcl.Steer.Acc
Vhcl Steering Trq	Vehicle.Steering.Trq	Vhcl.Steer.Trq

Vhcl.Motion

Vhcl.Motion	C Variable	Dictionary Variable
Vhcl v	Vehicle.v	Vhcl.v
Vhcl Distance	Vehicle.Distance	Vhcl.Distance
Vhcl sRoad	Vehicle.sRoad	Vhcl.sRoad
Vhcl sRoadAero	Vehicle.sRoadAero	Vhcl.sRoadAero
Vhcl Hitch_Pos x	Vehicle.Hitch_Pos[0]	Vhcl.Hitch.x
Vhcl Hitch_Pos y	Vehicle.Hitch_Pos[1]	Vhcl.Hitch.y
Vhcl Hitch_Pos z	Vehicle.Hitch_Pos[2]	Vhcl.Hitch.z
Vhcl Roll	Vehicle.Roll	Vhcl.Rol
Vhcl RollVel	Vehicle.RollVel	Vhcl.RollVel
Vhcl RollAcc	Vehicle.RollAcc	Vhcl.RollAcc
Vhcl Pitch	Vehicle.Pitch	Vhcl.Pitch
Vhcl PitchVel	Vehicle.PitchVel	Vhcl.PitchVel
Vhcl PitchAcc	Vehicle.PitchAcc	Vhcl.PitchAcc
Vhcl Yaw	Vehicle.Yaw	Vhcl.Yaw
Vhcl YawRate	Vehicle.YawVel	Vhcl.YawVel
Vhcl YawAcc	Vehicle.YawAcc	Vhcl.YawAcc

Vhcl.Pol

Vhcl.Pol	C Variable	Dictionary Variable
Vhcl Pol_Pos x	Vehicle.Pol_Pos[0]	Vhcl.Pol.x
Vhcl Pol_Pos y	Vehicle.Pol_Pos[1]	Vhcl.Pol.y
Vhcl Pol_Pos z	Vehicle.Pol_Pos[2]	Vhcl.Pol.z
Vhcl Pol_Vel x	Vehicle.Pol_Vel[0]	Vhcl.Pol.vx
Vhcl Pol_Vel y	Vehicle.Pol_Vel[1]	Vhcl.Pol.vy
Vhcl Pol_Vel z	Vehicle.Pol_Vel[2]	Vhcl.Pol.vz
Vhcl Pol_Acc x	Vehicle.Pol_Acc[0]	Vhcl.Pol.ax
Vhcl Pol_Acc y	Vehicle.Pol_Acc[1]	Vhcl.Pol/ay
Vhcl Pol_Acc z	Vehicle.Pol_Acc[2]	Vhcl.Pol.az
Vhcl Pol_Vel_1 x	Vehicle.Pol_Vel_1[0]	Vhcl.Pol.vx_1
Vhcl Pol_Vel_1 y	Vehicle.Pol_Vel_1[1]	Vhcl.Pol.vy_1
Vhcl Pol_Vel_1 z	Vehicle.Pol_Vel_1[2]	Vhcl.Pol.vz_1
Vhcl Pol_Acc_1 x	Vehicle.Pol_Acc_1[0]	Vhcl.Pol.ax_1
Vhcl Pol_Acc_1 y	Vehicle.Pol_Acc_1[1]	Vhcl.Pol/ay_1
Vhcl Pol_Acc_1 z	Vehicle.Pol_Acc_1[2]	Vhcl.Pol.az_1
Vhcl Pol_GCS Long	Vehicle.Pol_GCS.Elev	Vhcl.Pol.GCS.Elev

Vhcl.Pol	C Variable	Dictionary Variable
Vhcl Pol_GCS Lat	Vehicle.Pol_GCS.Lat	Vhcl.Pol.GCS.Lat
Vhcl Pol_GCS Eval	Vehicle.Pol_GCS.Eval	Vhcl.Pol.GCS.Eval

Vhcl.Wheel.pos

Vhcl.Wheel.<pos>	C Variable	Dictionary Variable
Vhcl Wheel <pos> t x	Vehicle.<pos>.t[0]	Vhcl.<pos>.tx
Vhcl Wheel <pos> t y	Vehicle.<pos>.t[1]	Vhcl.<pos>.ty
Vhcl Wheel <pos> t z	Vehicle.<pos>.t[2]	Vhcl.<pos>.tz
Vhcl Wheel <pos> r_zxy x	Vehicle.<pos>.r[0]	Vhcl.<pos>.rx
Vhcl Wheel <pos> r_zxy y	Vehicle.<pos>.r[1]	Vhcl.<pos>.ry
Vhcl Wheel <pos> r_zxy z	Vehicle.<pos>.r[2]	Vhcl.<pos>.rz
Vhcl Wheel <pos> Fx	Vehicle.<pos>.Fx	Vhcl.<pos>.Fx
Vhcl Wheel <pos> Fy	Vehicle.<pos>.Fy	Vhcl.<pos>.Fy
Vhcl Wheel <pos> Fz	Vehicle.<pos>.Fz	Vhcl.<pos>.Fz
Vhcl Wheel <pos> vBelt	Vehicle.<pos>.vBelt	Vhcl.<pos>.vBelt
Vhcl Wheel <pos> LongSlip	Vehicle.<pos>.LongSlip	Vhcl.<pos>.LongSlip
Vhcl Wheel <pos> SideSlip	Vehicle.<pos>.SideSlip	Vhcl.<pos>.SideSlip
Vhcl Wheel <pos> Trq_T2W	Vehicle.<pos>.Trq_T2W	Vhcl.<pos>.Trq_T2W
Vhcl Wheel <pos> Trq_WhlBearing	Vehicle.<pos>.Trq_WhlBearing	Vhcl.<pos>.Trq_WhlBearing

Car.Hitch.FrcTrq

Car.Hitch.FrcTrq	C Variable	Dictionary Variable
Car Hitch Frc2Car x	Car.Hitch.Frc2Car[0]	Car.Hitch.Frc2Car.x
Car Hitch Frc2Car y	Car.Hitch.Frc2Car[1]	Car.Hitch.Frc2Car.y
Car Hitch Frc2Car z	Car.Hitch.Frc2Car[2]	Car.Hitch.Frc2Car.z
Car Hitch Trq2Car x	Car.Hitch.Trq2Car[0]	Car.Hitch.Trq2Car.x
Car Hitch Trq2Car y	Car.Hitch.Trq2Car[1]	Car.Hitch.Trq2Car.y
Car Hitch Trq2Car z	Car.Hitch.Trq2Car[2]	Car.Hitch.Trq2Car.z

6.4 Demonstration examples

CarMaker for Simulink comes with a number of example models demonstrating various applications, features and modeling techniques. The examples are complete with

- the Simulink model
- Matlab parameter files and scripts, if needed
- configuration files, i.e. TestRuns, vehicle data, tire data, etc.

Please note: The examples presented in this chapter were intended to show how to make use of a particular interface of CarMaker for Simulink, but not to provide an elaborated application or a fully fledged, 100% physically sound replacement of a particular CarMaker subsystem.

Making a local copy of the examples

All Simulink models described in this chapter can be found in the *src_cm4sl* subdirectory of a standard project directory. Make sure you did add the *CarMaker for Simulink Extras* to your project directory using the *New Project* dialog of the CarMaker GUI.

6.4.1 ACC

This model shows how a simple ACC controller used for Active Cruise Control can be integrated.

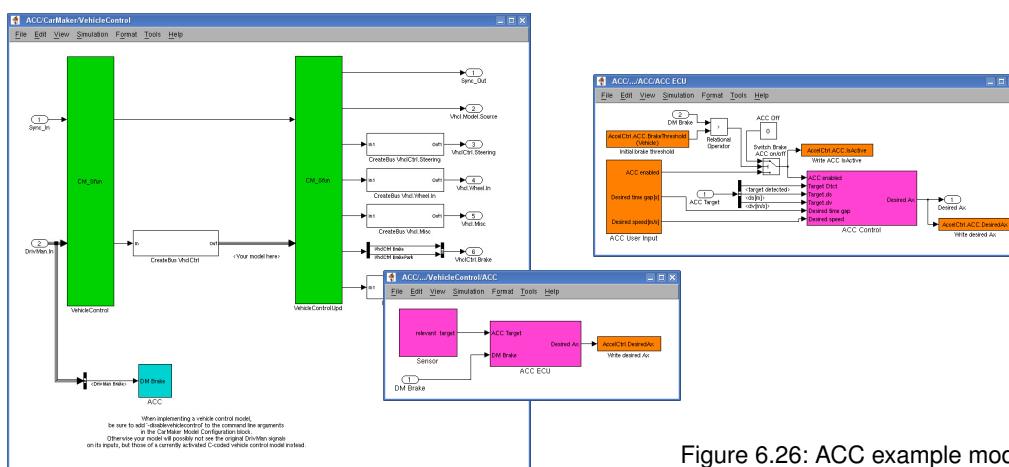


Figure 6.26: ACC example model

Files belonging to this example

- *ACC.mdl*
The Simulink example.
- *Examples/Simulink/CountryRoad_ACC*
The TestRun to be used with this example.

Details about the example

- The controller calculates a desired acceleration / deceleration. This controller output is used by the CarMaker internal acceleration controller. The brake and gas pedal are manipulated accordingly and thus the driver's wish is overwritten.
- The controller takes effect as soon as a traffic object is detected on the road.
- The model works with any TestRun containing obstacles and a vehicle model which contains the CarMaker acceleration controller (see TestRun Examples/Simulink/*CountryRoad_ACC*).

6.4.2 AccelCtrl_ACC

Another example for a ACC controller which effects both gas and brake pedal.

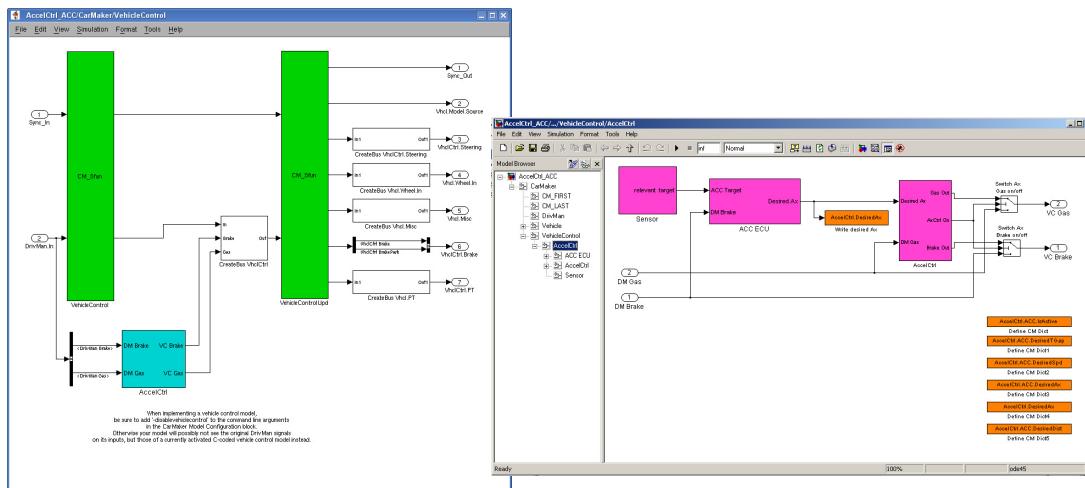


Figure 6.27: AccelCtrl_ACC example model

Files belonging to this example

- AccelCtrl_ACC.mdl

The Simulink example.

Details about the example

- This example is based on the ACC.mdl. The mere ACC controller was extended by an acceleration controller which uses the output of the ACC model to influence the gas and brake pedal. As soon as a relevant traffic object is detected, the controller slows down the vehicle.
- The model works with any TestRun containing obstacles.

6.4.3 BodyCtrl

In this example the Simulink model implements some kind of active body control, that tries to keep the vehicle upright by applying extra suspension forces to the chassis.

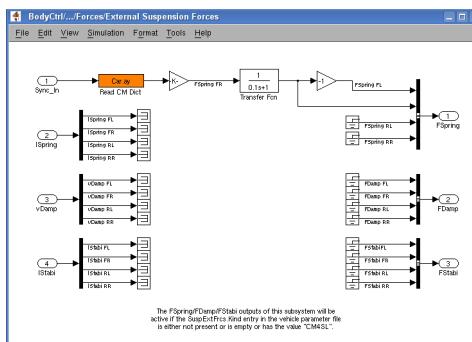


Figure 6.28: BodyCtrl example model

Files belonging to this example

- src_cm4s1/BodyCtrl1.mdl
- The Simulink model.

- Examples/Simulink/BodyCtrl
A TestRun to demonstrate the working of the model.

Details about the example

- The model is connected to CarMaker using the vehicle's external suspension forces interface.
- The *BodyCtrl* TestRun demonstrates how the chassis is kept upright while the vehicle is driving in a long curve. The model is not limited to this particular TestRun, though.
- In CarMaker for Simulink the *FSpring/FDamp/FBuf/FStabi* outputs of the *External Suspension Forces* subsystem will be active if the *SuspExtFrcs.Kind* entry in the vehicle parameter file is either not present or is empty or has the value *CM4SL*.
- This model is also available as an example of the CarMaker target for Simulink Coder (formerly Real-Time Workshop) to show how Simulink models can be integrated into CarMaker using Simulink Coder generated C code.

6.4.4 HydBrakeCU_ESP

The HydBrakeCU model provides an example of a brake control unit including a ESP controller along with an ABS and ASR model. It shows how such a controller can be integrated as part of the brake control unit.

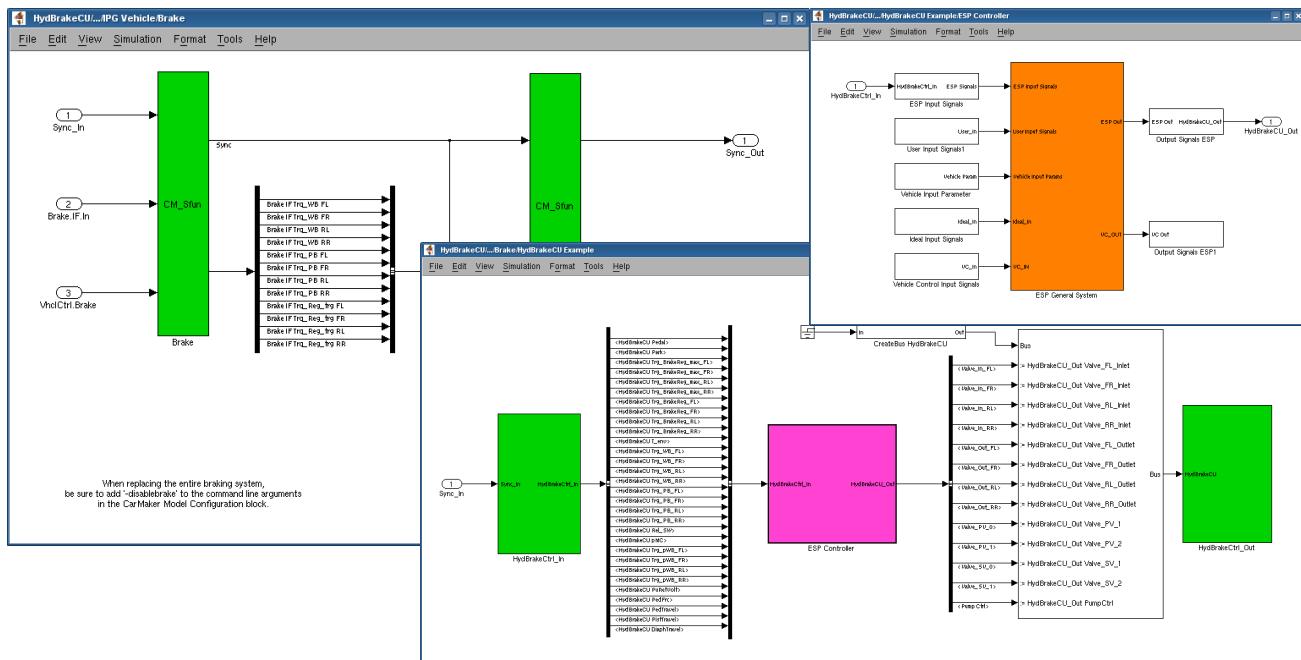


Figure 6.29: HydBrakeCU example model incl. ABS and ASR and ESP

Files belonging to this example

- *src_cm4sl/HydBrakeCU_ESP.mdl*
The Simulink example.
- *src_cm4sl/HydBrakeCU_params_ESP.m*
The parameter file for the brake model.

Details about the example

- The integrated ESP controller can trigger autonomous braking events since a ABS controller is included. The brake system can build up system pressure even without the driver's brake pedal activation.
- The integrated ABS controller uses the brake hydraulic's pilot valves instead. Thus, the wheel brake pressure can be regulated independently at each wheel, but the driver needs to build up the brake system pressure by hitting the brake pedal.
- The ASR controller uses the brake system's suction values for autonomous braking similarly to the ESP controller. Furthermore, the throttle position as input to the powertrain is adapted by the controller.
- The signal "ESP lamp" feeds the signal lamp in the Instruments panel to highlight an ESP event.
- This model is also available as an example of the CarMaker target for Simulink Coder (formerly Real-Time Workshop) to show how Simulink models can be integrated into CarMaker using Simulink Coder generated C code.

6.4.5 PTBatteryCU

This example shows how to replace CarMaker's internal battery control module on the Simulink level. It provides the interface I/O signals for a user-specific PTBatteryCU model and can be therefore understood as a model template. It was modeled after the standard Car-Maker example C code in src/ExtraModels/MyBatteryCU.c. See the source code for details.

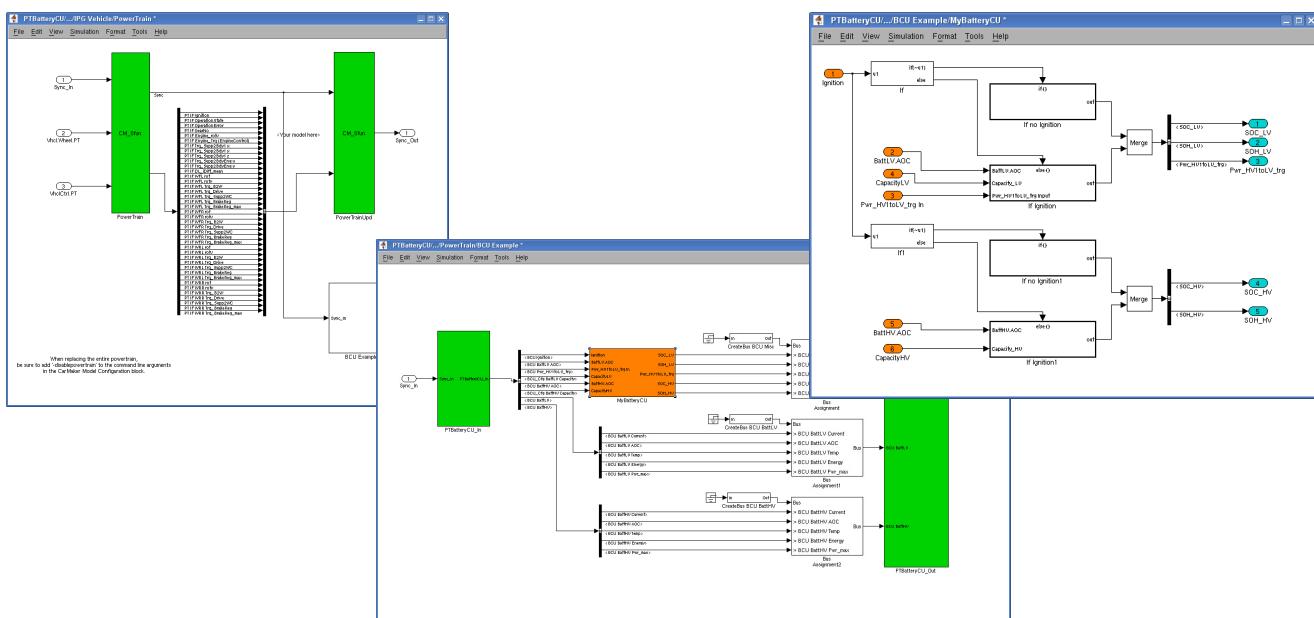


Figure 6.30: PTBatteryCU model example

Files belonging to this example

- `src_cm4sl/PTBatteryCU.mdl`
The Simulink model.

Details about the example

- A similar model is also available as an example of the CarMaker target for Simulink Coder (formerly Real-Time Workshop) to show how Simulink models can be integrated into CarMaker using Simulink Coder generated C code.

6.4.6 PTControl

This example includes a minimalist powertrain control strategy modeled inside Simulink. It provides the interface I/O signals for a user-specific PTControl model and can be therefore understood as a model template.

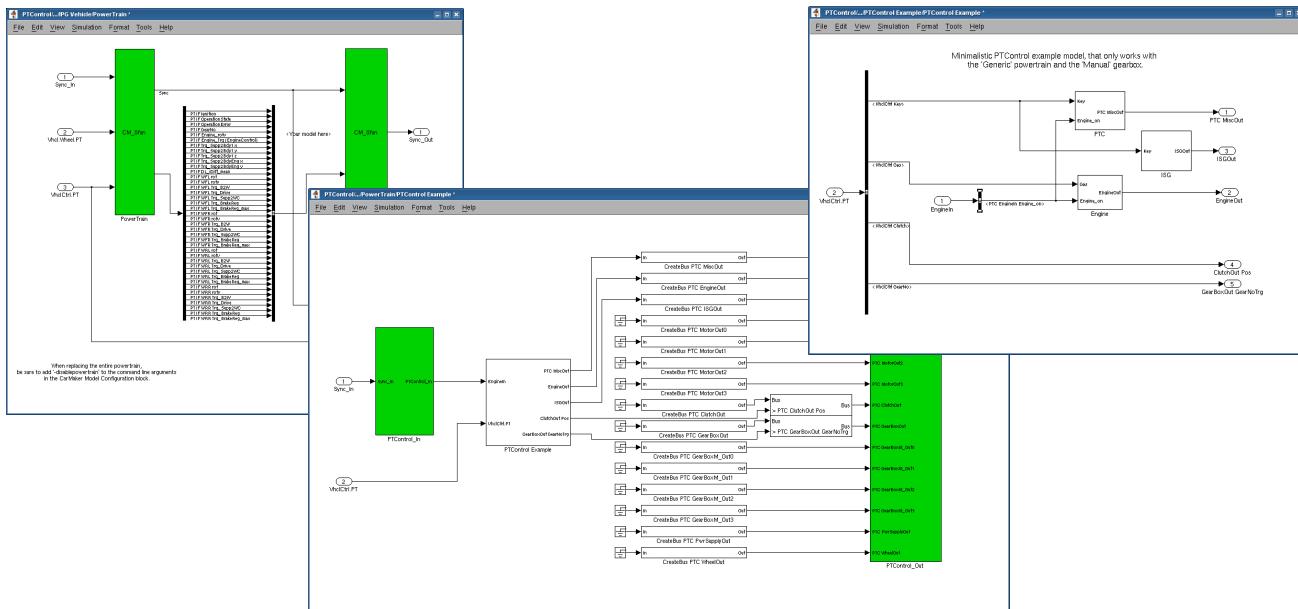


Figure 6.31: PTControl model example

Files belonging to this example

- `src_cm4s1/PTControl.mdl`
The Simulink model.

Details about the example

- This model only works with the generic powertrain and the manual gearbox
- A more complex model is available as a Simulink Coder plugin example
- The model passes the key position and the gas pedal position to the operation state machine and engine respectively

6.4.7 PTEngineCU

This example shows how to replace CarMaker's internal engine control module on the Simulink level. It provides the interface I/O signals for a user-specific PTEngineCU model and can be therefore understood as a model template. It was modeled after the standard Car-Maker example C code in src/ExtraModels/MyEngineCU.c. See the source code for details.

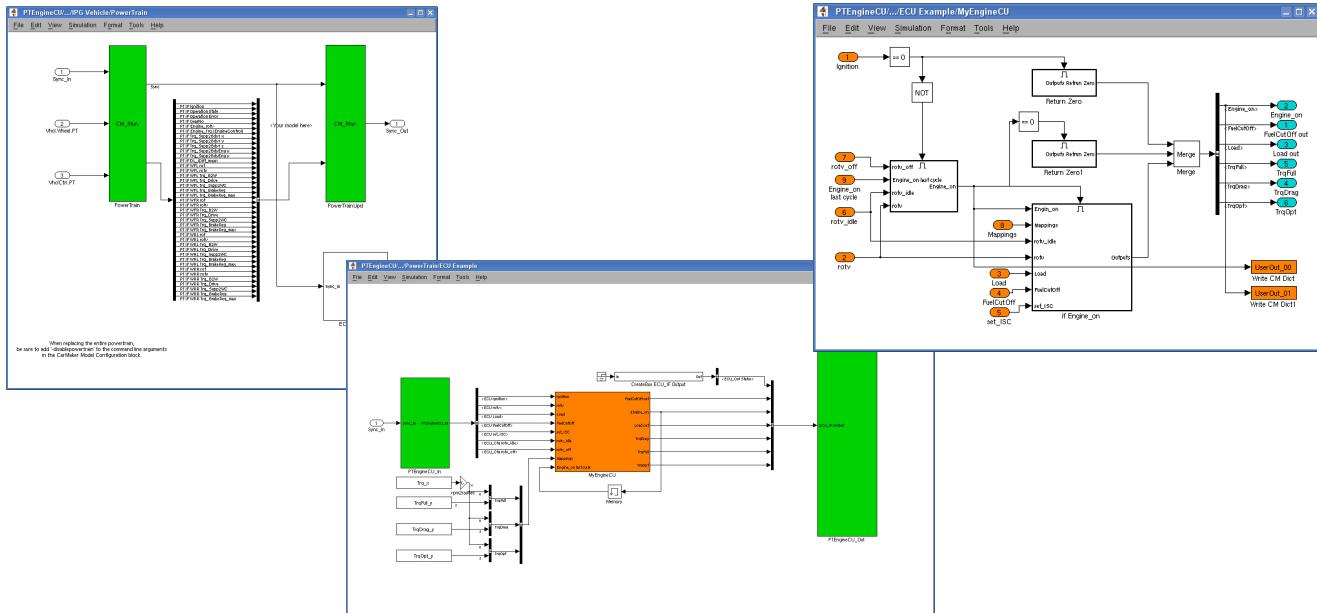


Figure 6.32: PTEngineCU model example

Files belonging to this example

- *src_cm4s1/PTEngineCU.mdl*
The Simulink model.

Details about the example

- A similar model is also available as an example of the CarMaker target for Simulink Coder (formerly Real-Time Workshop) to show how Simulink models can be integrated into CarMaker using Simulink Coder generated C code.

6.4.8 PTMotorCU

This example shows how to replace CarMaker's internal motor control module on the Simulink level. It provides the interface I/O signals for a user-specific PTMotorCU model and can be therefore understood as a model template. It was modeled after the standard Car-Maker example C code in src/ExtraModels/MyMotorCU.c. See the source code for details.

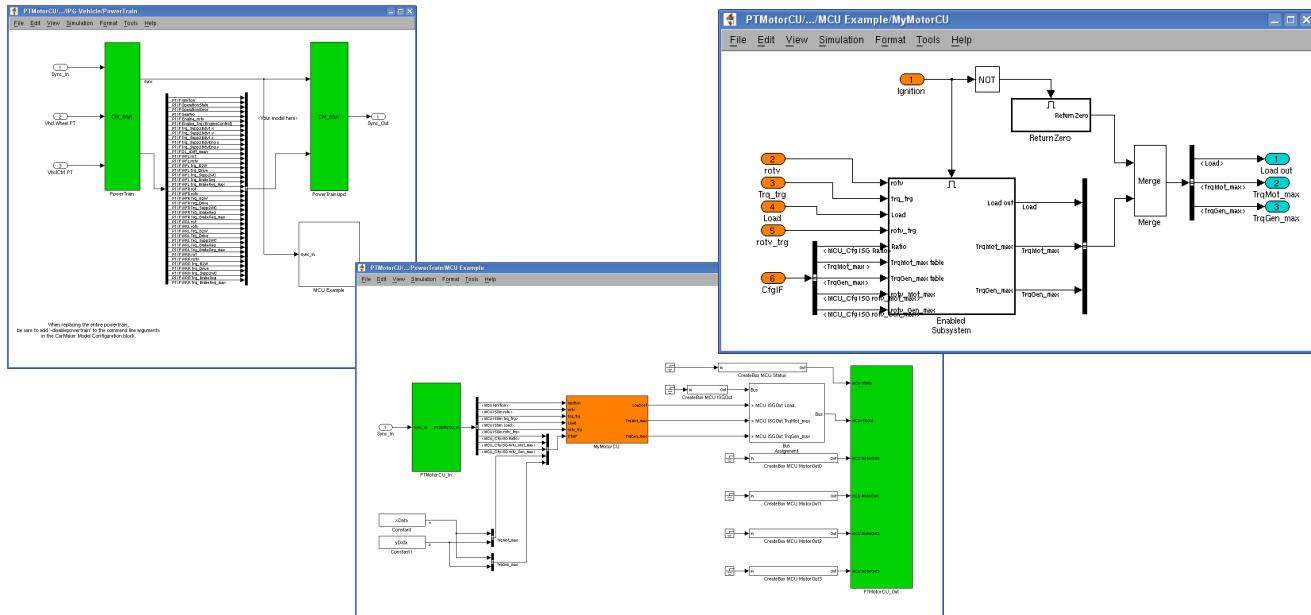


Figure 6.33: PTMotorCU model example

Files belonging to this example

- `src_cm4s1/PTMotorCU.mdl`
The Simulink model.

Details about the example

- A similar model is also available as an example of the CarMaker target for Simulink Coder (formerly Real-Time Workshop) to show how Simulink models can be integrated into CarMaker using Simulink Coder generated C code.

6.4.9 PTTransmCU

This example shows how to replace CarMaker's internal transmission control module on the Simulink level. It provides the interface I/O signals for a user-specific PTTransmCU model and can be therefore understood as a model template. It was modeled after the standard CarMaker example C code in src/ExtraModels/MyTransmCU.c. See the source code for details.

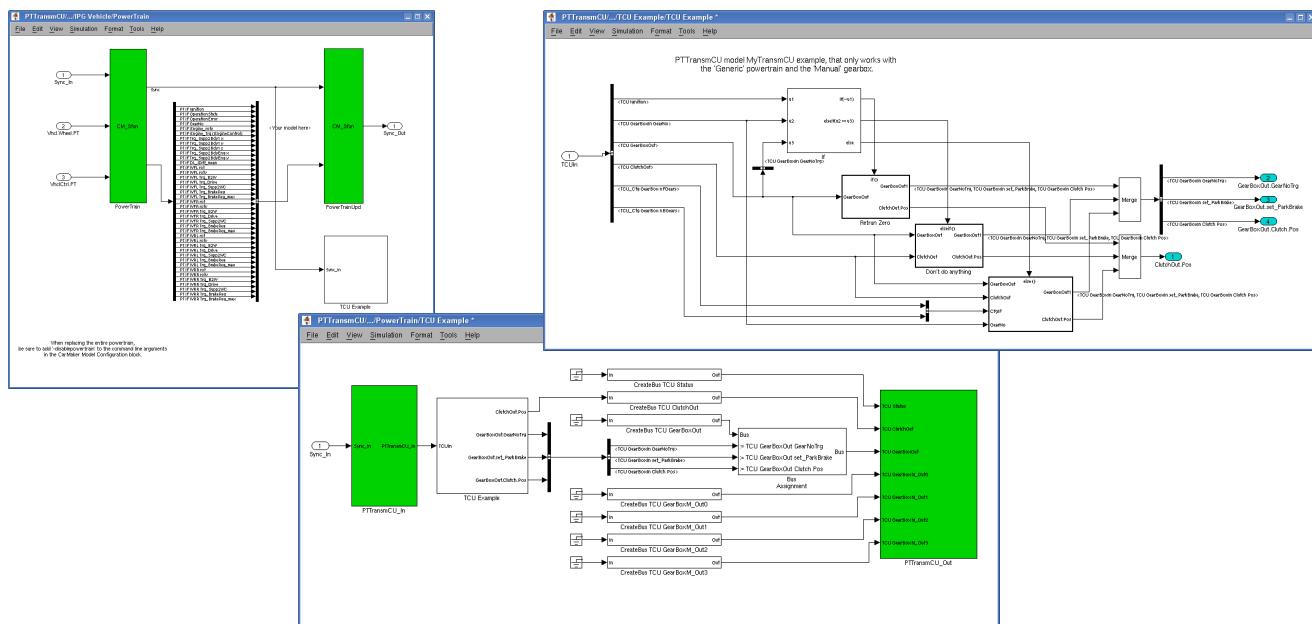


Figure 6.34: PTTransmCU model example

Files belonging to this example

- `src_cm4s1/PTTransmCU.mdl`
The Simulink model.

Details about the example

- This model only works with the generic powertrain and the manual gearbox
- A similar model is also available as an example of the CarMaker target for Simulink Coder (formerly Real-Time Workshop) to show how Simulink models can be integrated into CarMaker using Simulink Coder generated C code.

6.4.10 SoftABS_Hyd

The SoftABS_Hyd model contains a simple implementation of an Anti-Lock Brake System.

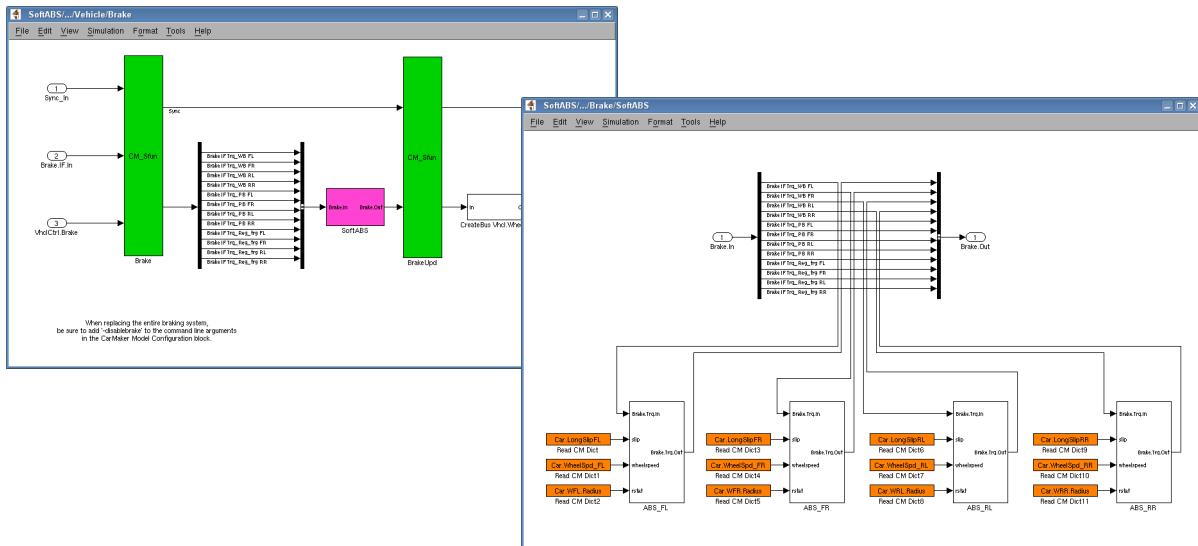


Figure 6.35: SoftABS_Hyd example model

Files belonging to this example

- `src_cm4s1/SoftABS_Hyd.mdl`
`src_cm4s1/SoftABS_Hyd_params.m`
The Simulink model and its parameter file.
- Data/TestRun/Examples/IntegratedControlSystems/ABS_Braking_musplit
- Data/TestRun/Examples/IntegratedControlSystems/ABS_Braking_FrictionPatchwork
The TestRuns to be used with this example.

Details about the example

- For each wheel, the brake pressure is adjusted in order to keep slip to an upper limit.
- Try an example like the
Examples/IntegratedControlSystems/ABS_Braking_FrictionPatchwork
TestRun and compare the results to a simulation with the generic.mdl model.

6.4.11 TractCtrl

The *TractCtrl* example implements some kind of traction control, controlling the braking torques of the front wheels and reducing engine torque if necessary.

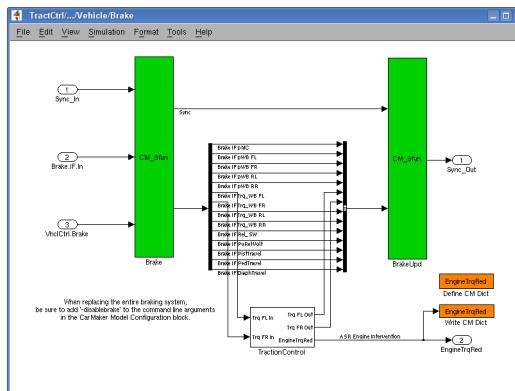


Figure 6.36: TractCtrl example model

Files belonging to this example

- src_cm4s1/TractCtrl.mdl*
The Simulink model.
- Data/TestRun/Examples/Simulink/TractCtrl*
A TestRun to demonstrate the working of the model.

Details about the example

- None; the model is quite simple.
- This model is also available as an example of the CarMaker target for Simulink Coder (formerly Real-Time Workshop) to show how Simulink models can be integrated into CarMaker using Simulink Coder generated C code.

6.4.12 UserBrake

This example shows how to replace CarMaker's internal braking module on the Simulink level.

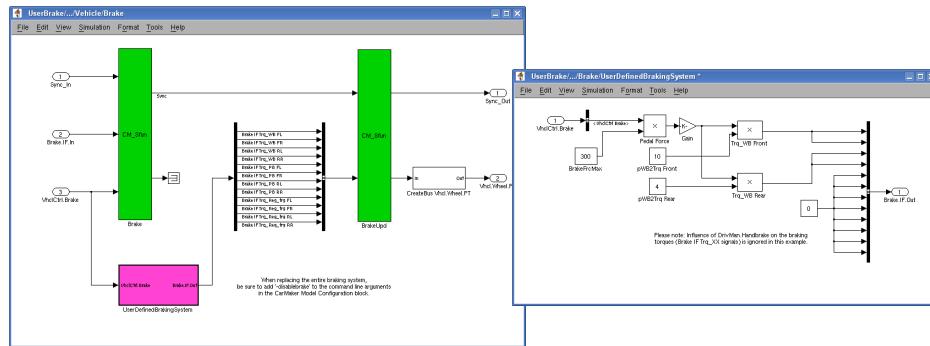


Figure 6.37: UserBrake example model

Files belonging to this example

- src_cm4s1/UserBrake.mdl*
The Simulink model.

Details about the example

- CarMaker's internal braking module is completely disabled, because the `-disablebrake` command line argument is set in the *CarMaker Model Configuration* block (see figure below).

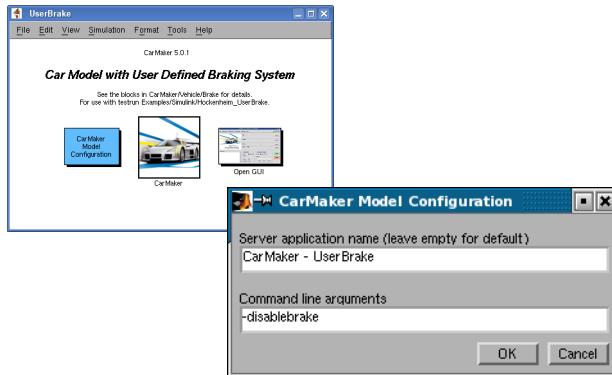


Figure 6.38: UserBrake model configuration

6.4.13 UserPowerTrain

This example shows how to replace CarMaker's internal powertrain module on the Simulink level. It does not contain a complete implementation of a powertrain, but just tries to show the functioning of the powertrain interface.

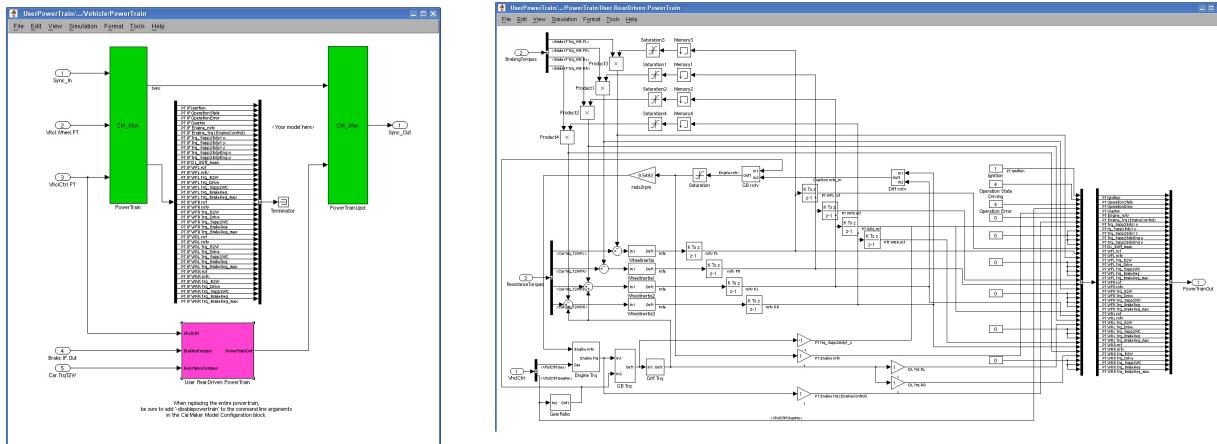


Figure 6.39: UserPowerTrain example model

Files belonging to this example

- `src_cm4sl/UserPowerTrain.mdl`
The Simulink model.
- `Data/Vehicle/Examples/DemoCar_UserPT`
A vehicle with a powertrain configuration as described below.
- `Data/TestRun/Examples/Simulink/UserPowerTrain`
A TestRun to demonstrate the working of the model.

Details about the example

- The model is very limited and only useful in conjunction with the *UserPowerTrain* TestRun. Do not try to use it with other TestRuns.

- CarMaker's internal powertrain module is completely disabled, because the `-disablepowertrain` command line argument is set in the *CarMaker Model Configuration* block (see figure below).
- When simulating with a powertrain implemented in Simulink, you have to make sure your vehicle configuration file (*Data/Vehicle/DemoCar_UserPT* for this example) contains the following entries:
 - `PowerTrain.Kind = “”`
 - `PowerTrain.Engine.on`
 - `PowerTrain.Engine.nidle`
 - `PowerTrain.Engine.rotv`
 - `PowerTrain.GearBox.iForwardGears`
 - `PowerTrain.GearBox.iBackwardGears`
 - `PowerTrain.DL.Diff.i`

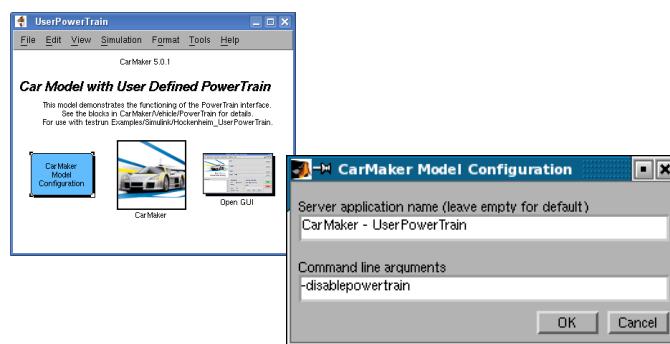


Figure 6.40: UserPowerTrain model configuration

6.4.14 UserSteer

This example shows how to replace CarMaker's internal steering module on the Simulink level.

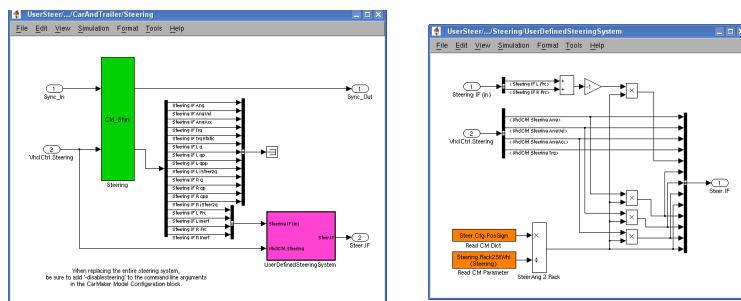


Figure 6.41: UserSteer example model

Files belonging to this example

- `src_cm4s1/UserSteer.mdl`
The Simulink model.

Details about the example

- CarMaker's internal steering module is completely disabled, because the `-disablesteer` command line argument is set in the *CarMaker Model Configuration* block (see figure below).

- A similar model is also available as an example of the CarMaker target for Simulink Coder (formerly Real-Time Workshop) to show how Simulink models can be integrated into CarMaker using Simulink Coder generated C code.
- Even though CarMaker's internal steering model is disabled (by means of the *-disablesteering* command line parameter), some steering model needs to be selected in the vehicle data editor. Any model or parameter set will do, as long as a *Steering.Rack2StWhl* parameter is present and contains a meaningful value. No other steering parameter is required.

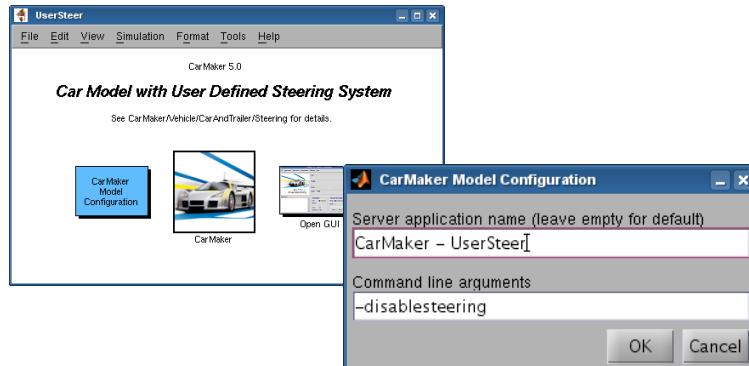


Figure 6.42: UserSteer model configuration

6.4.15 UserSteerTorque

This is another example of how to replace CarMaker's internal steering module on the Simulink level. In contrast to the *UserSteer* example, this model implements steering by torque. It was modeled after the standard CarMaker example C code in *src/ExtraModels/MySteering.c*. See the source code for details.

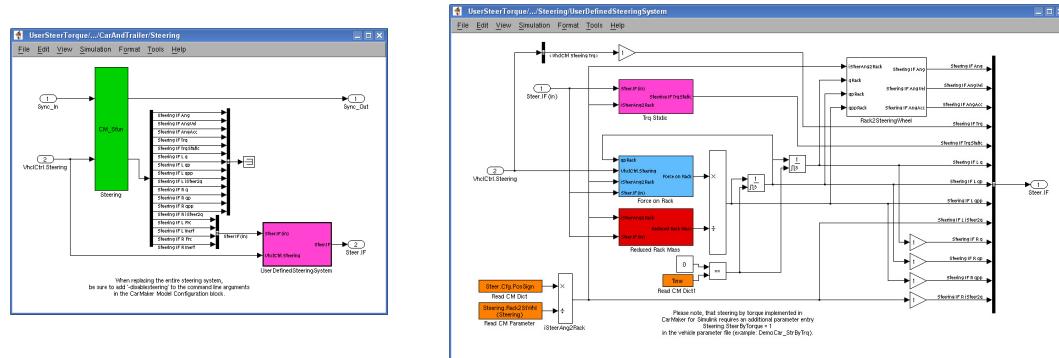


Figure 6.43: UserSteerTorque example model

Files belonging to this example

- src_cm4s1/UserSteerTorque.mdl*
The Simulink model.
- Examples/Simulink/Hockenheim_UserSteerTorque*
A TestRun to demonstrate the working of the model.

Details about the example

- CarMaker's internal steering module is completely disabled, because the *-disablesteering* parameter is set in the *CarMaker Model Configuration* block (see figure below).

- To simulate with this model, make sure to switch on *Steer by Torque* in the *Lateral Dynamics* part of the CarMaker GUI's maneuver dialog. The *Hockenheim_UserSteerTorque* TestRun is just a copy of the *Road/Hockenheim* TestRun with *Steer by Torque* enabled.
- Driver Adaption will not work with this model because the corresponding IPGDriver functionality is not yet suited to steering by torque. Use steering by angle instead.
- Even though CarMaker's internal steering model is disabled (by means of the *-disablesteering* command line parameter), some steering model needs to be selected in the vehicle data editor. Any model or parameter set will do, as long as a *Steering.Rack2StWhl* parameter is present and contains a meaningful value. No other steering parameter is required.

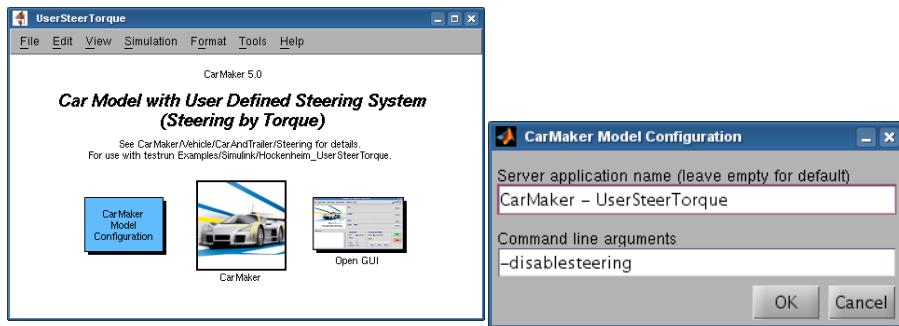


Figure 6.44: UserSteerTorque model configuration

6.4.16 UserTire

This example shows how to replace CarMaker's internal tire module on the Simulink level.

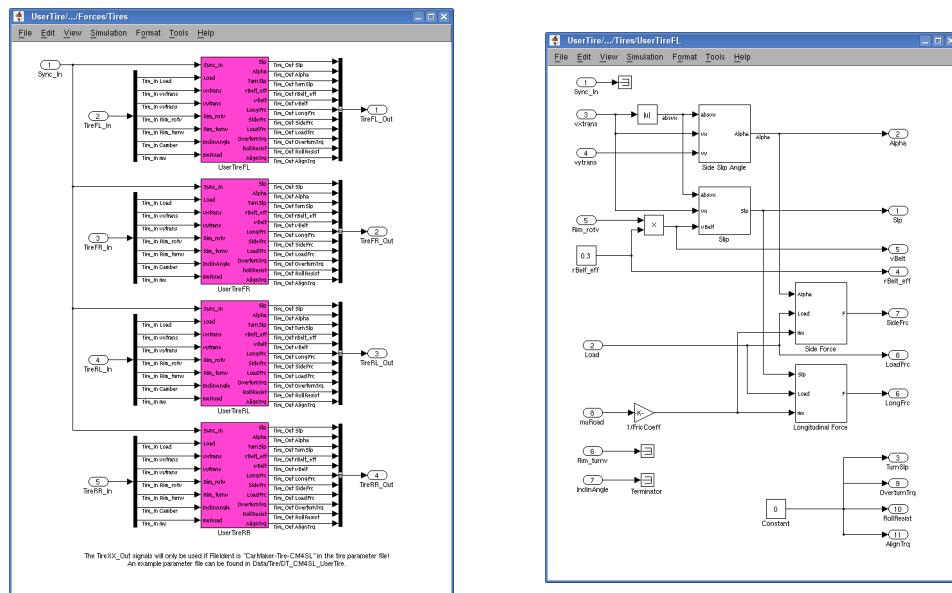


Figure 6.45: UserTire example model

Files belonging to this example

- src_cm4sl/UserTire.mdl*
The Simulink model.
- Examples/Simulink/Hockenheim_UserTire*
A TestRun to demonstrate the working of the model. It is a copy of the *Road/Hockenheim* TestRun with the appropriate tire model selected and a non-zero initial speed.
- Examples/DT_CM4SL_UserTire*
A tire data file used in the example TestRun.

Details about the example

- The *TireXX_Out* signals will only be used if in the tire parameter file the *FileIdent* entry has the value *CarMaker-Tire-CM4SL*, i.e. in the CarMaker GUI, for the TestRun, select a tire data file with this property.
- When simulating with a tire data file not containing this specific *FileIdent* entry, an internal CarMaker tire model will be used instead.
- Apart from the correct *FileIdent* entry, the tire parameter file must contain correct parameters for the tire. CarMaker needs these values for proper initialization.
- The example is not very well suited for stand still conditions.

6.4.17 Generic_UserVehicle

This example shows how to replace the whole CarMaker vehicle model.

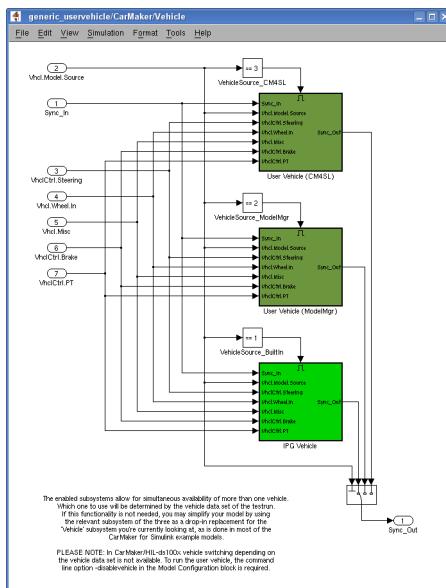


Figure 6.46: UserVehicle example model

Files belonging to this example

- `src_cm4sl/generic_uservehicle.mdl`
The Simulink model.

Details about the example

- The replacement of the CarMaker vehicle model can be activated by the vehicle data set used in the TestRun (see SingleTrack example).
- The Simulink model includes both vehicle version which enables the user to switch between the own vehicle model and the user vehicle model without changing the Simulink model.
- This example should serve as a template to integrate user models.
- See also section '[Replacing the Vehicle Model](#)' on page 770.

6.4.18 SingleTrack

This example uses the generic_uservehicle template to integrate a simple single track vehicle model which replaces the whole CarMaker vehicle model.

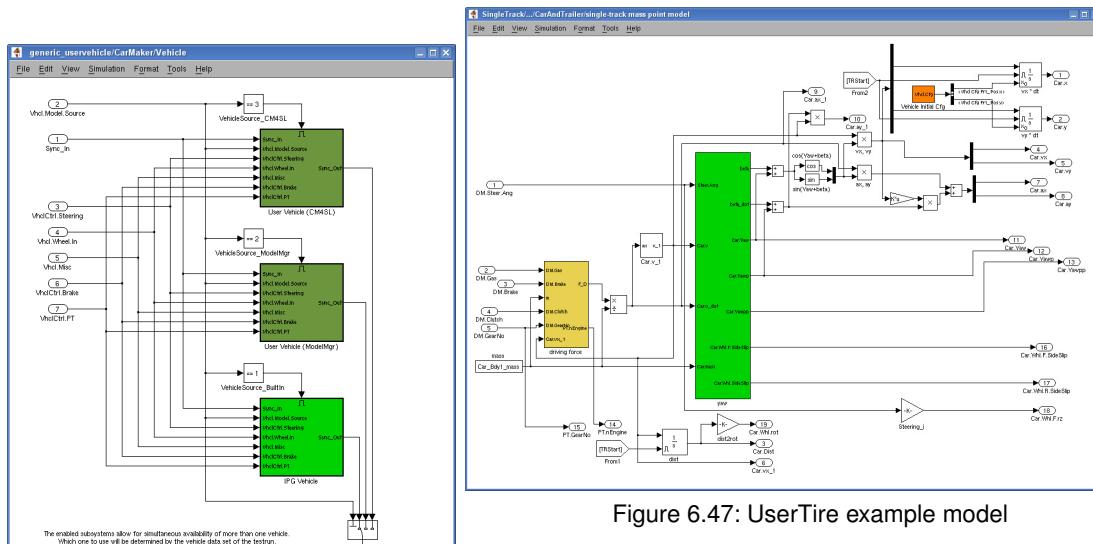


Figure 6.47: UserTire example model

Files belonging to this example

- src_cm4sl/SingleTrack.mdl*
- src_cm4sl/SingleTrack_params.m*
- The Simulink model and its parameter file.
- Examples/Demo_CM4SL_SingleTrack*
- A vehicle data set to activate the user vehicle model

6.5 Troubleshooting

**When I want to run a simulation, I get the following error message:
Minimum step size (...) is larger than the fastest discrete sampling
period (0.001) time.**

Simulink will not allow you to set a solver step size greater than 0.001 seconds in models containing CarMaker for Simulink blocks. The reason for this is the internal configuration of the CarMaker for Simulink blocks, which use a fixed sample time of one millisecond.

**After trying to add or remove blocks of the active CarMaker model
while the simulation is running, suddenly the Start/Stop buttons of the
CarMaker GUI do not work anymore.**

This behavior has been observed under MS Windows. After closing the error message window telling you "Cannot change the model ... while the simulation is running", there seems to be some disorder in Matlab's event processing. You may stop the simulation using **Simulation / Stop** from the model's window, however. After the simulation has been stopped, the buttons of the CarMaker GUI will work again as expected.

**S-functions using ssGet/SetUserData(ssGetRootSS(SimStruct *S)) do
not work or cause Matlab to crash.**

During a simulation of a Simulink model CarMaker for Simulink relies on having complete control over the user data pointer associated with the model. Adding an S-function to a CarMaker for Simulink model, that makes use of the pointer itself, would seriously confuse CarMaker for Simulink, resulting in Matlab crashes and possible loss of data. This situation is not likely to change in the future.

When I start CarMaker for Simulink, I see the following message:

```
Warning: CarMaker command engine task could not be started
CarMaker will be fully functional, but can only start
a simulation in Simulink. The CarMaker GUI's start/stop
buttons, Model Check, Driver Adaption, ScriptControl
etc. won't work
```

This message is most often seen when Java is not properly enabled in Matlab.

Chapter 7

CarMaker utilities for Matlab

7.1 Importing simulation results with `cmread`

With the `cmread` utility, that comes with the Matlab support package for CarMaker, it is possible to load CarMaker simulation results files ("erg files", named after their file extension `.erg`) into the Matlab workspace. Once available in the workspace, the simulation data can be used and manipulated in every conceivable way, e.g. for post processing and plotting.

Invoking `cmread`

Basically `cmread` works as follows:

- `cmread` let you specify a path, either to a directory or to a file. In case a directory is specified, you may select a file with the mouse in a file browser window. By default the current directory (in CarMaker for Simulink the current SimOutput directory) is implied.
- The return value of `cmread` must be assigned to a workspace variable. The result is a Matlab struct with one member for each results file variable loaded. This way, several results files may be loaded into the workspace (e.g. for comparison), the data of each file stored in a different workspace variable.
- By specifying one or more patterns it is possible to select only a subset of all variables contained in a results file. By default all variables are loaded into the workspace.

The next section contains some examples that should make clear how `cmread` works. For further details see the `cmread`'s online help page in Matlab, by typing

```
>> help cmread
```

at the Matlab prompt.

`cmread` example use

Loading a results file "Open a file browser window, let me select the results file with the mouse, and assign the data to the workspace variable `a`:

```
>> a = cmread;
```

“Open a file browser window with the contents of my CarMaker for Simulink simulation results directory, let me select the results file with the mouse, and assign the data to the workspace variable *a*”:

```
>> a = cmread('..../SimOutput/myhost');
```

You may have to replace the name ‘myhost’ with the hostname of the computer you are using.

“Load two different simulation results file into the workspace”:

```
>> a = cmread('SpecialManoeuver1.erg');
>> b = cmread('SpecialManoeuver2.erg');
```

Working with the loaded data

“Display the contents of variable *a*, which contains the loaded data”:

```
>> disp(a)
    Car_CFL_rx:[1x1 struct]
    Car_CFL_ry:[1x1 struct]
    Car_CFL_rz:[1x1 struct]
    ...
    Time      [1x1 struct]
```

Please note the difference between the names of the variables in the results file and the names of the struct members. *cmread* automatically renames variables that do not match the syntax of Matlab’s data types. This includes changing ‘.’ to ‘_’ and truncating long names if necessary.

“Display the contents of the *Car.vx* variable”:

```
>> disp(a.Car_vx)
    unit: 'm/s'
    nstates: 0
    data: [1x10144 double]
```

“Show a speed-over-time diagram, plot the course of the car”:

```
>> plot(a.Time.data, a.Car_vx.data)
>> plot(a.Car_Frl_tx.data, a.Car_Frl_ty.data)
```

Loading only a subset of the variables from a results file

“Open a file browser window showing the current directory, let me select the results file with the mouse, and assign the data of *Time*, all *DrivMan* and all *PT* variables to the workspace variable *a*”:

```
>> a = cmread('.', 'Time', 'DM*', 'PT*');
```

Function arguments following the specified directory ‘.’ may be variable names as well as patterns of names. *cmread* may be invoked with any number of name/pattern arguments. Patterns follow the syntax that normally can be used for filenames. Listed below are the special characters most often used for patterns.

- * matches any string including the empty string
- ? matches any single character
- [...] matches any of the characters enclosed between the brackets

7.2 Accessing CarMaker Infofiles

Infofile access, i.e. access to CarMaker parameter files, can be accomplished using CarMaker's *ifile* command for Matlab.

For a quick start see the *ifile* online help page, by typing

```
>> help ifile
```

at the Matlab prompt.

7.3 Sending Tcl Commands to the CarMaker GUI

Note: This feature is only available in CarMaker for Simulink.

Since Matlab also provides a complete programming environment including an elaborated scripting language, sometimes it might be useful to perform some action inside the CarMaker GUI. For this purpose *cmguicmd* command is available in CarMaker for Simulink. Any Tcl statement including all ScriptControl commands may be executed.

For a quick start see the *cmguicmd* online help page, by typing

```
>> help cmguicmd
```

at the Matlab prompt. Please find further information in [section 'Remote GUI Control' on page 762](#).

Chapter 8

Simulink Coder Interface

Besides the possibility to run CarMaker integrated in Simulink as "CarMaker for Simulink" you have the possibility to use Matlab's Simulink Coder (formerly Real-Time Workshop) to translate Simulink models into C-code and add it to CarMaker using its C-code interface. CarMaker offers Simulink Plug-ins for this purpose that allow you to integrate your Simulink model into CarMaker without caring about the Simulink Coder settings or manually programming in C-code. [section 8.3 'Integration of Simulink models with CarMaker's Model Plug-in'](#) and [section 8.4 'Example: Integrating a user defined model using the CarMaker Model Plug-in'](#) describe how to use these plug-ins. The other sections of this chapter provide some background information and advanced user functionalities.

There are several advantages of using this functionality. First of all, models that are integrated into CarMaker via the Simulink Coder can be used independent from Matlab/Simulink. This means, you can simulate the models on a computer on which Matlab is not installed and the simulation performance is better than with the cosimulation that is done in CarMaker for Simulink. Furthermore all dependencies on Matlab versions are eliminated when using this approach. Thus you can mix Simulink models coming from different Matlab versions together in one CarMaker executable. Additionally it is possible to provide these models e.g. to a supplier in a way that he can use them for simulation but not look inside. Finally, this kind of model integration works with all versions of CarMaker - no matter if it is CarMaker/Office standalone, CarMaker/HIL for XENO, CarMaker/HIL for Labcar or CarMaker/HIL for dSPACE. You even can combine it with CarMaker for Simulink and even for example integrate a brake model using the Simulink Coder Interface while at the same time integrating a powertrain model in CarMaker for Simulink.

Please find further information on this topic in the following pages.

8.1 Starting Matlab

Matlab and CarMaker work together by extending Matlab's search path, so that Matlab knows where to find the CarMaker blocksets and utility commands. The Matlab search path is extended by execution of a small script named `cmenv.m`, which is contained in the `src` or `src_cm4sl` subdirectory of a CarMaker project directory. Execution of this script may

be done manually, but there is also a way to invoke it automatically each time Matlab is started. Please find further information on the starting procedure of CarMaker for Simulink in [section 6.1 'CarMaker for Simulink basics' on page 93](#).

Please ensure that the Matlab working directory corresponds to the folder `src` of your Car-Maker project, if the model has to be integrated into CarMaker/Office standalone or Car-Maker/HIL for XENO. If the model needs to be used in combination with CarMaker for Simulink or CarMaker/HIL for dSPACE, please switch the Matlab working directory to the folder `src_cm4s1` of your CarMaker project.

In the following sections of this chapter we assume that you work in the `src` folder of your project directory.

8.2 The CarMaker target for Simulink Coder

In Simulink Coder (formerly Real-Time Workshop), using a special CarMaker code generation target, C-code can be generated from a Simulink model. The resulting C-code is especially tailored to fit into the CarMaker architecture.

Compiling the generated C-code and linking with the CarMaker libraries leads to an executable CarMaker simulation program. Inside the program, either standalone or under realtime conditions, the Simulink model runs as an integral part of the CarMaker simulation. Multiple Simulink models may be integrated in the same CarMaker executable, each one performing a different computational task.

Please note, that the user model needs to be isolated to use the Simulink Coder interface for C-code generation. The S-functions from CarMaker for Simulink subsystem chain must not be part of the model to compile. However, all other blocksets from the CarMaker for Simulink library may be used.

The CarMaker target is based on the *Generic Real-Time Target (grt_malloc)* coming with Simulink Coder.

Components involved when integrating a Simulink model into CarMaker:

- The generated model C-code
- A wrapper module, providing the model interface to CarMaker
- The run-time library `libMatSupp.a`

8.2.1 Code generation with the CarMaker target

The wrapper is the link between CarMaker and the C-code generated by Simulink Coder. The wrapper shields CarMaker from the details of the generated C-code, providing a consistent API (application programming interface) rather independent of the Matlab version used. In order to integrate a Simulink model, CarMaker will interface the model only by means of the model API provided by the wrapper.

The wrapper files

```
XXX_CarMaker_rtw/XXX_wrap.c  
XXX_CarMaker_rtw/XXX_wrap.h
```

will only be created once, during the first time you ever generate C-code of your model. Since the wrapper might require further customization, subsequent invocations of Simulink Coder will refrain from overwriting existing wrapper files. You may force recreation of the wrapper files by removing them before the next code generation.

The functional interface of a module wrapper consists of only a single function

```
XXX_Register()
```

which should be called once from the *User_Register()* function in file *src/User.c*. In this function you can also change the name under which your model is registered in CarMaker and which must be specified in the vehicle parameter file accordingly. The default name is the name of the Simulink model, but you may use any ordinary C string like e.g. "WhizBrake-4.2" instead.

The *BodyCtrl* and *UserSteer* examples are both integrated into CarMaker using a module wrapper. See [chapter 8.9, "Demonstration examples"](#) for details about this.

Using the *Plain* wrapper

In case a Simulink model does not implement one of the CarMaker's model manager modules, the general purpose *Plain* wrapper should be used. Since CarMaker's model manager will not be involved, a model instance pointer must be managed and the functional interface of the wrapper is slightly more complex:

```
XXX_Inst
XXX_DeclQuants()
XXX_New()
XXX_Delete()
XXX_Calc()
```

The typical invocation of these functions can be found in file *src/User.c*, in the SIMUMODEL integration code as used in the step-by-step example in [section 8.6 'Example: Integration of a Simulink model using the Plain wrapper' on page 173](#). Simply look for occurrences of the string *SIMULINK_EXAMPLE* in the source code of *src/User.c*.

See [chapter 8.9, "Demonstration examples"](#) and the source code in *Templates/RTW* in the CarMaker installation directory.

8.2.2 Choosing the right model wrapper

In most cases a Simulink model is used to implement a module managed by CarMaker's model manager. For this purpose a number of predefined module wrapper templates are available in the CarMaker target for Simulink Coder, among them are

Wrapper class	Subsystem Type
Aero	Aerodynamics
AirBrakeCtrl	Pneumatic brake control (TruckMaker only)
AirBrakeSys	Pneumatic brake system (TruckMaker only)
Brake	Brake
Environment	Ambient conditions
HydBrakeCtrl	Hydraulic brake control
HydBrakeSys	Hydraulic brake system
PowerTrain	Powertrain
PowerTrainXWD	Powertrain, free torque distribution
PTBattery	Battery
PTBatteryCU	Battery Control Unit
PTClutch	Clutch
PTControl	Powertrain Control Unit
PTControlOSM	Operation State Machine
PTDriveLine	Driveline

Wrapper class	Subsystem Type
PTDriveLineXWD	Driveline, OpenXWD
PTEngine	Engine Torque
PTEngineCU	Engine Control Unit
PTGearBox	GearBox
PTGenCoupling	Differential Locks
PTMotor	Motor
PTMotorCU	Motor Control Unit
PTPowerSupply	Power Supply
PTTransmCU	Transmission Control Unit
PowerTrain	PowerTrain
PowerTrainXWD	PowerTrain with XWD driveline
Steering	Steering
SuspExtFrcs	Active Suspension Systems
SuspEF_Buffer	Active Buffer
SuspEF_Damper	Active Damper
SuspEF_Spring	Active Spring
SuspEF_Stabi	Active Stabilizer
SuspKnC	Kinematics and Compliance
TireCPI	Tire model based on single a contact point
TireCPMod	Modification of tire contact point
TireSTI	Tire model based on standard tire interface
User Driver	Driver
VehicleControl	Vehicle Control
Vehicle_Car	Whole vehicle model (incl. tire)
VhclOperator	Vehicle Operator
Generic	Generic Plugin Model, if no other subsystem matches, no special treatment needed, see section 'Generic Plug-in Models'
Plain	Plain template if no other subsystem matches, special treatment, see 8.6, "Example: Integration of a Simulink model using the Plain wrapper"

Typically such a module is selected by an appropriate entry in the vehicle parameter set, e.g. *Steering.Kind* in case of a steering model. During a simulation the model will be invoked automatically by CarMaker's Model Manager.

Using these plug-ins, you do not need to modify any C-code manually.

8.3 Integration of Simulink models with CarMaker's Model Plug-in

The recommended approach to integrate a user model is to use the block *Create CarMaker Plug-in Model* from the CarMaker for Simulink blockset library. It provides model templates for all wrapper classes to integrate user models. Using this block, the user model can be integrated smoothly without any changes to the Simulink Coder settings, the wrapper files or CarMaker C-code files. This means, the Simulink Coder *Build* can be executed directly and the new CarMaker executable is created automatically. The user model is registered and selectable in the CarMaker GUI without any intermediate steps.

On top level of the *Create CarMaker Plug-in Model* dialog the architecture need to be defined (32 or 64bit). The second dropdown-menu is for the vehicle type. This will change the in- and outputs accourdingly. Next the wrapper class needs to be defined, as well as the model name and a model description. The model name is the name under which the model is registered in CarMaker. The model description is used in the CarMaker GUI after compilation. If no model description is stated the model name will be used.

Optionally, the sample rate of the user model can be specified which does not have to be the same as in the CarMaker model. A value of -1 applies the CarMaker default sample rate for this model kind.

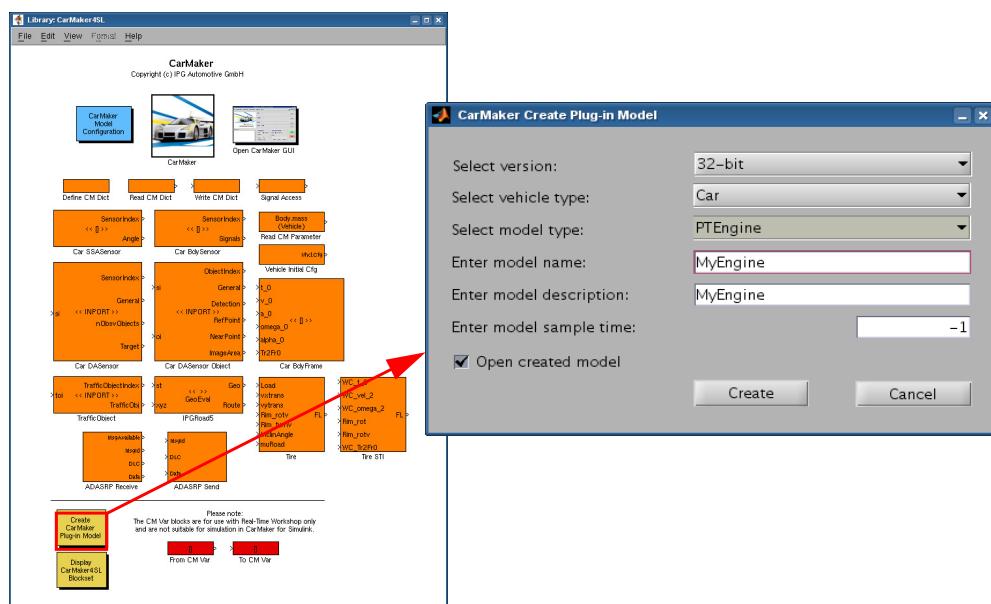


Figure 8.1: CarMaker Plug-in Model dialog parameters

For each model class, the interface structure on top level is similar. The model's input and output signals should remain unchanged, as the created wrap.c file already connects the signal names with the corresponding CarMaker C-code variables.

The model has up to three input buses with connected bus selector blocks. The *IF_In_Selector* block is present in every model and lets you select the input variables of the evaluation interface struct of the model class. The *CfgIF_In_Selector* and *CfgIF_Out_Selector* are only present if the initialisation interface struct of the model class contains any inputs and/or outputs. The *IF_Out_Selector* block lets you select the output variables of the evaluation interface struct of the model class. To select the quantities, double-click the selector block.

All bus selector block have already their first signal selected and terminated.

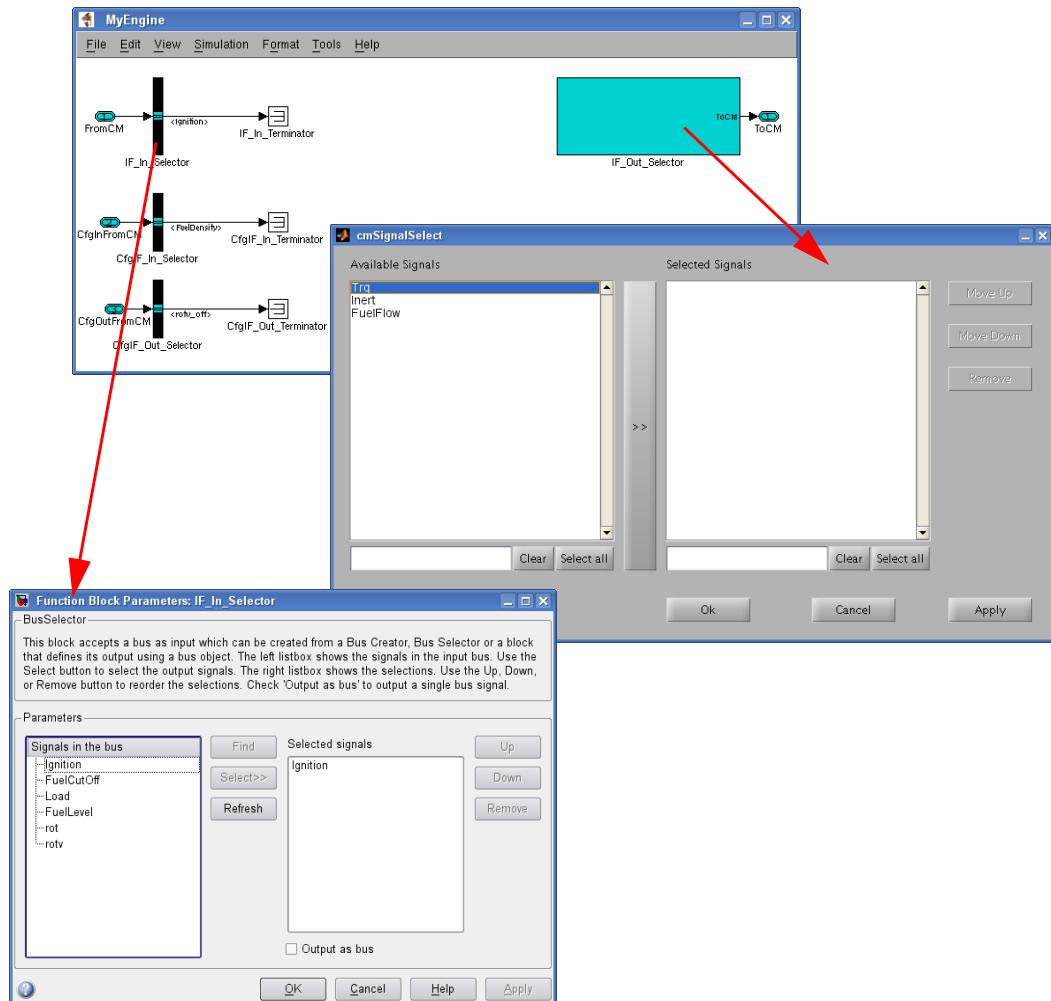


Figure 8.2: Layout of the engine model interface

Please do not change the name of any of the signal buses. For an overview of the in- and output signals of each module see [section 6.3 'CarMaker Subsystem' on page 122](#).

To compile the model it is sufficient to trigger the Simulink Coder *Build* - all Simulink Coder settings are already selected properly. The created C-code model is automatically registered in the CarMaker world and a new executable is created, all depending on the settings of the *Makefile* located in the "CarMaker Project source directory".

Please note: For 64 bit models, it is necessary to change the include path for the *Makedefs* file right at the beginning of the *Makefile* to *MakeDefs64*.



Make sure the new generated file is used for the simulation, for details see [chapter 1.3.3, "Using the new Executable"](#). The CarMaker GUI was extended automatically, so that the user model is available in the model kind selection menu of the vehicle subsystem corresponding to the chosen wrapper class. Please consider the special conditions for the model classes Vehicle ([section 'Replacing the Vehicle Model'](#)), Tire and TireCPMod ([section 5.1.2 'Exceptions and special cases'](#)).

Please note, that the *Plain* wrapper class is treated differently. The *Plain* wrapper gives the user the possibility to integrate a model independent of the pre-defined interfaces. However, the code generation process requires some manual changes to hook this very generic model as described in [section 8.6 'Example: Integration of a Simulink model using the Plain wrapper' on page 173](#).

Example: Integrating a user defined model using the CarMaker Model Plug-in

8.4 Example: Integrating a user defined model using the CarMaker Model Plug-in

As an example we will have a closer look at the *User_Brake* model. As described in [section 8.1 'Starting Matlab' on page 159](#), select the *src* (using CarMaker standalone) or *src_cm4sl* (using CarMaker for Simulink) subdirectory of your CarMaker project directory. Start with a double-click on the *Create CarMaker Plug-in Model* of the CarMaker for Simulink blockset library and select the wrapper class *Brake*. Name the model *MyBrake* and use *UserBrake_RTW Example* as model description.

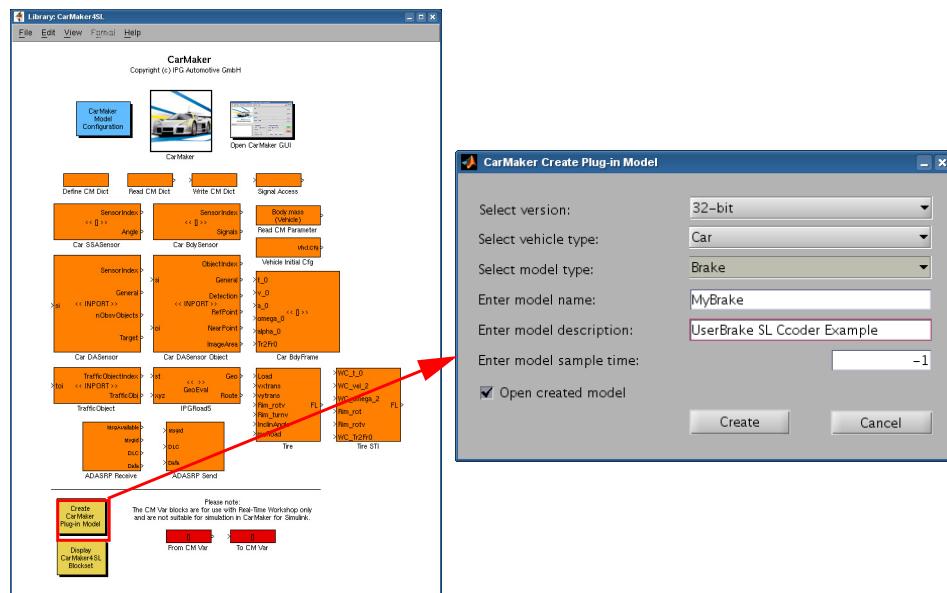


Figure 8.3: *CarMaker Plug-in Model* dialog parameters for a user brake model

Create a subsystem of the new created Simulink model, where we will insert a simple, linear brake model like the following:

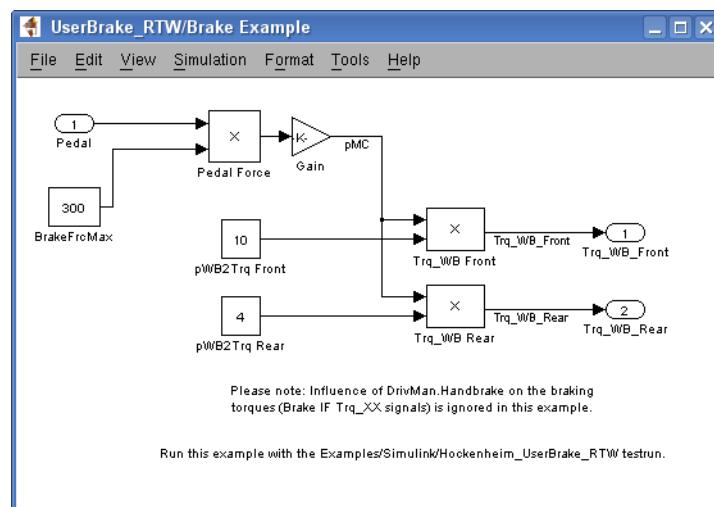


Figure 8.4: User defined brake model to be integrated

Example: Integrating a user defined model using the CarMaker Model Plug-in

The user brake model only outputs the pressure of the brake cylinders and the wheel brake torques on all four wheels. As input, the brake pedal travel is used.

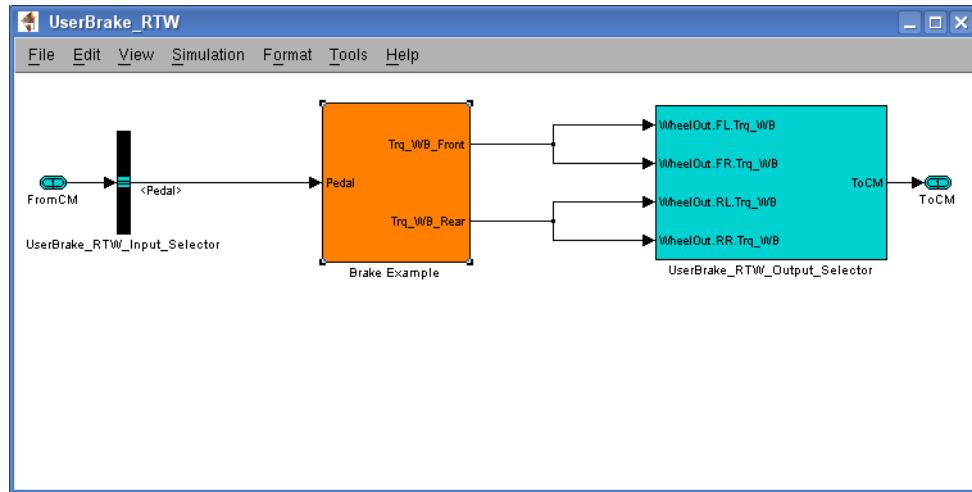


Figure 8.5: Integration of the user defined brake model

Now simply save the model and select *Tools > Code Generation > Build Model* from the Matlab/Simulink main menu. In the Matlab console some output can be found to check the current status of the model compilation process. In Matlab version 2014a and newer, this output can be found after the compilation under *view diagnostics*. Once the compilation was finished successfully, Matlab can be closed.

After opening CarMaker standalone, the newly created CarMaker executable needs to be selected. For this, go to *Application / Start & Connect* in the CarMaker main GUI and select the CarMaker executable just created from the *src* folder of your CarMaker project.

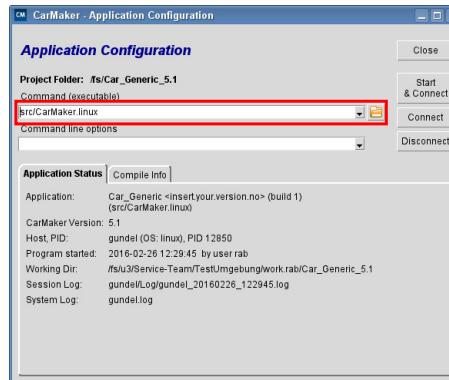


Figure 8.6: Selecting the newly created CarMaker executable

Open a TestRun, e.g. Examples/VehicleDynamics/Braking. To activate the new brake model go to the *Brake* tab in the vehicle data set dialog. Under *Model* select *UserBrake_RTW Example* and everything is ready to simulate!

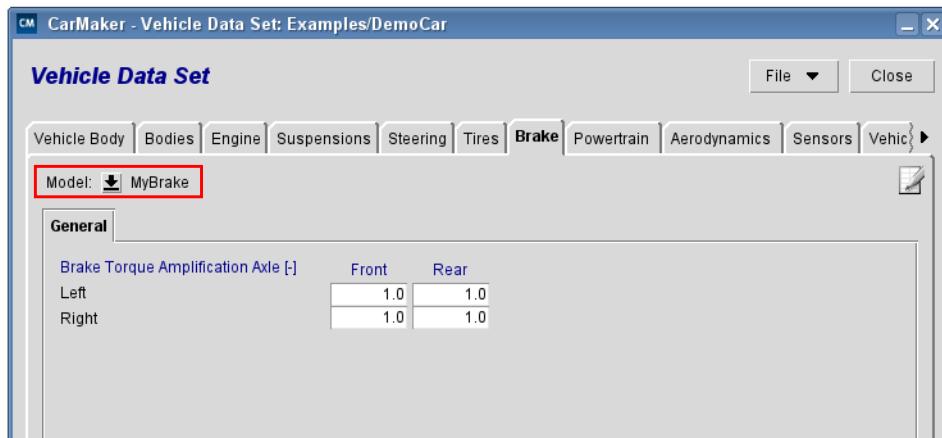


Figure 8.7: Activation of the user defined brake model in the vehicle data set

8.5 Details on the integration of Simulink models

A model integration without the *Create CarMaker Plug-in Model* block requires some customization in the model's simulation parameters, Simulink Coder settings (formerly Real-Time Workshop), wrapper files and tentatively of the CarMaker User.c file. All required settings are explained below.

Prerequisite on your side is a basic familiarity with the CarMaker C level interface and you should know how to rebuild and run the CarMaker simulation program and start a simulation. The Matlab specific settings refer to Matlab 7.10 (R2010a) on a Linux operating system. However, there should be no difficulty in running the example on another platform with a different version of CarMaker or any other Matlab version supported by CarMaker.

8.5.1 Setting simulation parameters

Before the C-code can be generated with Simulink Coder, a few of your model's simulation parameters need to be changed.

Open the dialog **Simulation > Configuration Parameters...** in the menubar of the model window.

Solver settings

On the **Solver** tab, the following changes from the default settings should be made:

- Simulation time
Stop time: *inf*, because CarMaker determines when the simulation stops
- Solver options
Type: *Fixed-step, ode1 (Euler)*, because of the Integrator block in our model
Fixed step size: *0.001*, because CarMaker runs with a step size of 1 ms
Mode: *SingleTasking*, always use SingleTasking with CarMaker

Your solver settings should now look like this:

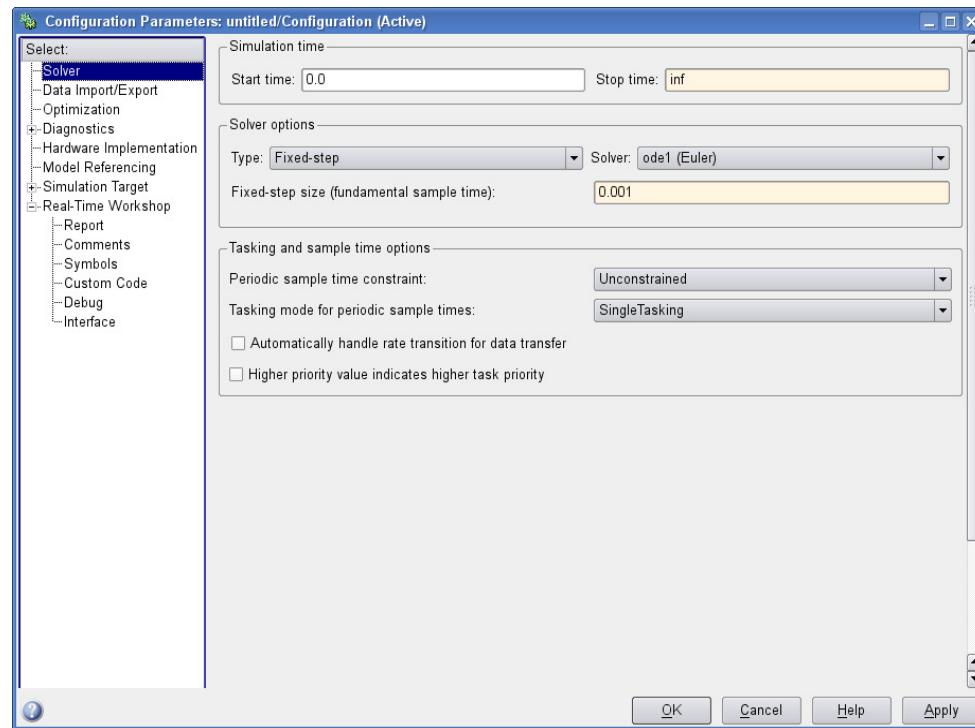


Figure 8.8: Solver settings

Simulink Coder settings

On the **Simulink Coder** (or *Real-Time Workshop*) tab, we first need to set the correct target so that the C-code to be generated fits with CarMaker.

- Category: *Target selection*
- Configuration

System target file: *CarMaker.tcl*, use the **Browse...** button to set this

Generate code only: *deselect*, please note this setting might conflict with the parameter settings in the *CarMaker code generation options* explained below.

The screen dump below shows what your target configuration settings should look like:

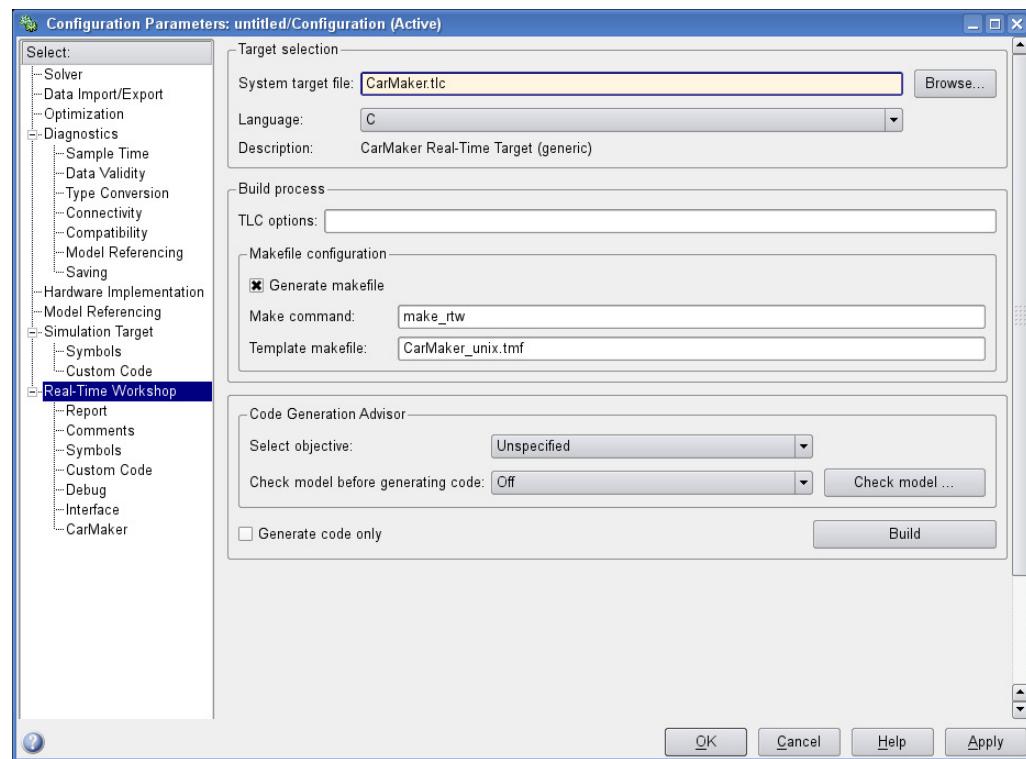


Figure 8.9: Real-Time Workshop, target configuration

The code generation options for the selected CarMaker target can be found in the CarMaker section on the same dialog:



- Category: *CarMaker code generation options*
- Matlab include path: `/fs/opt/matlab712-r2011a`, change this accordingly or leave empty for default. See [section 8.8.2 'CarMaker and Matlab installed on different computers' on page 189](#).
- CarMaker project directory: `../../src`, the default value is fine, change if you like to use other final targets (`src` for CarMaker/standalone or CarMaker/HIL, `src_cm4s1` for Car-Maker for Simulink)
- *Hook up model into project source code*: This option automatically registers your model in the CarMaker User.c. If that option is selected, no changes to the CarMaker C-code environment are necessary. However, the automated model registration does not work for the wrapper type *Plain*.
- *Automatic 'make' in project source directory*: The *make* command is called after the Simulink Coder *Build* is finished. This leads to a new CarMaker executable without manual compilation. Having selected this option, please do not choose *Generate code only* in the Simulink Coder (or Real-Time Workshop) target configuration.



Please note: Without using one of the templates from the *Create CarMaker Plug-in Model* blockset (see [section 8.3 'Integration of Simulink models with CarMaker's Model Plug-in' on page 163](#)), the wrapper files created by the Simulink Coder might need to be customized before compilation to connect the model's in- and outports to the Car-Maker C-code variables. In that case an automated code generation is not recommended.

- *CarMaker model wrapper type*: select the suitable wrapper class for your model

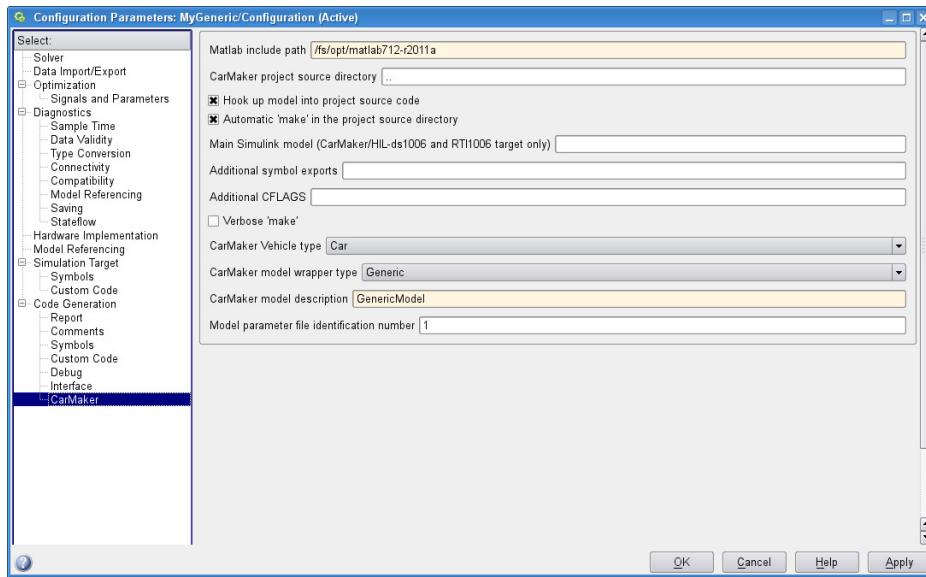


Figure 8.10: Real-Time Workshop, CarMaker code generation options

Simulink Coder settings for CarMaker for dSPACE

To compile plug-in models for CarMaker for dSPACE, additional modifications in the configuration parameters are necessary. These are described as follows:

- Set the *System target file* under *Configuration parameters > Code Generation* as *CarMaker.tlc* (Please note that the target file for the main simulink model is selected as *rti<Board_No>.tlc*.)
- For single processor setup, in the field *Main Simulink Model* (see Figure 8.10), the name of the CarMaker Model (e.g., generic) needs to be entered.
- In case of multi processor setup, it is first suggested to segregate the model blocks of the main Simulink model into different subsystems so as to easily classify them into master and slave units. It needs to be ensured that all the 3 main blocks of a CarMaker model (i.e., model configuration block, CarMaker block and Open GUI block) are classified together as master or slave and they are not separated. Following this, the plug-in model is created with the steps described in the previous sections and under Configuration Parameters of the plug-in model in *Code Generation > CarMaker*, the entry in the field *Main Simulink Model* has to be the name of the application where CarMaker models are classified (i.e., if CarMaker is classified as the master, then the name of the master-application needs to be entered here).
- In case you experience glitches with automatic compilation of the plug-in model, then deselect the option *Automatic 'make' in the project directory* and build the plug-in model and the main model separately.



8.5.2 Customizing the wrapper

After having adapted the simulation settings correctly and having finished the Simulink Coder *Build* a new subdirectory was created in the *src* folder of your CarMaker project. It is called *Modelname_CarMaker_rtw* and contains among others two wrapper files, which link CarMaker with the C-code generated by Simulink Coder:

```
XXX_CarMaker_rtw/XXX_wrap.c
XXX_CarMaker_rtw/XXX_wrap.h
```

Setting the solver

Which Simulink solver is used by a model is determined in the wrapper C-code itself and is completely independent of the solver setting in the model's simulation parameters. The corresponding code block can be found quite at the beginning of a `XXX_wrap.c` file:

```
# define rt_ODECreateIntegrationData rt_ODE1CreateIntegrationData
# define rt_ODEUpdateContinuousStates rt_ODE1UpdateContinuousStates
# define rt_ODEDestroyIntegrationData rt_ODE1DestroyIntegrationData
```

The default is to use the ODE1 solver. In order to use any of the other fixed step solvers supported by Simulink Coder, change all three occurrences of *ODE1*. Possible values are *ODE1*, *ODE2*, *ODE3*, *ODE4* and *ODE5*.

Be sure that all three #define's are set to the same solver; mixing solvers will very likely cause the simulation program to crash.

Connecting Imports and Outports of a Simulink model

Imports and *Outports* of a Simulink model constitute the preferred way of connecting the model to CarMaker. They are accessed in function `XXX_Calc()` of the wrapper:

```
MyModel_Calc (void *MP, tModelIF *IF, double dt)
{
    rtModel_MyModel *rtm = (rtModel_MyModel *)MP;
    ExternalInputs_MyModel *in = (ExternalInputs_MyModel *) rtmGetU(rtm);
    ExternalOutputs_MyModel *out = (ExternalOutputs_MyModel *) rtmGetY(rtm);

    in->ModelInportName = IF->CarMaker_CVariable;
    ..
    DoOneStep(rtm);
    IF->CarMaker_CVariable = out->ModelOutportName;
    ..
    return 0;
}
```

To address the CarMaker C-code variables correctly, search the header files in the include folder of your CarMaker installation directory or check [section 6.3 'CarMaker Subsystem' on page 122](#).



In case, the *From and To CM Var* blocks from the CarMaker for Simulink blockset library ([section 6.2.7 'Accessing C variables' on page 112](#)) were used to access input and output signals from/to CarMaker, no changes to the wrapper file are required.

8.5.3 Integrating the model into CarMaker

Integration of a Simulink model into CarMaker means logical and physical integration.

For logical integration, the model code must actually be called from within CarMaker in order to initialize the model code, call it during each simulation cycle, and perform cleanup actions after the simulation.

Physical integration comprises of adding the model library code to the CarMaker simulation program, i.e. linking the simulation program against the model library when rebuilding CarMaker.

Calling the model from User.c

In case, the option *Hook up model into project source code* was not activated in the CarMaker code generation settings (see [Figure 8.10](#)), the following changes to the User.c file in the *src* folder of your CarMaker project directory need to be done for the logical model integration:

- Including the wrapper header file

Typically, for a Simulink model `src/XXX.mdl`, a line like

```
#include "XXX_CarMaker_rtw/XXX_wrap.h"
```

must be added to file `User.c`, provided that you're actually calling the model from `User.c`.

- Registering the model

In the `User_Register` function, the model can be called by adding a line like

```
Modelclass_Register_MyModel();
```

in the `User.c` under `User_Register()`. See [section 5.2.1 'Registration' on page 69](#) for details.

Linking with the model library

Before the code can be compiled, the `Makefile` in the `src` directory needs to be customized which represents the physical part of the model integration. The `Makefile` contains settings for the compiler. The following statements must be present exactly once in the `Makefile`:

```
DEF_CFLAGS += -DRT -DUSE_RTMODEL  
MATSUPP_MATVER = v6.5.1-r13  
LD_LIBS += $(MATSUPP_LIB)
```

The Matlab version should match the one you are using; please check the Release Notes for Matlab versions supported by your current CarMaker version.

For each Simulink model to be integrated, add a line like the following:

```
OBJ += libXXX_$(ARCH).a
```

The `Makefile` of each CarMaker project directory template does already contain these lines, but they are commented out by default. You simply need to remove the comments and adapt the lines accordingly.

Please note: For 64 bit models, it is necessary to change the include path for the `Makedefs` file right at the beginning of the `Makefile` to `MakeDefs64`.



Compiling the model library

To execute the following commands, use a command shell on Linux systems or the commandline interpreter Msys on Windows systems. Please find further information in [section 1.3.2 'Building the new Executable' on page 21](#).

Typically, for a Simulink model `src/XXX.mdl`, the following command, issued in directory `src/XXX_CarMaker_rtw`, will compile the model library:

```
% cd <path to your project directory>/src/XXX_CarMaker_rtw  
% make -f XXX.mk
```

Compiling a new CarMaker executable

Based on the changes made to the `Makefile`, this new library will be linked when finally compiling the new CarMaker executable. For this, the `make` command needs to be called in the `src` directory of your CarMaker projects:

```
% cd <path to your project directory>/src  
% make  
% make
```

When `make` has finished you will find a new `CarMaker.linux` or `CarMaker.win32` executable file in the `src` directory, which can now be used to test your model running a simulation.

8.5.4 Activating the model

Having finished the compilation by executing the make command, CarMaker for Simulink can be left to start CarMaker standalone. In the main GUI, the newly generated executable needs to be activated (*Application > Configuration/Status > Command (executable)*) to make the integrated user models available.

In the CarMaker dialogs the user model can be made available in the model kind selection menu of the vehicle subsystem corresponding to the chosen wrapper class, by selecting *Add Model*. The model is then ready for simulation.

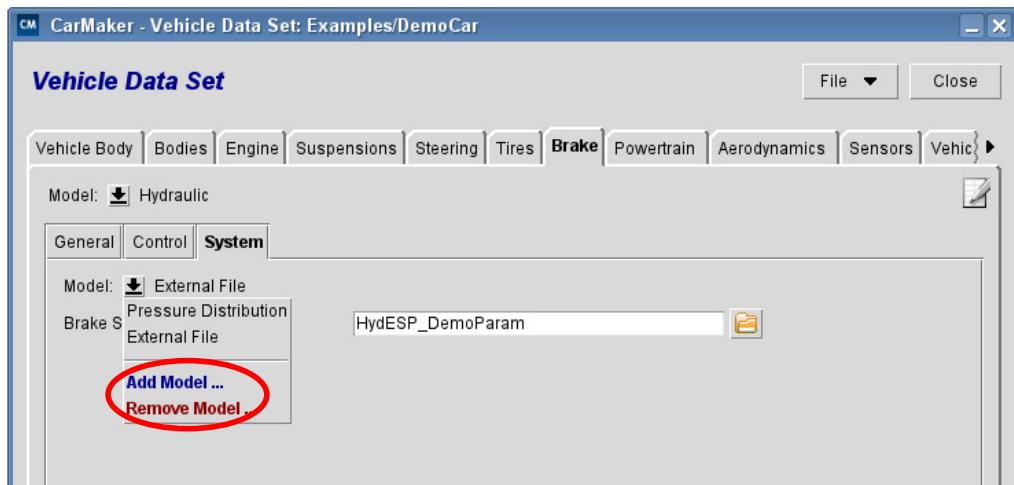


Figure 8.11: Adding a user model to the model kind selecting menu in the GUI

8.6 Example: Integration of a Simulink model using the Plain wrapper

The example was created with the stand-alone version of CarMaker under Linux, using Matlab 7.5 (R2007a). However, there should be no difficulty in running the example on another platform with a different version of CarMaker or any other Matlab version supported by CarMaker.

8.6.1 Creating the Simulink model

Start Matlab in the *src* subdirectory of the project directory.

Create a new Simulink model like the following one and save it under the name *SimuModel.mdl*.

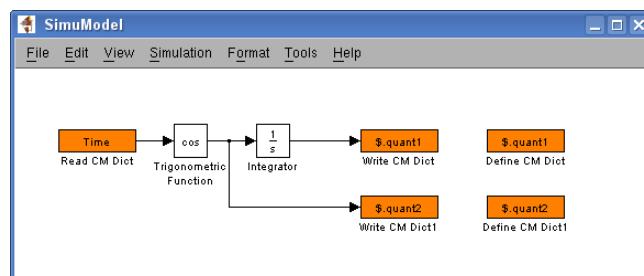


Figure 8.12: The SimuModel.mdl example model

In the Simulink library the orange blocks of the CarMaker Interface blockset can be found under *Blocksets & Toolboxes*, the Integrator block is under *Continuous*, and the Trigonometric Function comes from *Math Operations*.

In the **Block Parameters** dialog of the *Read CM Dict*, *Write CM Dict* and *Define CM Dict* blocks, leave all parameters at their default, just enter the name of the quantity. You will probably wonder about the notation we use for the quantity names: The \$ sign stands for the model name, i.e. *SimuModel* in our case. Using the model name as a prefix, we can keep all model quantities together, which makes them easier to identify among all the other quantities in CarMaker's data dictionary. Also, once we decide to give our model a different name, after rebuilding the code all of the model's quantity names will have changed automatically.

The model does not do something particularly useful. It uses the current simulation time as its input and provides its sine and cosine as outputs. We just want the model to be called by CarMaker once each simulation cycle, so that we can observe its changing output signals.

8.6.2 Setting simulation parameters

Before the C-code can be generated with Simulink Coder (formerly Real-Time Workshop), a few of your model's simulation parameters need to be changed. Please follow the instructions in [section 8.5.1 'Setting simulation parameters' on page 167](#).

As wrapper type, use the *Plain* wrapper. Activate just the options *Hook up model into project source code* and not *Automatic 'make' in project source directory*. The *make* has to be executed later on manually.

8.6.3 Generating and compiling the model C-code

Generating the C-code

In the menu bar of your model window invoke **Tools > Real-Time Workshop > Build Model**.

Simulink Coder (formerly Real-Time Workshop) will now automatically make a new subdirectory *SimuModel_CarMaker_rtw* and create the model C-code files inside that directory. After code generation is complete you will find the following files there:

```
SimuModel.c  
SimuModel.h  
SimuModel.mk  
SimuModel_private.h  
SimuModel_types.h  
SimuModel_wrap.c  
SimuModel_wrap.h  
modelsources.txt  
rtmodel.h  
rtw_proj.tmw
```

(Number and names of the files actually generated may vary depending on the Matlab version you are using for this example.)

Compiling and creating the model library

To compile, use a command shell on Linux systems or the commandline interpreter on Windows systems. Please find further information in [section 1.3.2 'Building the new Executable' on page 21](#). Change into the newly created *SimuModel_CarMaker_rtw* directory and compile the code with the following commands:

Example: Integration of a Simulink model using the Plain wrapper

```
% cd <path to your project directory>/src/SimuModel_CarMaker_rtw
% make -f SimuModel.mk
```

After the `make` command has finished, in the `src` directory above, you will find a new file named `libSimuModel_linux.a`, parallel to your `SimuModel.mdl` file. This file is called the model library and contains the compiled mode C-code that we are now going to integrate into the CarMaker simulation program.

Here is what your `src` directory might look like now:

```
.
|-- SimuModel.mdl
|-- SimuModel_CarMaker_rtw
|   |-- SimuModel.c
|   |-- SimuModel.h
|   |-- SimuModel.mk
|   |-- SimuModel_private.h
|   |-- SimuModel_types.h
|   |-- SimuModel_wrap.c
|   |-- SimuModel_wrap.h
|   |-- modelsources.txt
|   |-- rtmodel.h
|   '-- rtw_proj.tmw
|-- cmenv.m
`-- libSimuModel_linux.a
```

Logical integration: Calling the model from User.c

Edit the file `User.c` with your favorite ASCII editor for programming and look for the following line quite at the beginning:

```
/* #define SIMULINK_EXAMPLE */
```

Remove the comments as follows, leaving no blanks at the beginning or at the end:

```
#define SIMULINK_EXAMPLE
```

That should activate the remaining SIMUMODEL integration code in `User.c`. Your model will now be called by CarMaker during the simulation. We included the integration code in `User.c` to save you some typing when trying to run this example. To find out in detail about all calls to `libSimuModel_linux.a`, simply search the file for all occurrences of the string `SIMUMODEL`. See also the description of the *Plain* wrapper template in this manual.

Physical integration: Adding the model library to the simulation program

The steps required to build a CarMaker simulation program, i.e. compiling and linking it, are controlled by the `Makefile`. Edit `Makefile` with your favorite ASCII editor for programming, locate the following lines and change them accordingly:

```
### Linking with RTW-built Simulink models
OBJS += libSimuModel_$(ARCH).a
### END (Linking with RTW-built Simulink models)
```

Building the simulation program

In the terminal window, still in the `src` directory, enter the following command to build the simulation program:

```
% make
```

When *make* has finished you will find a new *CarMaker.linux* or *CarMaker.win32* executable file in the *src* directory, which can now be used to test your model running a simulation.

8.6.4 Running the model

Now that everything is ready, try the following steps:

- Start the CarMaker GUI.
- Tell it to use the *CarMaker.linux* executable file just created.
- Invoke **Application / Start & Connect**.
- Invoke **File / Open...** and load the *Braking TestRun*.
- Invoke **File / IPGControl**.
- Display *SimuModel.quant1* and *SimuModel.quant2* in a diagram window.
- Start a simulation by pressing the big green **Start** button.

After the simulation, if everything went right, your diagram should look approximately like this one:

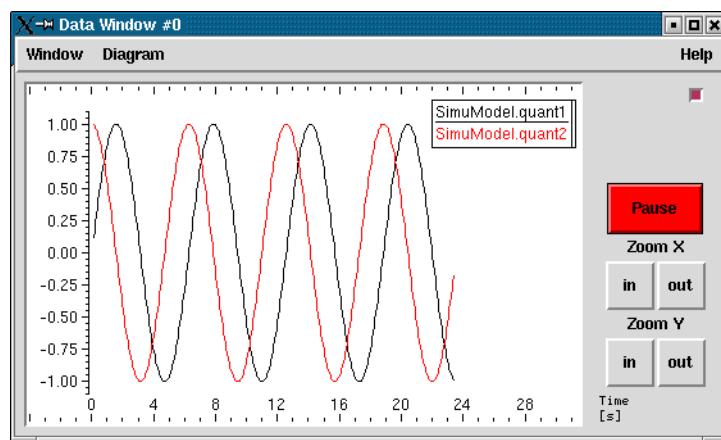


Figure 8.13: SimuModel output in IPGControl

Congratulations, you have just integrated your first *Plain* Simulink model into CarMaker!

8.7 CarMaker's tunable parameter interface

8.7.1 Introduction

Let us take a look at the following, very simple Simulink model.

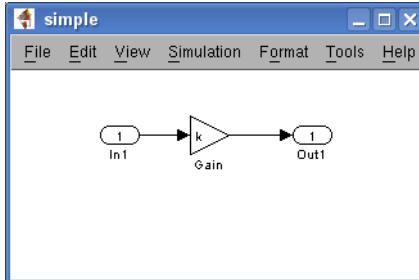


Figure 8.14: Simulink model with a single parameter

The gain factor is not a constant value but a workspace variable named *k*.

```
>> disp(k)
0.7827
```

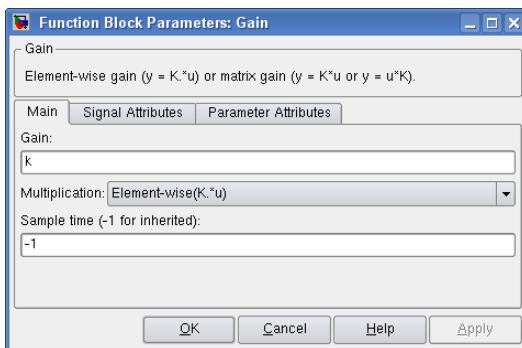


Figure 8.15: Simulink model with a single parameter

We want to generate C-code for this simple model and we want to be able to set a different gain factor at the beginning of each simulation. In other words, we want the gain block's parameter *k* to be a tunable parameter. Being able to change a block parameter's value *after* C-code generation, when the code is running, that's what tunable parameters are all about.

Usually, at the beginning of a simulation, CarMaker reads all information about TestRun, vehicle configuration etc. from external parameter files. These files are named Infofiles after the way the data they contain is formatted. The CarMaker target's tunable parameter interface is mostly about storing Matlab workspace variables in Infofiles, retrieving a tunable parameter's value from an Infofile or setting it directly.

To make use of CarMaker's tunable parameter interface with a Simulink model one normally has to perform the following steps:

- Enabling tunable parameters in the Simulink model.
- Adding statements to the model wrapper which modify the model's tunable parameters.
- Creating infofile entries with the values of all parameters to be tuned (optional).

The remaining sections of this chapter will guide you through the details of parameter tuning, always keeping an eye on the simple example introduced at the beginning.

8.7.2 Enabling tunable parameters in a Simulink model

Let us continue with our Simulink model. The next thing we do is to tell Simulink that the workspace variable k should be treated as a tunable parameter and tell Simulink Coder to include special parameter tuning data in the generated C-code.

In the model window's menu bar, in the **Simulation** menu, choose **Configuration Parameters...** which opens a dialog.

In the listbox on the left click on **Optimization**. On the tab select **Inline parameters** and click on the **Configure...** button.

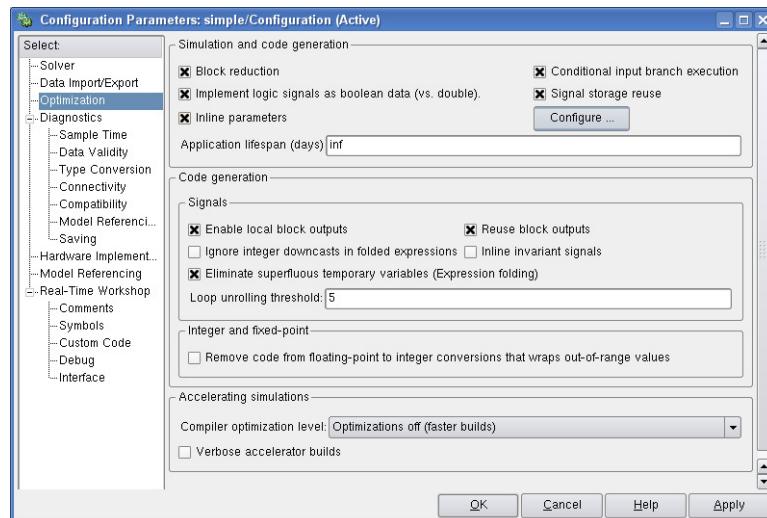


Figure 8.16: Selecting *Inline parameters*

In the **Model Parameter Configuration** dialog, we add k to the list of **Global (tunable) parameters**, like in the screen dump below.

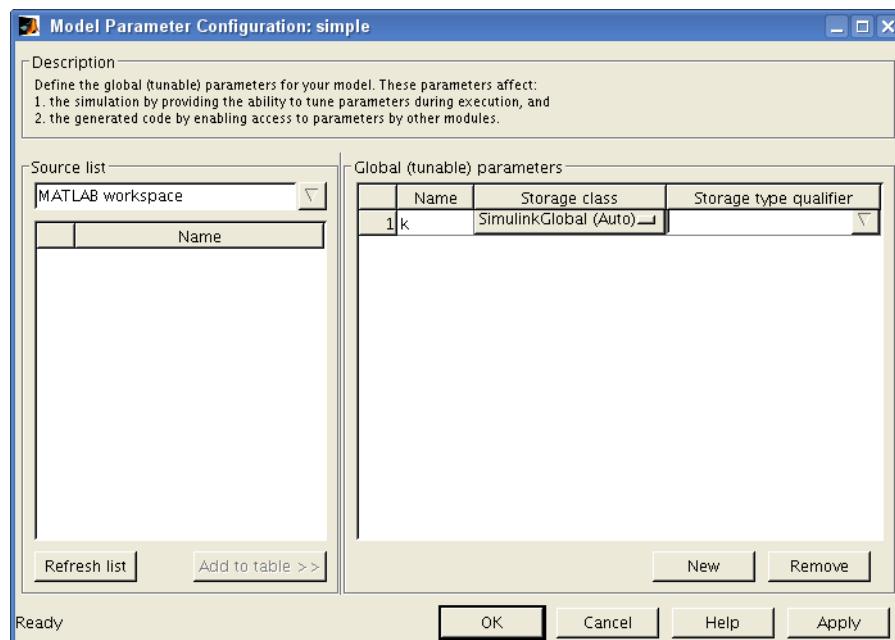


Figure 8.17: Marking parameter k as tunable

Make sure that the storage class of the parameter is *SimulinkGlobal*. The CarMaker target's tunable parameter interface functions apply to parameters of this storage class only.

Next, in the listbox on the left click, under **Simulink Coder** (or *Real-Time Workshop*) click on **Interface**. On the tab, in the **Data exchange** area, select *C-API* as **Interface** and make sure that **Parameters in C API** is enabled.

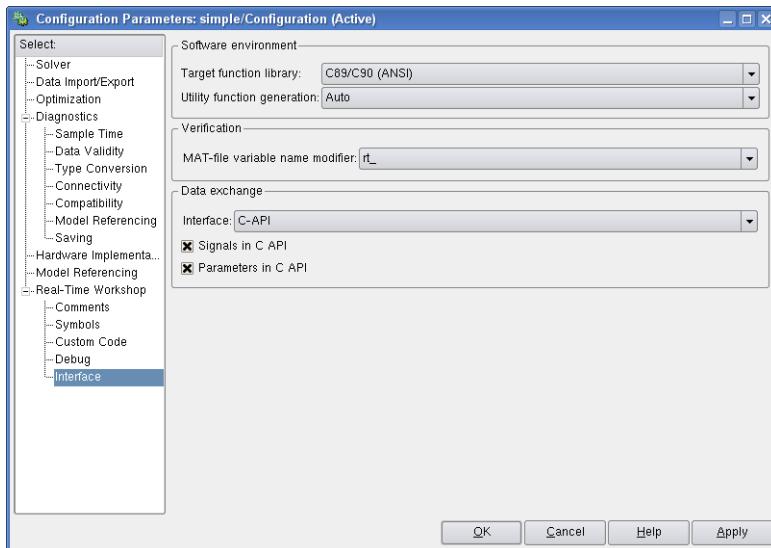


Figure 8.18: Enabling code generation for tunable parameters

8.7.3 Modifying tunable parameters in the model wrapper

After you have generated C-code for your model using the model settings explained in the preceding paragraphs, in principle everything is ready for parameter tuning at runtime. What remains to be done is to write the program statements that assign the tunable parameters of a model their desired values, either coming from an Infofile or being the result of some calculation.

The model wrapper (file `XXX_wrap.c`) generated by Simulink Coder contains a function called `XXX_SetParams()` which will be called automatically everytime the model is initialized at the beginning of a simulation. In a newly created wrapper module the function already calls the “`MatSupp_TunReadAllOpt`” function which reads all values of all tunable parameters from the Infofiles keys with a prefix called like the model name and the workspace values as default. Thus the model is already prepared for the use of tunable parameters. The function will be described in more detail later.

```
void
XXX_SetParams (SimStruct *rts, struct tMatSuppTunables *tuns, struct tInfos *Inf)
{
    /*Log("%s_SetParams()\n", Modelname);*/

    /*
     * Parameter tuning - Part 1
     * This is the place to modify parameters of a storage class
     * other than 'SimulinkGlobal'.
     */

    if (tuns == NULL)
        return; /* No tunable parameters in this model. */

    /* Define dict entries for tunable parameters (address update). */
    DeclParameterQuants(tuns);

    /*
     * Parameter tuning - Part 2
     * This is the place to modify parameters of storage class 'SimulinkGlobal',
     * e.g. using the CarMaker target's tunable parameter interface.
    */
}
```

```

    */
    const char *prefix = Model_Class2Str(Modelclass);
    MatSupp_TunReadAllOpt(tuns, Inf, prefix);
    /*MatSupp_TunReadAll(tuns, ...);*/
    /*MatSupp_TunReadDef(tuns, ...);*/
    /*MatSupp_TunRead(tuns, ...);*/
}

```

As you can see, parameter tuning for parameters which were not assigned the *Simulink-Global* storage class is also possible and has its place in the wrapper. See the *BodyCtrl* example in the *Examples/RTW* directory of your CarMaker installation, to see how tuning of an *Exported Global* parameter could be done.

To stay with our simple Simulink model, the parameter tuning code for our simple model may (after having removed most of the comments) look like this:

```

void
simple_SetParams (SimStruct *rts, struct tMatSuppTunables *tuns, struct tInfos *Inf)
{
    /*Log("%s_SetParams()\n", Modelname);*/

    if (tuns == NULL)
        return; /* No tunable parameters in this model. */

    MatSupp_TunReadAllOpt(tuns, Inf, Modelname);
}

```

Running the model in CarMaker would then proceed as follows: During initialization of a simulation, all tunable parameters will be reset to their original values (i.e. the values they had when the C-code was generated). After that *simple_SetParams()* will be called automatically. Our function tries to find entries for all the tunable parameters of the model in the vehicle parameter file (*lInf* handle). If an infofile entry named *simple.k* is present, *k* will be set to the entry's value. If no such entry is present, *k* will keep its original value (0.7827 in our case). The simulation will then proceed using the actual value of *k*.

A detailed description of the functions of CarMaker's tunable parameter interface can be found in the next chapter. At the end of each function's description you will find one or more small examples which may help you setting up your parameter tuning code in *XXX_SetParams()*.

For accessing Infofile values you may also want to refer to functions of the Infofile library. See the specific chapter in this manual and also the include file of the library, to be found at *include/infoc.h* in your CarMaker installation.

8.7.4 Parameter values in Infofiles

The last question is how to make Infofile entries for the tunable parameters of a model. Normally one wants to take a snapshot of the current values of the Matlab workspace variables in question (probably at the time the model C-code is generated) and create the corresponding entries in an Infofile.

CarMaker's tunable parameter interface provides several utility commands available at the Matlab prompt, which help you in automatically creating infofile entries for the tunable parameters of a Simulink model. Please refer to the online help of the following commands, their use should be rather self explaining:

```

>> help cmtuncreate
>> help cmtunappend
>> help cmtundump

```

These commands make use of the *ifile* suite of commands for infofile access, for which an online help is available, too:

```
>> help ifile
```

Tunable parameters will be stored using the *ifile_setmat* command (uses *InfoSetMat()*) and CarMaker's tunable parameter interface functions will try to read them using *InfoGetMat()* and *InfoGetMatDef()* of the infofile library.

The *tInfoMat* infofile entries created by *InfoSetMat()* are capable of storing most of Matlab's numeric types, including complex and n-dimensional matrices in an infofile entry. For the most common and simple cases like scalars and vectors, however, a *tInfoMat* entry with its structure header may sometimes be "too much". For this reason, compatibility with earlier infofile entry formats like

```
SomeScalar = 3.1415
SomeVector = 10 12 13 14 15 16 17
```

is provided when reading an entry with *InfoGetMat()* / *InfoGetMatDef()*. This means in many cases you may stay with the simpler entries like the ones mentioned, which also has the advantage of being compatible with other CarMaker components with no such strong connection to Matlab.

8.7.5 Adding tunable parameters to the CarMaker dictionary

After initialization a tunable parameter's value remains constant during the whole simulation, so normally there's no point in making it a dictionary quantity so that e.g. it can be monitored in IPGControl. But think of what would be possible if the parameter could be changed on the fly using Direct Variable Access (DVA).

For example, you may want to experiment with a parameter's value during the simulation, observing the immediate change in your model's behavior.

Or suppose you want to be able to activate or deactivate parts of your Simulink model at will. Simply add a Gain block to a signal which multiplies the signal by either 0 or 1, make the block's gain factor tunable and add it to the CarMaker dictionary.

Definition of such a parameter quantity means adding some code to the function *DeclParameterQuants()*, to be found in a Simulink model's wrapper module (*XXX_wrap.c*). Let us add a single scalar parameter *MyParam* to the dictionary:

```
static void
DeclParameterQuants (struct tMatSuppTunables *tuns)
{
    MatSupp_TunDDictDefScalar(tuns, "MyParam", INFOMAT_DOUBLE, "kappa", "kg/s");
}
```

The quantity will be called *kappa* and its unit will be *kg/s*. Giving the resulting dictionary quantity a name different from that of the parameter allows for e.g addition of some kind of name prefix like the model's name etc.

Please note that the *DeclParameterQuants()* function will be called twice, first with *tuns* equal to **NULL**. The reasons for the two invocations lies in the internal sequence of events during CarMaker TestRun initialization. When the function is entered for the first time, the model's *XXX_New()* function has not yet been called so absolutely no tunable parameter information is available. Still, this is the time when dictionary quantities must be defined, so the necessary information about the tunable parameter's type must be supplied explicitly to *MatSupp_TunDDictDefScalar()*, e.g. *INFOMAT_DOUBLE* in our case.

Only scalar tunable parameters may be defined as dictionary quantities. For complex tunable parameters the quantity will reflect only the state of the real part.

8.7.6 Tunable parameter interface functions

MatSupp_TunRead() – Read a parameter from an infofile

Syntax

```
int MatSupp_TunRead (
    const tMatSuppTunables *tuns,
    const char *param,
    const struct tInfos *inf,
    const char *key
)
```

Description

Reads the value of tunable parameter *param* from the entry named *key* in infofile *inf*. In case the name of a struct parameter is passed, the value of all struct members will be read from the infofile.

The function is intended to be used from within *XXX_SetParams()* in the model wrapper. In this context a valid *tuns* handle as well as a vehicle data infofile handle *Inf* are available by default.

If successful, the function returns 0. In case of an error, -1 is returned and an error message will be issued to the CarMaker log.

Example

```
MatSupp_TunRead(tuns, "MyParam", Inf, "Body.mass");
```

MatSupp_TunReadDef() – Read a parameter from an infofile

Syntax

```
int MatSupp_TunReadDef (
    const tMatSuppTunables *tuns,
    const char *param,
    const struct tInfos *inf,
    const char *key,
    const struct tInfoMat *def
)
```

Description

Reads the value of tunable parameter *param* from the entry named *key* in infofile *inf*. If for any reason the value cannot be set from the infofile entry, the provided default value is used instead. A default value of *NULL* means to use the parameter's original value as the default.

The function is intended to be used from within *XXX_SetParams()* in the model wrapper. In this context a valid *tuns* handle as well as a vehicle data infofile handle *Inf* are available by default.

If successful, the function returns 0. In case of an error, -1 is returned and an error message will be issued to the CarMaker log.

Example

```
MatSupp_TunReadDef(tuns, "MyParam1", Inf, "Steering.Rack2StWh1", NULL);

tInfoMat *mat = InfoMatMakeScalar(INFOMAT_DOUBLE, 0, 3.14, 0);
MatSupp_TunReadDef(tuns, "MyParam2", Inf, "Body.mass", mat);
```

MatSupp_TunReadAll() – Read all parameters from an infofile

Syntax

```
int MatSupp_TunReadAll (
    const tMatSuppTunables *tuns,
    const struct tInfos *inf,
    const char *keyprefix
)
```

Description

This is a convenience function that reads the values of all tunable parameters of a Simulink model from the infofile *inf*. Each parameter's name is used as the key to look up the corresponding infofile entry.

Optionally a prefix can be prepended to all names – e.g. if a parameter's name is "kappa" and *keyprefix* is "SuperABS", the resulting infofile key will be "SuperABS.kappa". A *keyprefix* value of `NULL` or an empty string "" both mean that no prefix should be prepended.

Internally the function calls `MatSupp_TunRead()`, i.e. for each parameter a corresponding infofile entry is considered to be mandatory. If an entry for a parameter cannot be read an error message will be issued to the CarMaker log.

The function returns the number of parameters that could not be read.

The function is intended to be used from within `XXX_SetParams()` in the model wrapper. In this context a valid *tuns* handle as well as a vehicle data infofile handle *Inf* are available by default.

Example

```
MatSupp_TunReadAll(tuns, Inf, Modelname);
MatSupp_TunReadAll(tuns, Inf, "SuperABS");
MatSupp_TunReadAll(tuns, Inf, "");
```

MatSupp_TunReadAllOpt() – Read all parameters from an infofile

Syntax

```
void MatSupp_TunReadAllOpt (
    const tMatSuppTunables *tuns,
    const struct tInfos *inf,
    const char *keyprefix
)
```

Description

This is a convenience function that reads the values of all tunable parameters of a Simulink model from the infofile *inf*. Each parameter's name is used as the key to look up the corresponding infofile entry. As default, this function is called automatically in each wrapper file.

Optionally a prefix can be prepended to all names – e.g. if a parameter's name is "kappa" and *keyprefix* is "SuperABS", the resulting infofile key will be "SuperABS.kappa". A *keyprefix* value of `NULL` or an empty string "" both mean that no prefix should be prepended.

Internally the function calls `MatSupp_TunReadDef(..., NULL)`, i.e. for each parameter a corresponding infofile entry is considered to be optional. No error will be issued if a parameter cannot be read.

The function is intended to be used from within `XXX_SetParams()` in the model wrapper. In this context a valid *tuns* handle as well as a vehicle data infofile handle *Inf* are available by default.

Example

```
MatSupp_TunReadAllOpt(tuns, Inf, Modelname);
MatSupp_TunReadAllOpt(tuns, Inf, "SuperABS");
MatSupp_TunReadAllOpt(tuns, Inf, "");
```

MatSupp_TunGetDbl() – Get a parameter's value

Syntax

```
double MatSupp_TunGetDbl (
    const tMatSuppTunables *tuns,
    const char *param
)
```

Description

Returns the value of tunable parameter *param*, automatically converting from the parameter's internal type to type *double*.

Only non-complex scalar parameters may be accessed with this function. For other types use `MatSupp_TunGetDblVec()` or `MatSupp_TunGetMat()`.

In case of an error the function returns 0.0 and an error message will be issued to the Car-Maker log.

The function is intended to be used from within `XXX_SetParams()` in the model wrapper. In this context a valid *tuns* handle is available by default.

Example

```
kappa = MatSupp_TunGetDbl(tuns, "kappa");
```

MatSupp_TunGetDblVec() – Get a parameter's value

Syntax

```
double *MatSupp_TunGetDblVec (
    const tMatSuppTunables *tuns,
    const char *param,
    int *nvalues
)
```

Description

Returns the value of tunable parameter *param* in an array of type *double*, automatically converting from the parameter's internal type. The number of elements will be stored in *nvalues*. After use the array should be *free()*'ed by the caller.

Both non-complex scalar and vector parameters may be accessed with this function. For other types *MatSupp_TunGetMat()* should be used.

In case of an error the function returns `NULL` and an error message will be issued to the Car-Maker log.

The function is intended to be used from within *XXX_SetParams()* in the model wrapper. In this context a valid *tuns* handle is available by default.

Example

```
double *values;
int i, nvalues;

values = MatSupp_TunGetDblVec(tuns, "MyParam", &nvalues);
for (i=0; i<nvalues; i++)
    Log("MyParam[%d] = %g\n", i, values[i]);
free(values);
```

MatSupp_TunGetMat() – Get a parameter's value

Syntax

```
tInfoMat *MatSupp_TunGetMat (
    const tMatSuppTunables *tuns,
    const char *param
)
```

Description

Returns the value of tunable parameter *param* in a *tInfoMat* structure. After use *InfoFreeMat()* should be called to release the *tInfoMat* structure.

Almost all parameter types, real or complex, may be accessed with this function.

In case of an error the function returns `NULL` and an error message will be issued to the Car-Maker log.

The function is intended to be used from within *XXX_SetParams()* in the model wrapper. In this context a valid *tuns* handle is available by default.

Example

```
tInfoMat *mat;

mat = MatSupp_TunGetMat(tuns, "MyParam");

/* ...modify mat's value as you like... */

MatSupp_TunSetFromMat(tuns, "MyParam", mat);
InfoFreeMat(mat);
```

MatSupp_TunSetFromDbl() – Set a parameter's value

Syntax

```
int MatSupp_TunSetFromDbl (
    const tMatSuppTunables *tuns,
    const char *param,
    double value
)
```

Description

Sets tunable parameter *param* to the specified *value*, automatically converting to the parameter's internal type.

Only non-complex scalar parameters may be accessed with this function. For other types use *MatSupp_TunSetDblVec()* or *MatSupp_TunSetMat()*.

If successful, the function returns 0. In case of an error, -1 is returned and an error message will be issued to the CarMaker log.

The function is intended to be used from within *XXX_SetParams()* in the model wrapper. In this context a valid *tuns* handle is available by default.

Example

```
MatSupp_TunSetFromDbl(tuns, "MyParam", 42.0);
```

MatSupp_TunSetFromDblVec() – Set a parameter's value

Syntax

```
int MatSupp_TunSetFromDblVec (
    const tMatSuppTunables *tuns,
    const char *param,
    int nvalues,
    const double *values
)
```

Description

Sets tunable parameter *param* to the data stored in the *values* array with *nvalues* many entries, automatically converting the data to the parameter's internal type.

Both non-complex scalar and vector parameters may be accessed with this function. For other types *MatSupp_TunSetMat()* should be used.

If successful, the function returns 0. In case of an error, -1 is returned and an error message will be issued to the CarMaker log.

The function is intended to be used from within *XXX_SetParams()* in the model wrapper. In this context a valid *tuns* handle is available by default.

Example

```
double vec[3];  
  
vec[0] = 3.14;  
vec[1] = 6.28;
```

```
vec[2] = 9.42;
MatSupp_TunSetFromDblVec(tuns, "MyParam", 3, vec);
```

MatSupp_TunSetFromMat()

Syntax

```
int MatSupp_TunSetFromMat (
    const tMatSuppTunables *tuns,
    const char *param,
    const tInfoMat *value
)
```

Description

Sets tunable parameter *param* to the specified *value*.

Almost all parameter types, real or complex, may be accessed with this function.

If successful, the function returns 0. In case of an error, -1 is returned and an error message will be issued to the CarMaker log.

The function is intended to be used from within *XXX_SetParams()* in the model wrapper. In this context a valid *tuns* handle is available by default.

Example

```
tInfoMat *mat;
double vec[3];

vec[0] = 3.14;
vec[1] = 6.28;
vec[2] = 9.42;
mat = InfoMatMakeRowVector(INFOMAT_DOUBLE, 3, vec, NULL);
MatSupp_TunSetFromMat(tuns, "MyParam", mat);
InfoFreeMat(mat);
```

MatSupp_TunDDictDefScalar() – Define a parameter quantity

Syntax

```
int MatSupp_TunDDictDefScalar (
    const tMatSuppTunables *tuns,
    const char *param,
    tInfoMatType type,
    const char *name, const char *value
)
```

Description

Makes tunable scalar parameter *param* an analog, non-monotonic CarMaker dictionary quantity. The quantity can be given a name different from that of the parameter.

The *type* argument serves as an aid when the function is called with *tuns* equal to `NULL`. It must match the tunable parameter's internal type (e.g. `INFOMAT_DOUBLE`).

If successful, the function returns 0. In case of an error, -1 is returned and an error message will be issued to the CarMaker log.

The function is intended to be used from within *DeclParameterQuants()* in the model wrapper. In the latter context an appropriate *tuns* handle is available by default.

Examples

```
MatSupp_TunDDictDefScalar(tuns, "MyParam", INFOMAT_SINGLE, "kappa", "kg/s");

/* Use the automatically generated model name as a prefix. */
MatSupp_TunDDictDefScalar(tuns, "MySwitch", INFOMAT_UINT32, QUOTE(MODEL) ".enable", "");

/* Use a fixed model name as a prefix. */
MatSupp_TunDDictDefScalar(tuns, "MySwitch", INFOMAT_UINT32, "MyModel.enable", "");
```

MatSupp_TunListAll() – Enumerate all tunable parameters by name

Syntax

```
char **MatSupp_TunListAll (
    const tMatSuppTunables *tuns,
    int *pcount
)
```

Description

Returns the names of all tunable parameters of the model in a list. The list is NULL-terminated. If *pcount* is non-NULL, the number of items in the list will be stored at the specified location.

After use the list and its members must be *free()*'ed by the caller (alternatively function *InfoFreeTxt()* can be used, it is declared in <infoc.h>).

In case the model has no tunable parameters, an empty NULL-terminated list is returned.

Example

```
int nparams, i;
char **params = MatSupp_TunListAll(tuns, &nparams);
for (i=0; i<nparams; i++)
    Log("Tunable parameter #%d: %s\n", i, params[i]);
InfoFreeTxt(params);
```

8.8 Miscellaneous

8.8.1 Upgrading to a new CarMaker version

When upgrading to a new CarMaker version, always be sure to update your `cmenv.m` script as well:

- The `cmenv.m` script is updated automatically when you start a project update from the CarMaker main GUI (File > Update Project).
- It can be also replaced manually with its successor, to be found in `Templates/Car/src/cmenv.m` in the installation directory of the new CarMaker version.
- Or change the setting of the `cminstdir` variable in the first few lines of your `cmenv.m` script and make it point to the new version.

There might be other files, too, that may need to be updated. Check the wrapper source code and regenerate your model C-code. The *Release Notes* document coming with Car-Maker will provide you with the necessary information.

8.8.2 CarMaker and Matlab installed on different computers

If CarMaker and Matlab are not available on the same computer, two problems arise:

- C-code generation with Simulink Coder (formerly Real-Time Workshop) is not possible because the CarMaker target is not available on the Matlab host.
- Compiling the model library of a Simulink model is not possible because the necessary Matlab *.h include files are not available on the CarMaker host.

A solution exists for both problems.

Making the CarMaker target available on the Matlab host

The text below describes the steps for a CarMaker host running Linux and a Matlab host running MS Windows, but operating systems do not really play an important role here. The same technique also applies to scenarios with a different “operating system mix”; of course OS specific file and directory names need to be adapted accordingly.

Copying the CarMaker target directory

Copy the directory containing the CarMaker target

```
/opt/ipg/hil/<version>/Matlab
```

(i.e. the one in the installation directory of the CarMaker version you use) from the CarMaker host to the Matlab host. Be sure that the copy is complete and includes all subdirectories.

All S-functions coming with the CarMaker target need to be recompiled for the Matlab version installed on the Matlab host. For this reason the source code of all S-functions is part of the package.

Start Matlab and change to the directory of the CarMaker target that matches your Matlab version (here version 7.5 is assumed):

```
>> cd <location_of_the_copy_you_made>/Matlab/v75-r2007b
```

Invoke the m-file script that recompiles all S-functions:

```
>> mksfun
```

The compilation process should proceed without any warnings or errors.

After compilation is done, check that the following files exist in your current working directory:

```
>> dir *.dll
mhdefhil.dll
mhfromhil.dll
mhfromhilvar.dll
mhtohil.dll
mhtohilvar.dll
```

If all files exist, you are done. The CarMaker target for Simulink Coder is ready for use.

Generating C code

In order to use the CarMaker target on the Matlab system, its directory must be added to Matlab's search path. This was described in detail at the beginning of this manual. But since the Matlab system is without a proper CarMaker installation, the *cmenv.m* script will very likely not work. That means before you can use the package you have to issue the necessary command in Matlab manually (or put it into a small m-file script). In our case, the command looks like:

```
>> addpath <location_of_the_copy_you_made>/Matlab/v75-r2007b)
```

After that everything should work as described in the other chapters of this manual. C-code generation with Simulink Coder and the CarMaker target should be possible without any problems.

Make sure that the **Generate code only** checkbox in the Simulink Coder tab of the *Simulation parameters* dialog is checked, since compilation can only be done on the CarMaker system. Thus, after code generation is complete, all generated files must be transferred to the CarMaker system.

Important note

When transferring files between the CarMaker host and the Matlab host, make absolutely sure that you use identical file names on both sides. Beware of subtle differences in terms of upper/lower case of characters.

Differing filenames may lead to compilation errors on the CarMaker system. Both the generated C-code and the generated Makefile contain the original file names and identifiers derived from them and are very sensitive to changes in the naming of files.

“File not found”, “No such file or directory” or “Do not know how to make XXX” when compiling a model are typical indications of unexpectedly renamed files.

Making the Matlab include files available on the CarMaker host

A supplemental CarMaker package (*CarMaker-matinc-XX.tgz*) containing the Matlab include files is available on request. Simply add the package to your CarMaker installation using the IPG installer.

In order to make use of the newly installed include files, Simulink Coder must be informed that the include files can be found somewhere in the CarMaker installation directory. Open the **Simulation / Simulation Parameters...** dialog of your model. On the **Simulink Coder** (or Real-Time Workshop) tab, the code generation options for the CarMaker target must be changed as follows:

- Category: CarMaker code generation options
- Matlab include path: `$(MATSUPP_DIR)/matlab`

Now regenerate the C code of your model.

If regeneration of the C code is not an option, an alternative would be to edit the generated *XXX.mk* Makefile of your model C code and change the value of the *CM_MATINCPATH* variable to the above mentioned value:

```
CM_MATINCPATH = "$(MATSUPP_DIR)/matlab"
```

After the changes are made, you should be able to compile the model library without a Matlab installation available on the CarMaker host.

8.8.3 Troubleshooting

Simulink Coder automatically deletes files

Since Matlab version 6.0, Simulink Coder (formerly Real-Time Workshop) places the C-code files generated from a Simulink model in a subdirectory <ModelName>_<Target-Name>_rtw. Since Matlab version 6.5, Simulink Coder, before generating new code, automatically does a cleanup in this subdirectory and deletes files it considers to be "old" or "temporary".

This behaviour may collide with the common practice of keeping additional user code needed to compile the model in this directory. C-code files, as well as e.g. customized Makefiles are affected.

Closer observation shows that files containing the text "target specific file" in the very first line will not be touched or deleted by Simulink Coder, so this might be a workaround to the problem. Nevertheless it seems to be safer to keep user code files in a different place and not in the subdirectory obviously under Simulink Coder's control.

Integration of multiple models leads to link error

The C-code generated by Simulink Coder contains a function *rt_CallSys()*. This function has global scope, which makes the linker issue a "duplicate definition error" when linking two or more Simulink models into CarMaker.

Since *rt_CallSys()* does not seem to be used anywhere in the code or in the runtime library, a possible workaround is commenting out the function completely in the generated source code. The drawback of this method is, that each time you generate code from one of the Simulink models, you will have to repeat this step.

Building the model library fails with the following error: 'No rule to make target 'xyz.o' ... Stop'.

A frequent cause for this error is that your Simulink model contains one or more S-functions whose C source files are not located where the generated model Makefile (*.mk) expects them. The C source code of all S-functions of your model is required for building the model library. The related source files must be in the same directory as the Simulink model. This is a restriction imposed by Simulink Coder, not by the CarMaker target.

Another possible cause might be, that an S-function was built from more than a single source file. How to cope with this case depends on the Matlab version used:

Starting with Matlab 7.0.1 (R14 SP1) an S-function's parameter dialog provides an additional entry field **S-function modules**, allowing to specify all source files belonging to this particular S-function. Simulink Coder makes use of this information when generating the model Makefile (*.mk). If the field is left empty, a single C source file with the name of the S-function is expected to be found in the same directory as your Simulink model. When the build fails with the above mentioned error message, check that really all files are specified correctly and that all source files are in place.

Older versions of Real-Time Workshop have no way of knowing what an S-function's source files are, so a single C file named after the S-function and residing in the same directory as the Simulink model is assumed. To solve the build problem copy all required S-function source files into the directory where your Simulink model resides and edit the generated Makefile (or preferably a copy of it) so that it knows about the names of all required source files.

8.9 Demonstration examples

The CarMaker target for Simulink Coder comes with example models for nearly all plugin kinds. They can be added to your CarMaker project directory in the *New Project* dialog of the CarMaker GUI, checkbox item *Simulink Coder (RTW) Examples*.

8.9.1 Contents of the examples directory

Each example *XXX* typically consists of the following parts:

- *src/XXX_RTW.mdl*
- *src/XXX_RTW_bus.m*
- *src/XXX_RTW_params.m*
src/XXX_CarMaker_rtw/XXX_wrap.h (for some models)
src/XXX_CarMaker_rtw/XXX_wrap.c (for some models)
Simulink model of the example, together with the wrapper source files.
- *Examples/Simulink/...* (for some models)
Examples/ExtraModels/...
TestRun and vehicle parameter files prepared to run the example.

8.9.2 Preparing the examples

Before you can actually use the examples, some preparation is necessary.

Just like explained before, *Makefile* need to know which Matlab version you are using. In order to use example *XXX*, open the *src/Makefile* using your favourite ASCII editor and change the line

```
MATSUPP_MATVER = v7.2-r2006a
```

The Matlab version should match the one you are using; please check the Release Notes for Matlab versions supported by your current CarMaker version.

8.9.3 Rebuilding an example

A complete rebuild of example *XXX* just takes a few steps.

- In the *src* directory, open the model *XXX.mdl* using Matlab.
- Start the Simulink Coder *Build* by selecting *Tools > Simulink Coder (or Realtime Workshop) > Build Model*.
- The model is compiled in C-code, directly hooked into the CarMaker environment and a new CarMaker executable is created. Matlab can be closed.
- By activating the newly created executable in the CarMaker standalone GUI, the example models are available. Select the prepared test run under Examples/Simulink and start a new simulation.

These are the same steps as described in section 8.3 'Integration of Simulink models with CarMaker's Model Plug-in' on page 163

Chapter 9

Functional Mock-up Interface (FMI)

The Functional Mock-up Interface (FMI) offers the possibility to make use of models exported by third party tools in a standardized form as so-called Functional Mock-Up Units (FMUs).

CarMaker supports most of the functionality of FMI version 1.0. However, the FMU needs to meet the following constraints:

- The FMU must follow the specification of *FMI for Co-Simulation Stand Alone* (not *FMI for Co-Simulation Tool*)
- The model needs to fit to one of CarMaker's model classes for FMI.
- The FMU must contain binary code for the platform you want to simulate on.

The FMU will then be dynamically linked to the current CarMaker application. This means, the FMU is co-simulated. There is no need to compile a new CarMaker executable.

However, before an FMU can be used in the CarMaker environment, it must be integrated and configured.

9.1 Integrating an FMU

The *FMU Plug-ins* dialog which is accessible via *CarMaker main GUI > Application > FMU Plug-ins* supports the user during integration and configuration of an FMU. To integrate a new FMU into your simulation environment, you have to import the FMU first. This can be achieved by using the *Import* button on the right side of the FMU dialog's *Overview* tab: Select the desired FMU and it will be copied automatically into the *Plugins* folder of your project directory.

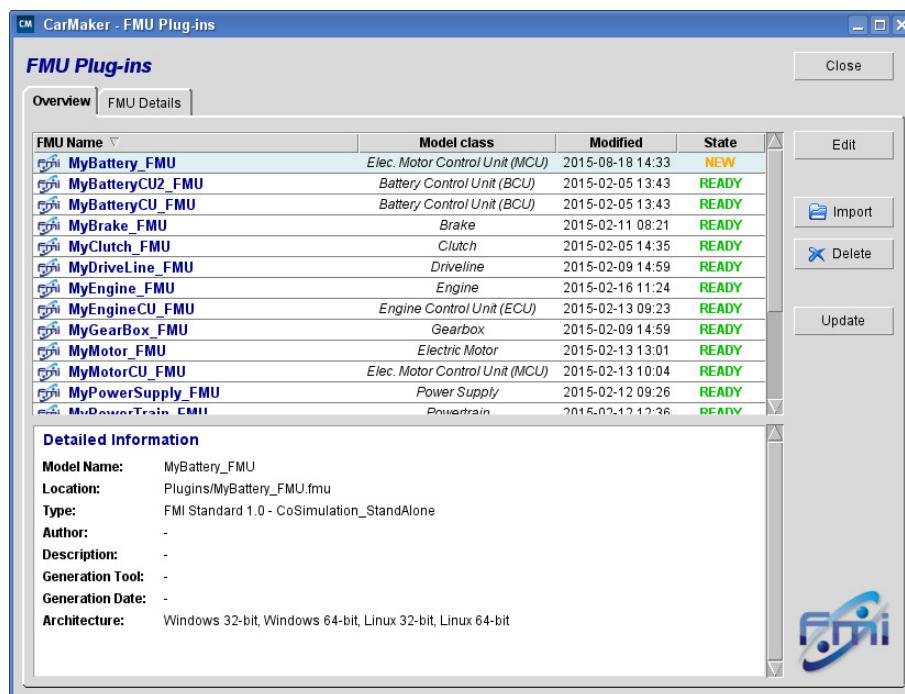


Figure 9.1: FMU dialog with available FMUs in the project directory

On the first tab of the FMU dialog shown in [Figure 9.1](#), all available FMUs are listed in an overview. Besides the name of the FMU you will find some more information like the model class (if the FMU has already been integrated) and the modification date. The column *State* describes the current state of the FMU. There are three possible states:

- NEW** A new FMU which has not been integrated in the CarMaker simulation environment yet.
- READY** The respective FMU is integrated in the CarMaker simulation environment and is ready to use.
- ERROR** An error occurred during the loading of the FMU or the FMU is not supported by the Car-Maker simulation environment. If an error occurred, you will find more information in the lower part of the window.



Running the same FMU multiple times

In case, the same FMU should be integrated several times, this is possible without configuring the in- and output signals once again.

When importing an FMU into a project directory where an FMU with the same name already exists, the user is prompted to enter a new name. The given name also defines the prefix for the respective quantities of the FMU. If a mapping has already been defined for the existing, identical FMU this will be imported automatically for the new one.



Figure 9.2: Managing multiple instances of same FMU

9.2 Configuring the FMU

If the FMU is opened in CarMaker for the first time, it must be adapted to the CarMaker simulation environment before it can be used. Select the desired FMU and click *Edit* to change to the tab *FMU Details* shown in Figure 9.3.

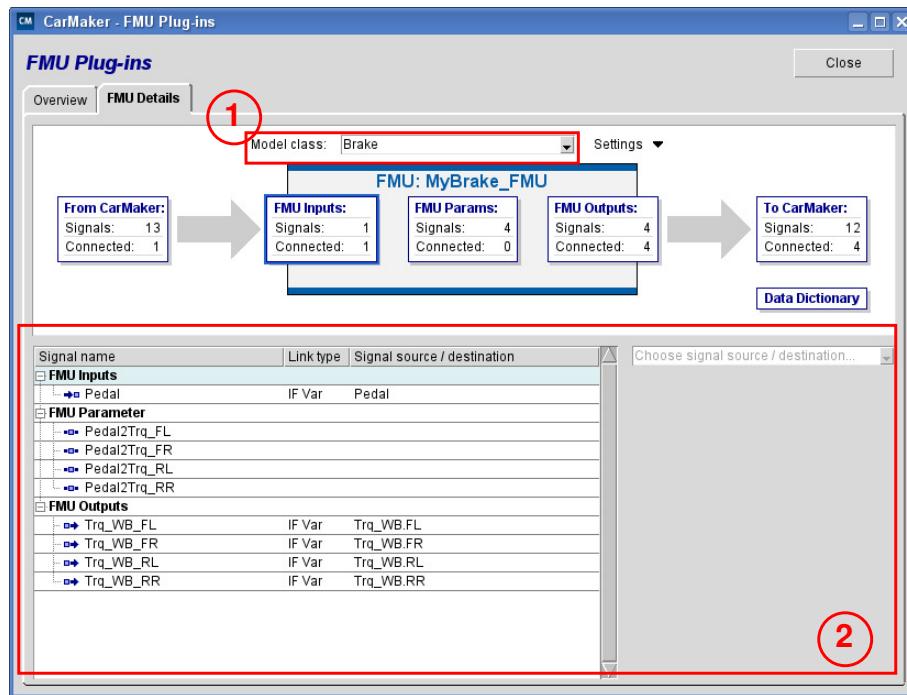


Figure 9.3: FMU Edit

9.2.1 Selecting the Model Class

Before the FMU can be configured, the appropriate model class must be selected (see (1) in Figure 9.3). Possible model classes are the following:

Table 9.1: Model class and corresponding Infofile key

Model class	Infofile Key
Aerodynamics	Aero
Brake	Brake
Brake Controller (Hydraulic)	Brake.Control
Brake System (Hydraulic)	Brake.System
Brake Controller (Pneumatic)	Brake.Control
Brake System (Pneumatic)	Brake.System
Environment	Env
PowerTrain	PowerTrain
PowerTrain OpenXWD	PowerTrain.PTXWD
Battery	PowerTrain.PowerSupply.BattLV PowerTrain.PowerSupply.BattHV
Battery Control Unit (BCU)	PowerTrain.BCU
Clutch	PowerTrain.Clutch

Table 9.1: Model class and corresponding Infofile key

Model class	Infofile Key
Coupling Model	PowerTrain.DL.FDiff.Cpl PowerTrain.DL.RDiff.Cpl PowerTrain.DL.CDiff.Cpl PowerTrain.DL.HangOn.Cpl
DriveLine	PowerTrain.DL
DriveLine OpenXWD	PowerTrain.DLXWD
Electric Motor	PowerTrain.MotorISG PowerTrain.Motor PowerTrain.Motor1 PowerTrain.Motor2 ...
Elec. Motor Control Unit (MCU)	PowerTrain.MCU
Engine	PowerTrain.Engine
Engine Control Unit (ECU)	PowerTrain.ECU
Gearbox	PowerTrain.GearBox PowerTrain.GearBoxM PowerTrain.GearBoxM1 PowerTrain.GearBoxM2 ...
Power Supply	PowerTrain.PowerSupply
Powertrain Control	PowerTrain.Control
Powertrain State Control (OSM)	PowerTrain.ControlOSM
Transmission Control Unit (TCU)	PowerTrain.TCU
Steering	Steering
Suspension External Forces	SuspExtFrcs
Suspension Buffer	SuspExtFrcs.Buffer
Suspension Damper	SuspExtFrcs.Damper
Suspension Spring	SuspExtFrcs.Spring
Suspension Stabilizer	SuspExtFrcs.Stabi
Tire	Tire.0 Tire.1 ...
Tire Contact Point Modification	TireCPMod
User Driver	UserDriver
Vehicle Control	VehicleControl
Vehicle Operator	VhclOperator
Generic (if no other subsystem matches)	GenericPlugin.<Name_FMU> section 'Generic Plug-in Models'

9.2.2 Connecting Input and Output Signals

When this is done, all FMU input parameters must be connected to the suitable input/output signals. For this, use the table as shown in (2) in [Figure 9.3](#).

The source for a FMU input signal can be

- a Real Time Expression (see User's Guide, appendix "Realtime Expressions")
- a constant value
- an interface output variable
- any Data Dictionary quantity (UAQ)

On the output side of the FMU the following items can be connected to the FMU output signals:

- the interface inputs of the model class
- RealtimeExpressions
- constant values
- any Data Dictionary quantity (UAQ)

To adapt the FMU to your needs, select the suitable CarMaker model class and connect the input and output signals. The current configuration is saved when the view is changed back from the *FMU Details* tab to the *Overview* tab, when a simulation is started or when the Car-Maker GUI is closed.

9.2.3 Special Options

Some helpful options are available in the *Settings* drop down menu.

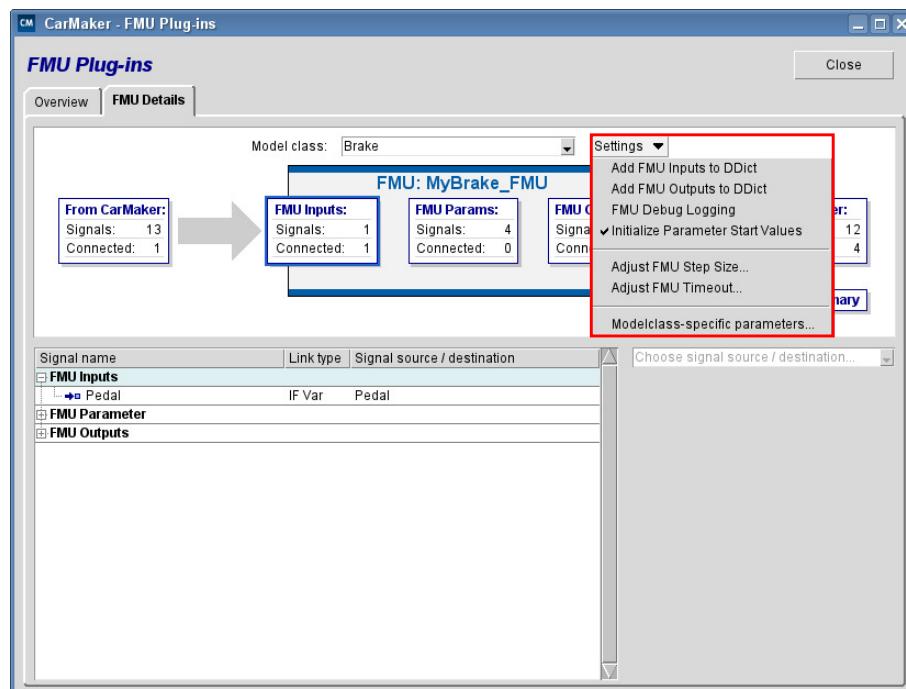


Figure 9.4: Settings drop down menu with additional options

- Import FMU configuration** If the same or a similar FMU with identical input/output parameters was integrated in your CarMaker simulation environment before, the respective parameters are automatically assigned and must not be connected to the suitable parameters again. If you want to adopt a previously defined configuration from another FMU, you can import the complete configuration by choosing *Import FMU configuration* from the *Settings* menu (see [Figure 9.4](#)).
- Reset FMU configuration** Choosing *Reset FMU Configuration* reverts all changes you made since the FMU's detailed view was opened.
- Add FMU In-/Outputs to DDict** The context menu also allows you to add the FMU's inputs and outputs to the CarMaker Data Dictionary as quantities. This is especially helpful for debugging or if the content of the FMU output is used elsewhere in your simulation model (e.g. as input for another FMU).
- FMU Debug Logging** To activate the storage of the FMU's log messages choose *FMU Debug Logging*. See [section 9.5 'FMU Debugging Assistsances'](#).
- Initialize Parameter Start Values** If the FMU should use other start values than those pre-defined in the FMU, some parameters can be added to the vehicle data set. To load these parameters from the vehicle Info-file at simulation start and to initialize these as start values for the FMU, use the option *Initialize Parameter Start Values* (see [section 9.3.4 'FMU Parameterization'](#)).
- Adjust FMU Step Size** Additionally, the *Adjust Step Size* command lets you specify a custom step size for the currently selected FMU. Please note that the step size of the application and of the FMU have to be multiples of each other.
- Adjust FMU Timeout** With this parameter it is possible to specify a custom timeout for the FMU initialization (parameter to function `fmiInstantiateSlave()` of the FMU) for FMUs which do not accept Car-Maker's default value.
- Modelclass-specific Parameters** In some cases the model class needs to provide a set of parameters for the remaining Car-Maker environment. These additional parameters can be edited manually in the *Modelclass-specific Parameters* window which is accessible in the *Settings* menu of the FMU dialog.



In case a model class needs to provide certain parameters to the CarMaker periphery, they are listed in the in the *Modelclass-specific Parameters* of the example FMUs provided by IPG. Lines marked with a hash symbol (#) are comments to explain the meaning of the parameter and its default value if not specified otherwise. The parameters and their values are decribed in detail in the Reference Manual, in one of the last section "User C-code/ Simulink Plug-in/ FMU"of the corresponding modelclass.

As example for *Modelclass-specific Parameters* consider a gearbox model which should be implemented as FMU. IPGDriver needs certain information for its shifting logic like the gearbox kind (manual or automatic) and the transmission ratios of the single gears:

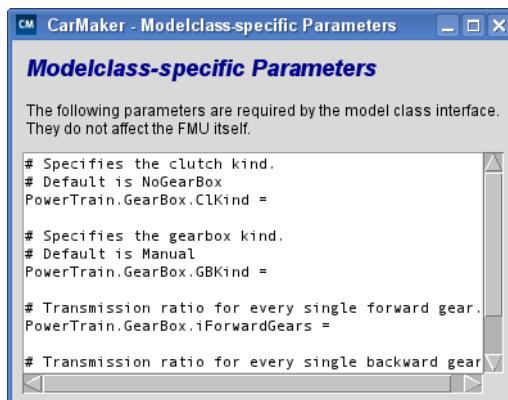


Figure 9.5: Modelclass specific parameters of example FMU "MyGearBox_FMU"

The *Modelclass-specific Parameters* cannot be used to enter parameters specific to the FMU itself. They only provide interface parameters to the remaining CarMaker environment. Note, that there are model classes which do not need any such parameters.

FMU model specific parameters can be listed under *Additional Parameters* in the vehicle data set, using the model class specific prefix (see [Table 9.1:](#)).

9.3 Simulating with an FMU

To start a simulation using the configured FMU, you must select the desired FMU via the model selection menu of the respective CarMaker subsystem.

9.3.4 FMU Parameterization

FMU parameters are set once before initialization of the model.

In a first step each parameter is set to its start value as specified in the FMU's internal description file (*modelDescription.xml*). This step might seem unnecessary, as these values usually describe only the compiled-in start values in the FMU's program code. However, this allows to modify the start values simply by manually unpacking the FMU (basically a ZIP-file), editing the *modelDescription.xml* file and packing everything again. This is sometimes useful for debugging purposes. The setting of parameter start values from the *modelDescription.xml* file can be disabled by unchecking the *Initialize Parameter Start Values* menu entry in the *FMU Plug-ins* dialog for the FMU in question.

Afterwards, as a second step, parameter start values are read (if possible) from appropriate Infofile entries specified as additional parameters in the vehicle's *Additional Parameters* window: The key which identifies an FMU parameter is assembled from the appropriate model class string (as shown in the third column of table 5.1) and the parameter name itself. For example, if the selected FMU resembles a gearbox, the FMU parameter's name is *Param1* and the desired value is *1* the correct entry would be:

```
PowerTrain.GearBox.Param1 = 1
```

9.4 Example FMUs

CarMaker comes with several example FMUs, available as an optional extra when creating or updating a CarMaker project directory:

- MyBatteryCU_FMU (battery control unit)
- MyBattery_FMU (battery model)
- MyBrake_FMU (brake model)
- MyClutch_FMU (clutch model)
- MyDriveLine_FMU (drive torque distribution to wheels)
- MyEngineCU_FMU (combustion engine control unit)
- MyEngine_FMU (combustion engine model)
- MyGearBox_FMU (gear box model)
- MyMotorCU_FMU (electric motor control unit)
- MyMotor_FMU (electric motor model)

- MyPowerSupply_FMU (battery energy and voltage)
- MyPowerTrain_FMU (conventional powertrain model)
- MyPTControl_FMU (powertrain control unit)
- MyPTControlOSM_FMU (powertrain state control, operation state machine)
- MyPTGenCoupling_FMU (differential model)
- MySuspExtForces_FMU
(external suspension force model for buffer, spring, stabilizer, damper)
- MyTransmCU_FMU (transmission control unit)
- MyVehicleControl_FMU (vehicle control unit for throttle manipulation)

As one may conclude from the names, the implementation of each FMUs closely follows its counterpart from the *Sources: Extra Models* collection.

After installation the FMUs and their .plugininfo files are located in the *Plugins* directory. A vehicle data file for each FMU is also included (*Examples/FMI* directory).

9.5 FMU Debugging Assistances

Sometimes the situation arises, that an FMU does not behave as expected when run by a different master (CarMaker in this case), so that it might be useful to switch on debug logging. The particular FMU can then tell more about its internal state during the simulation and the additional log output might provide valuable information to the creator of the FMU or to the developer of the tool with which the FMU was generated.

9.5.5 Turning on debug logging

The FMU log messages can be recorded in an external file (not CarMaker Session Log!). The FMU.log file will be created, once the following SimParameters key is set to 1:

```
FMU.LoggingToFile
```

The content of the FMU log file can be specified by excluded several categories. Again, this can be set in the SimParameters. Please find further information in the Reference Manual, chapter “SimParameters”.

Alternatively, the Debug logging for an FMU can be activated on tab *FMU Details* by right-clicking on the overview in the upper part of the window and selecting *FMU Debug Logging* in the menu that appears (see [Figure 9.4](#)). In the same way debug logging can be turned off again. This option will write to CarMaker’s log file.



It is NOT recommended to leave debug logging permanently switched on, as this may slow down calculations significantly and swamp the CarMaker log file with unnecessary output.

9.5.6 Handling voluminous logging output

With debug logging turned on, depending on the output of the FMU, a huge number of log messages might be issued. This might lead to the situation that upon reaching CarMaker’s internal logfile size limit, all further logging output will be ignored. The size limit (about 10 MB) was chosen intentionally, is hardcoded into CarMaker and cannot be changed by the user.

CarMaker offers several ways to handle excessive logging of FMUs. They all apply to logging output from FMUs in general – debug logging enabled is not a prerequisite.

Redirecting logging output to an external file

In order to separate FMU debug logging output from regular CarMaker log messages, log messages of FMUs in general can be sent to an external file. This is controlled by the following entry in the project directory's *Data/Config/SimParameters* file:

```
FMU.Logging.ToFile = 1
```

Value 1 means logging output of all FMUs will be sent to file *FMU.log* in your project directory and no file size limit will be imposed by CarMaker. Default value is 0, i.e. messages will go to the CarMaker logfile. Do not forget to remove the extra file when debugging is done.

Start logging output at defined simulation time

CarMaker provides the possibility to suppress all FMU log messages before a given time in the simulation, by adding an entry named *FMU.Logging.Start* to the FMU's .pluginfo file.

```
FMU.Logging.Start = value
```

Logging will start, when SimCore.Time reaches the given value. There is also the possibility to specify a global time, when FMU logging should start. To create a global setting, simply place an entry like the one above in the project directory's *Data/Config/SimParameters* file.

Filtering logging output by log category

When an FMU issues a log message, it must also provide a log category for the message. The log category is a simple string and CarMaker outputs the category directly in front of each message. During debugging, usually not every log message is of interest and often only messages of certain categories prove to be useful.

CarMaker provides a means to filter out specific categories by adding an entry named *FMU.Logging.ExcludeCategs* to the FMU's .pluginfo file. The following example specifies that all messages of categories *info* and *chitchat* be ignored:

```
FMU.Logging.ExcludeCategs:  
    info  
    chitchat  
    #babble
```

The last entry in the list shows a way to temporarily comment out an entry without having to remove the line entirely – list entries starting with a hash character (#) will be ignored.

The categories listed under that entry are specific to the FMU described in the .pluginfo file, but a global list is also available (useful e.g. when an FMU does not provide proper instance information with the log message, so that CarMaker has no means to relate the log message to a particular FMU). To create a global setting, simply place an entry like the one above in the project directory's *Data/Config/SimParameters* file. Both the global and the FMU-specific category exclude lists will be considered when filtering messages.

Filtering logging output with a user-defined function

CarMaker's FMI implementation also allows for fine-grained logging output filtering by letting the user hook-up a C-code function, that decides for each log message issued by an FMU whether it should be thrown away or not. Here is a fictitious example, which fits into file *src/User.c* of your project directory:

```
...  
#include <string.h>  
#include <Log.h>  
...  
static int
```

```
MyFilter (const char *instancename, const char *category, const char *msg)
{
    // Throw away all messages of category "info".
    if (strcmp(category, "info") == 0) return 0;

    // Throw away all messages about module "foo" (i.e. containing the string "foo:").
    if (strstr(msg, "foo:") != NULL) return 0;

    return 1;      // Allow message to be logged.
}

...
int
User_Init (void)
{
    ...
    FMULoggingFilterHook = MyFilter;           // Hook-up my function.
    ...
    return 0;
}
...
```

For details see also the description of *FMULoggingFilterHook* in include file *Log.h*.

Chapter 10

MIO – M-Module Input/Output

10.1 Supported M-Modules

- M27: Binary Output (16 channels)
- M28: Binary Output (16 channels)
- M31: Binary Inputs (16 channels)
- M32: Binary Inputs (16 channels)
- M35 / M35N: Analog Inputs (16/8 channels)
- M36N: 16 Bits Analog Inputs (16/8 channels)
- M43: Relay Outputs (8 channels)
- M51: Quadruple CAN Interface
- M62 / M62N: Analog Outputs (16 channels)
- M66: Binary Inputs/Outputs (32 channels)
- M77: Quadruple RS232/423 – RS422/485 UART
- M81: Binary Outputs (16 channels)
- M400: Wheelspeed Generator, 6 Channels
- M401: Frequency Generator and Frequency Meter (3 Units)
- M402: Engine Signal Generator, 8 Channels
- M403: Engine Signal Detector, 8 Channels
- M404: Waveform Generator, 8 Channels
- M405: LIN Interface, 12 Channels
- M406: PSI5 Interface, 8 Slaves + 1 Master
- M407: SPI Interface, 2 Slaves or 1 Master
- M408: PWM Meter and SENT Receiver, 20 Channels
- M409: Power Supply Control, 2 Units
- M410: Quadruple CAN/CANFD Interface
- M412: Parksensor Simulation Board
- M413: 5x 4:1 Multiplexer / De-Multiplexer

- [M440: Resistor Cascade, 6 Channels](#)
- [M441: PWM Out and SENT Transmitter, 12 / 16 Channels](#)

Discontinued M-Modules

- [M34: Analog Inputs \(16/8 channels\)](#)
- [M72: Motion Counter / Multipurpose Module](#)

10.2 Programming M-Module I/O

The initialization process for the MIO software module depends on your hardware configuration. First, you have to initialize the MIO software module in general. Then, depending on the installed M-Module hardware, you initialize the installed M-Modules by calling the appropriate `MIO_Mxx_Config()` functions. A flow chart of the initialization process is shown in [Figure 10.1](#).

Of course, a wrong initialization may affect the operability of the whole test stand. So, you have to react on errors during the hardware initialization. To simplify this process, the MIO software module logs errors to the session log and informs about with negative return values to MIO initialization functions. However, since the CarMaker Log module allows to check the error counter, it is not necessary to check every single call to `MIO_xx()` functions.

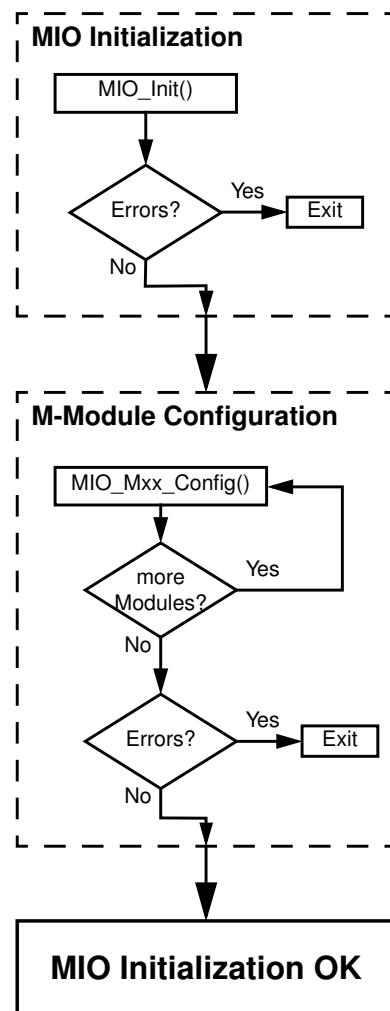


Figure 10.1: Initialization of the MIO software module

10.2.1 MIO Initialization

The first step to initialize the MIO hardware is the initialization of the MIO software module in general, by calling the function `MIO_Init()`. This function initializes the carrier boards and reads the module type of all installed M-Modules.

10.2.2 M-Module Configuration

Now, you have to configure the installed M-Modules, each with the appropriate `MIO_Mxx_Config()` function. Each `MIO_Mxx_Config()` function requires at least one argument, the slot number of the M-Module.

When registering a module, the MIO software module checks, if there is a M-Module installed at the given slot, and if the type of the installed module matches the selected `MIO_Mxx_Config()` function. If a mismatch is detected, an error is logged to the session log and a negative value is returned. However, in some circumstances a module mismatch cannot be ruled out completely: some older M-Modules do not have an Id EEPROM, so the MIO software module cannot determine the type of installed module and has to accept the module type, passed by the `MIO_Mxx_Config()` function.



The slot numbers of the installed M-Modules depend on the hardware configuration, i.e. on the order in which the BIOS and the XENO kernel recognize and initialize PCI devices. In the 19" chassis with 6HE backplane and system slot left (5 slot chassis and 8 slot chassis with system slot at the bottom), the slots are numbered starting with 0 on the topmost rightmost M-Module and increasing to the lowest leftmost M-Module. In the RoadBox – 3HE backplane with system slot right – the slots are numbered starting with 0 on the lower leftmost M-Module and increasing to the upper rightmost M-Module. A typical hardware configuration for a 19" chassis is shown in [Figure 10.2](#):

D203	Slot 3	Slot 2	Slot 1	Slot 0	men
D203	Slot 7	Slot 6	Slot 5	Slot 4	men
D203	Slot 11	Slot 10	Slot 9	Slot 8	men
D203	Slot 15	Slot 14	Slot 13	Slot 12	men
F14N			USB	ETHERNET	men

Figure 10.2: Typical MIO configuration (19" chassis)

After all M-Modules have been registered, and prior to any access to the modules, you have to check again if there was an error during the MIO hardware initialization. If you do not handle errors at this point of time, the application might crash later on when accessing the MIO hardware, because of bus errors or segmentation violations. An example for the error handling with the initialization of the MIO software module is shown in [Listing 10.1](#).

10.2.3 Error Handling

After calling `MIO_Init()`, it is strongly recommended to check, if the initialization was successful, or if an error occurred. If there was an error and you do not handle it before continuing with the configuration of the MIO hardware, the application might crash because of segmentation violations. For an example of the error handling after calling `MIO_Init()`, refer to [Listing 10.1](#).

The rest of the MIO hardware initialization can be done in one step, without special error handling in between the different function calls.

Initializing the MIO hardware is a critical part. If the hardware configuration does not match the requirements of the application, the MIO hardware cannot be accessed safely. Accessing wrong or nonexistent MIO hardware, will produce CompactPCI timeouts and might even crash the application.

Whenever the MIO software module runs into an error during the initialization of a module (e.g. during a `MIO_Mxx_Config()` function), an error message is printed to the session log file. With each error message, the CarMaker Log module increases an error counter, which can be checked later on, to determine if errors have been logged. Therefore, it is not necessary to check for errors after every MIO function call. This helps to keep the source code of your application manageable and readable. However, there are two points of time, at which it is strongly recommended to check for errors: after calling `MIO_Init()` and after the registration of the MIO hardware, before accessing M-Modules. The following [Listing 10.1](#) shows the MIO initialization of a typical CarMaker/HIL MIO hardware configuration. You can use it as a template for your own application.

[Listing 10.1](#): Initialization of the MIO hardware

```

1: int
2: IO_Init (void)
3: {
4:     Log("I/O Configuration: %s\n", IO_ListNames(NULL, 1));
5:
6:     /* hardware configuration "none" */
7:     if (IO_None)
8:         return 0;
9:
10:    /* hardware configuration "demoapp" */
11:    if (IO_DemoApp) {
12:        int nErrors = Log_nError;
13:
14:        /*** MIO initialization */
15:        if (MIO_Init(NULL) < 0) {
16:            LogErrF(EC_General, "MIO initialization failed. I/O disabled (1)");
17:            IO_SelectNone();
18:            return -1;
19:        }
20:        // MIO_ModuleShow ();
21:
22:        /* ModuleSetUp */
23:        MIO_M35_Config (Slot_AD); /* 12bit Analog In, (16 Channels) */
24:        MIO_M62_Config (Slot_DA); /* 12bit Analog Out, (16 Channels) */
25:        MIO_M51_Config (Slot_CAN, -1 /*default IRQlevel*/);
26:
27:        /* check for errors */
28:        if (nErrors != Log_nError) {
29:            LogErrF(EC_General, "MIO initialization failed. I/O disabled (2)");
30:            IO_SelectNone();
31:            return -1;
32:        }
33:
34:        MIO_M51_SetCommParam (Slot_CAN, 0, 500000, 70, 2, 0);
35:        /* configure acceptance filter for Rx CAN messages (here: accept all) */
36:        MIO_M51_EnableIds (Slot_CAN, 0, 0 /* start ID */, 2048 /* num IDs */);
37:        /* when working with extended CAN messages, activate transparent mode */
38:        // MIO_M51_SetTransMode (Slot_CAN, 0, 1);
39:    }
40:
41:    return 0;
42: }
```

Checking for errors can be done by checking the error counter `Log_nError` before, and after the critical part. If the counter has increased, e.g. because one of the `MIO_xx_Config()` functions failed, then the error description has already been written to the session log.

If there was an error during the initialization of the MIO hardware, it is not safe to access any M-Module. It is strongly recommended to disable I/O and prompt the user to solve the problems first.

10.2.4 Custom MIO Drivers

It is possible to integrate custom MIO drivers into CarMaker. This is useful if M-Modules are used, that are not supported by CarMaker by default. Additionally it is possible to replace the CarMaker's default driver for a supported module by a custom driver.

Implementing the Driver

There are some functions, structs and variables that are essentially needed for the MIO drivers. These are shown for an M66 in the listing below.

Listing 10.2: Needed functions for custom MIO drivers. (Part1)

```
1: #include <stdio.h>
2: #include <stdlib.h>
3: #include <string.h>
4: #include "mio.h"
5: #include "mio_m66.h"
6:
7:
8: #define MIO_MT_M66      66
9: #define MIO_M66_nChannels 32
10:
11:
12: typedef struct {
13:     int StateMask;
14:     int OutputMask;
15: } tMIO_M66_Cfg;
16:
17:
18: typedef struct {
19:     unsigned short Channel[MIO_M66_nChannels];
20: } tMIO_M66_A08; /* Hardware Description in the A08 address space
21:
22:
23: /* MIO_M66_Reset - Reset Hardware to initial state */
24: static void MIO_M66_Reset (int Slot)
25: {
26:     /* Do what has to be done to set module to initial state */
27: }
28:
29:
30: /* MIO_M66_Delete - Delete allocated memory but the MIO_Slots[Slot].PrivatePtr */
31: static void MIO_M66_Reset (int Slot)
32: {
33:     /* Do what has to be done to delete allocated memory. Don't free it twice! */
34: }
35:
36:
37: /* MIO_M66_Config - Register Module, Allocate Memory and Reset the Module */
38: int MIO_M66_Config (int Slot)
39: {
40:     tMIO_M66_Cfg *cfg;
41:
42:     /* Register the module with type M66 in <Slot> */
43:     if (MIO_RegisterModule(Slot, MIO_MT_M66) < 0) return -1;
44:     /* Allocate memory for private data. free() happens automatically in
45:      ** MIO_DeleteAll() */
46:     cfg = calloc(1,sizeof(tMIO_M66_Cfg));
47:     /* Store pointer to private data for later access */
48:     MIO_Slots[Slot].PrivatePtr = cfg;
49:     /* Reset Hardware to initial state */
50:     MIO_M66_Reset (Slot);
51:     return 0;
52: }
```

Listing 10.3: Needed functions for custom MIO drivers. (Part2)

```

53: /* Example for the description function */
54: // static char* MIO_M35_GetDescr(int Slot, int Type, int SubType)
55: //
56: // switch (SubType) {
57: //     case 0x00:
58: //         case 0x04: return "16 analog voltage inputs, single-ended";
59: //         case 0x01:
60: //             case 0x05: return "16 analog current inputs, single-ended";
61: //             case 0x02: return "8 analog voltage inputs, differential";
62: //             case 0x03: return "8 analog current inputs, differential";
63: //             default: return "16 /8 analog inputs";
64: // }
65: }
66:
67:
68: /* M66-ModuleDescriptor */
69: static tMIO_ModuleDesc MIO_M66_Desc = {
70:     MIO_MT_M66,           /* ModuleNumber */
71:     0,                   /* 1 if module needs A24 address space, otherwise 0 */
72:     "Binary Inputs/Outputs, 32 channels", /* Description of the Module */
73:     MIO_M66_Config,      /* Config Function */
74:     MIO_M66_Reset,       /* Reset Function */
75:     NULL,                /* Delete Function */
76:     NULL                 /* Description Function */
77: };
78:
79:
80: /* MIO_M66_RegisterDriver() - Registers the Driver within CarMaker */
81: int MIO_M66_RegisterDriver(void)
82: {
83:     return MIO_RegisterDriver(&MIO_M66_Desc);
84: }
85:
86: /* Following functions depend on module type, so only 1 example is shown below */
87: /*
88: ** MIO_M66_Read - Call in IO_In()
89: **
90: ** Get one input channel
91: ** Return Value:
92: **     Bit 0 contains the current state of the channel
93: **     Bit 1 contains a "1" if a rising edge occurred since the last reading
94: **     Bit 2 contains a "1" if a falling edge occurred since the last reading
95: */
96: unsigned int MIO_M66_Read (int Slot, int Channel)
97: {
98:     volatile tMIO_M66_A08 *IO = MIO_Slots[Slot].IO;
99:     return (IO->Channel[Channel] >> 3) & 0x07;
100: }
```

Description of the Listing

- Line 1ff
Include the needed header files.
- Line 8:
The module type. This must be the same as the value from the ID EEPROM of the M-Module.
- Line18ff
This is the description of the hardware of the M-Module. This description makes it easier to handle accesses to the hardware, because no handling of pointers is needed.

- Line 24ff
The reset function should do anything that is needed to set the module to an initial state. The function is automatically executed when calling MIO_ResetModules from CarMaker and therefore must not be called manually from outside the custom driver. If no reset is needed, the function can be left empty.
- Line 30ff
The delete function should free all the memory that is allocated for the module but not the PrivatePointer if this. The private pointer is freed automatically from CarMaker.
- Line 37ff
The config function must be called for each module that is supported by the driver and is used by the RT-executable. This function must register the module, allocate the memory for this module and call the reset function for this module.
Hint: the memory is should not be allocated for the driver but for exactly the module that the config function is called for!
- Line 55ff
The description function can be used if it is not possible to describe the module in one string, e.g if the module has several subtypes with a different number of channels.
- Line 68ff
The module descriptor contains the information that the CarMaker must know about the driver.
 - Module Number
This must be the same as the value from the ID EEPROM of the M-Module.
 - A24 Access Needed
If A24 access is needed because the module does not support that A08 address space set this value to 1. Otherwise it should be set to 0.
 - Description of the Module
Here you can set a text string that describes the module. If the module description is somewhat more complex use the description function instead of the description string. In this case the description string must be `NULL`.
 - Config Function / Reset Function / Delete Function / Description Function
This contains the function pointers of the corresponding functions. The function pointers are used to call the functions automatically from CarMaker. Set the function pointers to `NULL` if the functions do not exist.
If a description function is used, the description string must be set to `NULL`!
- Line 80ff
Registering the driver tells CarMaker that the driver for the module type is available. From the time the driver is registered, CarMaker can use the functions of the module and the information about it.
- Line 88ff
Implement the functions for the modules here.
The CarMaker internal variable `MIO_Slots[Slot].IO` contains the modules base address in the A08 address space.
The CarMaker internal variable `MIO_Slots[Slot].IO24` contains the modules base address in the A24 address space.

An example MIO driver can be found in the CarMaker's installation directory in the subdirectory `/Examples/MIO/`

Integrating the Driver into IO.c in CarMaker

The custom driver must be implemented in IO.c as shown below

Listing 10.4: Custom Driver Integration

```
1: int IO_Init (void) /* = IO_Start() in RTMaker */
2: {
3:     [...]
4:     if (MIO_Init(NULL) < 0) return RT_Failure;
5:     /* Register Custom or Replacement Driver here */
6:     MIO_M66_RegisterDriver();
7:     /* Call the correct config function for each slot (Examples following) */
8:     MIO_M441_Config(Slot_PWM);
9:     MIO_M36_Config(Slot_AnalogIn);
10:    MIO_M405_Config(Slot_LIN);
11:    [...]
12:    MIO_M66_Config(Slot_M66);
13:    [...]
14:    return RT_Success;
15: }
16:
17:
18: void IO_In (void)
19: {
20:     /* Call input functions here */
21: }
22:
23:
24: void IO_Out (void)
25: {
26:     /* Call output functions here */
27: }
```

The MIO drivers of all modules which are supported by CarMaker by default, are registered during first call of the modules `MIO_Mxxx_Config()` function.

The custom register function (e.g. `MIO_M66_RegisterDriver()`, see Line 6 above) must be called after `MIO_Init()` but before calling `MIO_Mxxx_Config()` (e.g. `MIO_M66_Config()`, see Line 12 above).

It is possible to overwrite the CarMakers default drivers easily by re-registering a driver for the same module type after `MIO_Init()`.

10.3 Administrative Functions

10.3.1 Initialization and M-Module Configuration

MIO_Init()

```
int MIO_Init(void *IO_Space)
```

Description

This function initializes the MIO module. It has to be called once at program start and prior to any other MIO function. The registration table of M-Modules will be cleared, the configuration of all modules will get lost.

Parameters

- `IO_Space`

This parameter must be `NULL`. If the initialization was successful, `MIO_Init()` returns 0, otherwise it returns an error code (< 0). If the initialization was successful, `MIO_Init()` returns a file handle to the MIO driver (≥ 0), otherwise it returns an error code (< 0).

Example

```
if (MIO_Init(NULL) < 0) return -1;
```

MIO_GetModuleType()

```
int MIO_GetModuleType(int Slot)
```

Description

This function returns the module type (M-Module number) of the M-Module in the given Slot.

Return Value

- If an identifiable M-Module is found the M-Module number is returned
- No M-Module located at slot# Slot: 0, `MIO_MT_EMPTY`
- Unknown module found: -1, `MIO_MT_UNKNOWN`
- No info for this slot available: -2, `MIO_MT_INVALID`:

MIO_RegisterModule()

```
int MIO_RegisterModule(int Slot, int Type)
```

Description

Before a M-Module can be used, it has to be registered first, by calling the function `MIO_RegisterModule()`.



The registration of M-Modules is done automatically by the `MIO_Mxx_Config()` function. You should not call this function directly. However, in some cases – e.g. if you implement support for an unsupported M-Module on your own – this function might be useful.

`Slot` gives the M-Module slot where the module is mounted, `Type` specifies the module type (M-Module number).

The function `MIO_RegisterModule()` scans the given M-Module slot and checks by reading the contents of the Id EEPROM, if the module type matches `Type`. If `Slot` is out of range, if there is a different or even no M-Module mounted at `Slot`, then an error code is returned. If `Type` matches the type of the mounted M-Module at `Slot` – or if the module has no Id EEPROM – then `MIO_RegisterModule()` returns 0.

MIO_ResetModules()

```
void MIO_ResetModules(void)
```

Description

This function scans all configured M-Modules and calls the specific `MIO_Mxx_Reset()` functions, in order to (re-)initialize all configured M-Modules.

MIO_DeleteAll()

```
void MIO_DeleteAll(void)
```

Description

At the end of your application, before exiting and after the last access to any M-Module, this function should be called. `MIO_DeleteAll()` marks all configured M-Modules as unused and frees all occupied resources. Before using any M-Module again, the associated `MIO_Mxx_Config()` function has to be called again.

10.3.2 MIO and M-Module Information

MIO_GetVersion()

```
const char *MIO_GetVersion(void)
```

Description

This function returns a string with some information of the MIO software module. The string looks like

```
mio.o 1.3.0 2011-10-11
```

and contains the name of the MIO software module (`mio.o`), the version of the module and the compilation date. This identification is used to check, if the header file which you use when compiling your application, matches the used MIO software module.

MIO_ModuleInfo()

```
char *MIO_ModuleInfo(int Slot, MIO_Type ModuleType)
```

Description

Returns a description for a M-Module. If `Slot` has a valid value – which means that there is a module found at `Slot` – then [MIO_ModuleInfo\(\)](#) returns a description of this module. If `Slot` has an invalid value (e.g. -1), then a description for a M-Module of type `ModuleType` is returned.

MIO_GetBaseAddress()

```
void *MIO_GetBaseAddress(int Slot)
```

Description

Returns the *A8* base address for a M-Module. `Slot` has to be a valid value in the range of 0 .. `MIO_MAX_SLOTS-1` (191), otherwise the result might be unpredictable or even crash the application.

Return Value

- If `Slot` addresses a not existing Slot `NULL` is returned
- If there is no M-Module installed in the slot `NULL` is returned
- If a module is found in the slot the *A8* base address of the module is returned

MIO_GetBaseAddress24()

```
void *MIO_GetBaseAddress(int Slot)
```

Description

Returns the *A24* base address for a M-Module. `Slot` has to be a valid value in the range of 0 .. `MIO_MAX_SLOTS-1` (191), otherwise the result might be unpredictable or even crash the application.

Return Values

- If `Slot` addresses a not existing Slot `NULL` is returned
- If there is no M-Module installed in the slot `NULL` is returned
- If the carrier board does not support *A24* address range `NULL` is returned
- If the carrier board supports the *A24* address range and a module is found in the slot the *A24* base address of the module is returned

MIO_ModuleShow()

```
void MIO_ModuleShow(void)
```

Description

This function prints configuration information for all registered M-Modules to *stdout*. Please use this function for debugging purpose, only.

10.3.3 Error Handling

MIO_ErrorGet()

```
int MIO_ErrorGet(tMIO_ErrCode *ErrCode, const char **ErrMsg)
```

Description

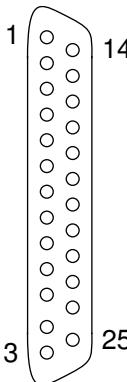
Obsolete

[MIO_ErrorGet\(\)](#) is only a dummy function for compatibility with older CarMaker revisions.

10.4 M-Module Function Description

10.4.1 M27: Binary Output (16 channels)

Connector assignment



Pin	Signal	Pin	Signal
1		14	Out_1
2	Out_0	15	Out_3
3	Out_6	16	Out_5
4	Out_4	17	Out_7
5	Out_10	18	Out_9
6	Out_8	19	Out_11
7	Out_14	20	Out_13
8	Out_12	21	Out_15
9	I+24V	22	I+24V
10	I+24V	23	GND
11	GND	24	GND
12	GND	25	GND
13	GND		

Function Overview

Initialization and Configuration

- [MIO_M27_Config\(\)](#)

Set Output Signals

- [MIO_M27_Set\(\)](#)

MIO_M27_Config()

```
int MIO_M27_Config(int Slot)
```

Description

By calling the function [MIO_M27_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M27 is specified. This function must be called one time, before any attempted access to the module.

The initial state of all output channels is undefined and determined by the given module hardware. If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

MIO_M27_Set()

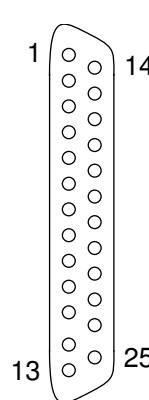
```
void MIO_M27_Set(int Slot, int Value)
```

Description

Sets the value of all binary outputs (e.g. sets the relay positions). The bit position indicates the channel number. For example, channel 0 would be set with the lowest bit, channel 1 the second lowest bit, etc.

10.4.2 M28: Binary Output (16 channels)

Connector assignment



Pin	Signal	Pin	Signal
1	Out_0	14	Out_1
2	Out_2	15	Out_3
3	Out_4	16	Out_5
4	Out_6	17	Out_7
5	Out_8	18	Out_9
6	Out_10	19	Out_11
7	Out_12	20	Out_13
8	Out_14	21	Out_15
9	GND	22	GND
10	GND	23	I+24V
11	I+24V	24	I+24V
12	I+24V	25	I+24V
13	I+24V		

Function Overview

Initialization and Configuration

- [MIO_M28_Config\(\)](#)

Set Output Signals

- [MIO_M28_Set\(\)](#)

MIO_M28_Config()

```
int MIO_M28_Config(int Slot)
```

Description

(see function description for [MIO_M27_Config\(\)](#))

MIO_M28_Set()

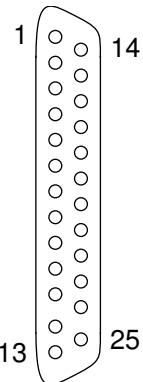
```
void MIO_M28_Set(int Slot, int Value)
```

Description

(see function description for [MIO_M27_Set\(\)](#))

10.4.3 M31: Binary Inputs (16 channels)

Connector assignment



Pin	Signal	Pin	Signal
1	In_0	14	In_1
2	In_2	15	In_3
3	In_4	16	In_5
4	In_6	17	In_7
5	In_8	18	In_9
6	In_10	19	In_11
7	In_12	20	In_13
8	In_14	21	In_15
9	I-GND	22	I-GND
10	I-GND	23	I-GND
11	I-GND	24	I-GND
12	I-GND	25	I-GND
13	I-GND		

Function Overview

Initialization and Configuration

- [MIO_M31_Config\(\)](#)

Get Input Signals

- [MIO_M31_Read\(\)](#)

MIO_M31_Config()

```
int MIO_M31_Config(int Slot)
```

Description

By calling the function [MIO_M31_Config\(\)](#), the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M31, is specified. This function must be called one time, before any attempted access to the module.

The module M31 is software-compatible to the module M32.

Runtime Functions

MIO_M31_Read()

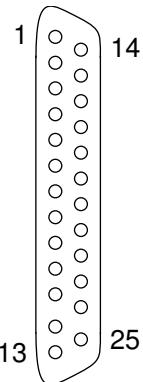
```
int MIO_M31_Read(int Slot)
```

Description

Reads the values of the 16 input ports as a 16 bit value. Bit n corresponds to input n.

10.4.4 M32: Binary Inputs (16 channels)

Connector assignment



Pin	Signal	Pin	Signal
1	In_0	14	In_1
2	In_2	15	In_3
3	In_4	16	In_5
4	In_6	17	In_7
5	In_8	18	In_9
6	In_10	19	In_11
7	In_12	20	In_13
8	In_14	21	In_15
9	I-24V	22	I-24V
10	I-24V	23	I-24V
11	I-24V	24	I-24V
12	I-24V	25	I-24V
13	I-24V		

Function Overview

Initialization and Configuration

- [MIO_M32_Config\(\)](#)

Get Input Signals

- [MIO_M32_Read\(\)](#)

MIO_M32_Config()

```
int MIO_M32_Config(int Slot)
```

Description

(see function description for [MIO_M31_Config\(\)](#))

MIO_M32_Read()

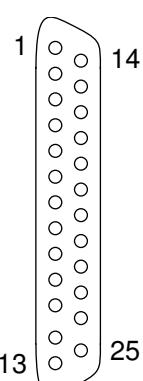
```
int MIO_M32_Read(int Slot)
```

Description

(see function description for [MIO_M31_Read\(\)](#))

10.4.5 M35 / M35N: Analog Inputs (16/8 channels)

Connector assignment M35N00 / M35N01 (16 single-ended inputs)



Pin	Signal	Pin	Signal
1	In_0	14	In_8
2	In_1	15	In_9
3	In_2	16	In_10
4	In_3	17	In_11
5	In_4	18	In_12
6	In_5	19	In_13
7	In_6	20	In_14
8	In_7	21	In_15
9	BI	22	Trigger
10	GND	23	reserved
11	GND	24	reserved
12	GND	25	reserved
13	GND		

Connector assignment M35N02 / M35N03 (8 differential inputs)

Pin	Signal	Pin	Signal
1	In_0	14	In_0-
2	In_1	15	In_1-
3	In_2	16	In_2-
4	In_3	17	In_3-
5	In_4	18	In_4-
6	In_5	19	In_5-
7	In_6	20	In_6-
8	In_7	21	In_7-
9	BI	22	Trigger
10	GND	23	reserved
11	GND	24	reserved
12	GND	25	reserved
13	GND		

M35N Compatibility Issues

Some versions of the M35N module do not operate according to the specification, due to different firmware bugs:

- M35N modules of revision of 5.1 or lower do not start a conversion when writing at address 0x02. As a consequence, data acquisition with the function `MIO_M35_ReadV()` does not work properly: the data which is read from the module is invalid.
- M35N modules of revision of 5.2, start a conversion when writing at address 0x02. But, when reading data from address 0x00 right after starting a conversion (by writing to address 0x02), the result is not from the conversion which was started right before.

Instead, the data fetched derives from the conversion before last. When reading data with the function `MIO_M35_ReadV()`, this leads to a permutation of the channels: e.g. if data should be obtained from channels 0, 1, 2 and 4 (`ChMask = 0x17`) and stored in the array `Values[4]`, then the data of channel 0 will be stored in `Values[1]`, channel 1 in `Values[2]`, channel 2 in `Values[3]` and channel 4 in `Values[0]`.

- All versions of M35N modules do not work properly on A201 carrier boards of the first generation. When reading inputs, the values toggle between 0 and the real value. Upgrade to the newer A201S carrier board is recommended.

If you are using a M35N module of revision 5.2 or earlier, please contact IPG for a firmware update.

Function Overview

Initialization and Configuration

- `MIO_M35_Config()`

Get Input Signals

- `MIO_M35_Read()`
- `MIO_M35_Read_bp()`
- `MIO_M35_ReadV()`
- `MIO_M35_ReadV2()`
- `MIO_M35_ReadV3()`
- `MIO_M35_ReadV4()`

MIO_M35_Config()

```
int MIO_M35_Config(int Slot)
```

Description

By calling the function [MIO_M35_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M35 is specified. This function must be called one time, before any attempted access to the module.

The modules M34, M35 and M35N are software-compatible.

Initial state of input channels:

Amplification factor for an input signal = 1

If the module cannot be configured-initialized, the function returns an error code, otherwise 0 is returned.

MIO_M35_Read()

```
int MIO_M35_Read(int Slot, int Ch, int Gain)
```

Description

The function reads the measured value of an input port in unipolar mode (measuring range, 0 V .. 10 V with amplification factor of 1).

Time requirement on successive calls, without change on channel or gain:

M-Module	Conversion time
M5	10 us
M34	10 us
M35	14 us
M35N	7.8 us

If channel or gain are changed by the preceding call then the double conversion time is needed. With the help of the function [MIO_M35_ReadV\(\)](#) several channels can be queried more efficiently (approximately simple transformation time per channel).

Slot and Ch indicate the card location and the channel number.

The parameter Gain determines the amplification factor:

- Gain = 0x00 for an amplification factor of 1
- Gain = 0x01 for an amplification factor of 2
- Gain = 0x02 for an amplification factor of 4
- Gain = 0x03 for an amplification factor of 8

Return Value: The 14 bits measured value (with M34: 12 bits), left justified in a 16 bit integer value.

MIO_M35_Read_bp()

```
int MIO_M35_Read_bp(int Slot, int Ch, int Gain)
```

Description

Same function as [MIO_M35_Read\(\)](#), except that the read is done in bipolar mode.

Return Value: The 14 bits measured value (with M34: 12 bits), left justified in a 16 bit integer value.

MIO_M35_ReadV()

```
void MIO_M35_ReadV(int Slot, unsigned ChMask, int GainMode, int *Values)
```

Description

Similar to [MIO_M35_Read\(\)](#), except that several channels can be read simultaneously. The function is very efficient when many channels are being read.

`ChMask` is a bit field (bit 0..15 used), which specifies the channels which can be queried. If bit n has the value 1, the channel n is queried.

Example

```
int s, ChMask, v[8];
s = 0;
ChMask = 0x00ff;
MIO_M35_ReadV(s, ChMask, 0, v);
```

Causes the inquiry of 8 channels (0 .. 7).

Bits 0 .. 1 of `GainMode` specify the amplification factor, as explained for parameter `Gain` of function [MIO_M35_Read\(\)](#).

If bit 2 of `GainMode` is set to 1, then all values are read in bipolar mode. Otherwise, unipolar mode is used.

Example

```
MIO_M35_ReadV(s, ChMask, 0, v); /* unipolar, amplification factor 1 */
MIO_M35_ReadV(s, ChMask, 1, v); /* unipolar, amplification factor 2 */
MIO_M35_ReadV(s, ChMask, 4, v); /* bipolar, amplification factor 1 */
MIO_M35_ReadV(s, ChMask, 5, v); /* bipolar, amplification factor 2 */
```

`Values` is a pointer to the memory location of an integer array, where the values (concise, beginning with the lowest channel number) will be stored. The referenced array must be large enough, to hold at least as many integer values, as the number of selected channels.

MIO_M35_ReadV2()

```
void MIO_M35_ReadV2(int Slot1, unsigned ChMask1, int GainMode1, int *Values1,
                     int Slot2, unsigned ChMask2, int GainMode2, int *Values2)
```

Description

Similar to [MIO_M35_ReadV\(\)](#), except that two M35N modules (also M35 or M34) are read simultaneously. This results in a much better performance, than scanning the two modules separately (see [Performance aspects](#)).

MIO_M35_ReadV3()

```
void MIO_M35_ReadV3(int Slot1, unsigned ChMask1, int GainMode1, int *Values1,
                     int Slot2, unsigned ChMask2, int GainMode2, int *Values2,
                     int Slot3, unsigned ChMask3, int GainMode3, int *Values3)
```

Description

Similar to [MIO_M35_ReadV\(\)](#), except that three M35N modules (also M35 or M34) are read simultaneously. This results in a much better performance, than scanning the three modules separately (s. [Performance aspects](#)).

MIO_M35_ReadV4()

```
void MIO_M35_ReadV4(int Slot1, unsigned ChMask1, int GainMode1, int *Values1,
                     int Slot2, unsigned ChMask2, int GainMode2, int *Values2,
                     int Slot3, unsigned ChMask3, int GainMode3, int *Values3,
                     int Slot4, unsigned ChMask4, int GainMode4, int *Values4)
```

Description

Similar to [MIO_M35_ReadV\(\)](#), except that four M35N modules (also M35 or M34) are read simultaneously. This results in a much better performance, than scanning the four modules separately (s. [Performance aspects](#)).

Performance aspects

Due to the electrical layout and the long conversion times of the analog input modules M35N, M35 and M34, reading multiple input channels one-by-one with [MIO_M35_Read\(\)](#), can be very time consuming. In the worst case, when reading all 16 input channels successively with [MIO_M35_Read\(\)](#), the double conversion time is needed (e.g. $32 \times 10 \text{ us} = 320 \text{ us}$ with M34 module). Since the channel is changed on every call, a duty cycle of one extra conversion is required for each channel.

As soon as two or more channels are to be read, it is highly recommended to use the function [MIO_M35_ReadV\(\)](#), which speeds up the access to nearly the single conversion time for each channel (e.g. $16 \times 10 \text{ us} = 160 \text{ us}$ when reading all 16 channels with an M34 module).

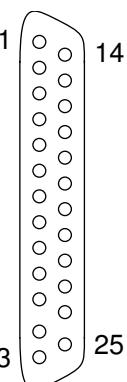
However, when reading two or more M35N, M35, or M34 modules simultaneously (even if mixed together), the access times can still be tuned when using the optimized functions [MIO_M35_ReadV2\(\)](#), [MIO_M35_ReadV3\(\)](#), or [MIO_M35_ReadV4\(\)](#). The following table gives an impression of how the access time can be speed up with up to 4 M34 modules, when reading all input channels (16 per module):

# of modules	MIO_M35_ReadV4	MIO_M35_ReadV3	MIO_M35_ReadV2	MIO_M35_ReadV	MIO_M35_Read
1	-	-	-	160 us	320 us
2	-	-	190 us	320 us	640 us
3	-	215 us	-	480 us	960 us
4	240 us	-	380 us	640 us	1280 us

Using M35, M35N modules results in similar improvement.

10.4.6 M36N: 16 Bits Analog Inputs (16/8 channels)

Connector assignment M36N00 / M36N02 (16 single-ended inputs)



Pin	Signal	Pin	Signal
1	In_0	14	In_8
2	In_1	15	In_9
3	In_2	16	In_10
4	In_3	17	In_11
5	In_4	18	In_12
6	In_5	19	In_13
7	In_6	20	In_14
8	In_7	21	In_15
9	BI	22	Trigger
10	GND	23	-
11	GND	24	-
12	GND	25	-
13	GND		

Connector assignment M36N01 / M36N03 (8 differential inputs)

Pin	Signal	Pin	Signal
1	In_0	14	In_0-
2	In_1	15	In_1-
3	In_2	16	In_2-
4	In_3	17	In_3-
5	In_4	18	In_4-
6	In_5	19	In_5-
7	In_6	20	In_6-
8	In_7	21	In_7-
9	BI	22	Trigger
10	GND	23	-
11	GND	24	-
12	GND	25	-
13	GND		



M36N Compatibility Issues

Due to firmware bugs, only modules of revision 2.1 or higher are supported.

If you are using a M36N module of revision 2.0 or earlier, please contact IPG for a firmware update.

Function Overview

Initialization and Configuration

- [MIO_M36_Config\(\)](#)

- [MIO_M36_SetMaxCh\(\)](#)
- [MIO_M36_Setup\(\)](#)
- [MIO_M36_SetupV\(\)](#)

Get Input Signals

- [MIO_M36_GetRes\(\)](#)
- [MIO_M36_Read\(\)](#)
- [MIO_M36_ReadV\(\)](#)
- [MIO_M36_WaitForNewData\(\)](#)

MIO_M36_Config()

```
int MIO_M36_Config(int Slot)
```

Description

By calling the function [MIO_M36_Config\(\)](#), the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M36N is specified. This function must be called one time before any attempted access to the module.

The initial state of the input channels is undefined. To start the conversion with well defined channel configurations, you need to setup the input channels, first.



By default it is assumed, that the module has 16 single ended inputs. If your M36N module is one of the models with 8 differential input channels, you will need to adjust the number of available channels by calling the function [MIO_M36_SetMaxCh\(\)](#). Otherwise, the results might be unpredictable.

If the module cannot be configured-initialized, the function returns an error code. Otherwise, 0 is returned.

The function resets the input ports of the module to the initial state:

Conversion is enabled on all channels

Application factor for all input signals is 1.

All channels operate in unipolar mode

MIO_M36_SetMaxCh()

```
void MIO_M36_SetMaxCh(int Slot, int MaxCh)
```

Description

Sets the number of available input channels. Depending on the type of M36N module, this function needs to be called after [MIO_M36_Config\(\)](#), but before [MIO_M36_Setup\(\)](#), to tell the MIO module the number of available channels.

Slot indicates the card location.

MaxCh gives the number of channels on the module. The default number of channels is 16, which is suitable for all types of M36N modules with single ended inputs.

MIO_M36_Setup()

```
int MIO_M36_Setup(int Slot, int Ch, int GainMode)
```

Description

Sets the measuring range of a channel.

Slot and Ch indicate the card location and the channel number.

`GainMode` indicates the desired measuring mode and range of the selected channel. Bit 7 switches between unipolar (0) and bipolar (1: 0x80) mode. Bits 4 .. 6 select the measuring range:

GainMode	Measuring Range	
0x00	0V ..	10V unipolar
0x10	0V ..	5V unipolar
0x20	0V ..	2.5V unipolar
0x30	0V ..	1.25V unipolar ^a
0x40	0V ..	625mV unipolar
0x80	-10V ..	+10V bipolar
0x90	-5V ..	+5V bipolar
0xa0	-2.5V ..	+2.5V bipolar
0xb0	-1.25V ..	+1.25V bipolar ^b
0xc0	-625mV ..	+625mV bipolar

a. 0 .. 20 mA with current inputs

b. -20 .. 20 mA with current inputs

If the channel number `Ch` is out of range, then the function will return -1. A 0 is returned upon success.

MIO_M36_SetupV()

```
int MIO_M36_SetupV(int Slot, unsigned ChMask, int *GainModes)
```

Description

Similar to [MIO_M36_Setup\(\)](#), except the measuring range can be set for several channels.

`Slot` indicates the card location.

`ChMask` is a bit mask which indicates the channels which should be configured. If bit n has the value 1 the measuring range of channel n will be configured.

`GainModes` is a pointer to an array where the measuring ranges of the channels (beginning with channel 0) are specified. The size of the field must correspond to at least the number of existing channels on the M-Module (16 channels for modules with single ended inputs, 8 channels for modules with differential inputs).

The n-th member of the `GainModes` array gives the configuration for the n-th channel, if the n-th bit in `ChMask` is 1. For each control word, bit 7 switches between unipolar (0) and bipolar (1: 0x80) mode and bits 4 .. 6 select the measuring range:

GainMode	Measuring Range	
0x00	0V ..	10V unipolar
0x10	0V ..	5V unipolar
0x20	0V ..	2.5V unipolar
0x30	0V ..	1.25V unipolar ^a
0x40	0V ..	625mV unipolar
0x80	-10V ..	+10V bipolar
0x90	-5V ..	+5V bipolar

GainMode	Measuring Range
0xa0	--2.5V .. +2.5V bipolar
0xb0	-1.25V .. +1.25V bipolar ^b
0xc0	-625mV .. +625mV bipolar

- a. 0 .. 20 mA with current inputs
b. -20 .. 20 mA with current inputs

The function returns the number of activated channels.

MIO_M36_GetRes()

```
double MIO_M36_GetRes(int Slot, int Ch)
```

Description

Returns the bit resolution for the given channel number in Volt. It can be used to convert the binary input values to voltage values.

Slot and Ch indicate the card location and the channel number.



This function only tells suitable resolutions for the voltage version of the M36N module. For the current version of the module, the resolution is 0.61 µA in bipolar mode and 0.305 µA in unipolar mode. But, since the amplification factor needs to be always 8 for current measurement, you can use the resolution returned by this function and multiply it with 0.016 to receive a resolution in Ampere.

MIO_M36_Read()

```
int MIO_M36_Read(int Slot, int Ch)
```

Description

The function reads the measured value of an input port. The measurement mode and range depend on the settings for the given channel Ch.

Since the M36N module continuously converts all activated input channels, the measuring value of the last conversion on channel Ch is returned. Therefore, the execution time for this function is very low (about 1µs). But, since the returned value typically originates from the last measuring cycle, the value can be up to 130µs old (depending on the number of activated channels).

Return Value: The 16 bits measured value. To convert this value into Volt or Ampere, refer to function [MIO_M36_GetRes\(\)](#).

MIO_M36_ReadV()

```
void MIO_M36_ReadV(int Slot, int *Values)
```

Description

Similar to [MIO_M36_Read\(\)](#), except that all activated channels are read simultaneously.

The function is very efficient when many channels are being read and should be always preferred to [MIO_M36_Read\(\)](#).

`Values` is an array pointer to a memory location where the values (concise, beginning with the lowest channel number) will be stored.

MIO_M36_WaitForNewData()

```
void MIO_M36_WaitForNewData(int Slot)
```

Description

Waits in a busy loop, until the current activated measuring cycle is finished. Reading data right after a call to this function assures, that the values are newly converted and not older than 130µs (depending on the number of activated channels).

However, this function will never wait more than 200µs.

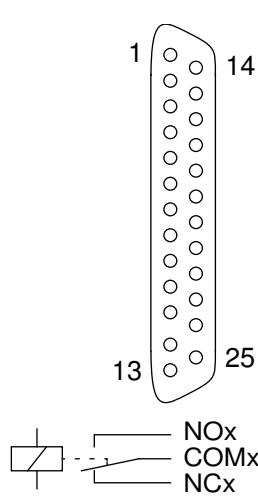


Use this function with care! It is not recommended to use this function in realtime tasks, since it might break realtime conditions.

However, it is ok to call this function during initialization after calling [MIO_M36_Setup\(\)](#), [MIO_M36_SetupV\(\)](#) or [MIO_M36_Setup\(\)](#). This assures that there is no data left from old measuring cycles, which could cause reading invalid values.

10.4.7 M43: Relay Outputs (8 channels)

Connector assignment M43



Pin	Signal	Pin	Signal
1	NO0	14	COM0
2	NC0	15	NO1
3	COM1	16	NC1
4	NO2	17	COM2
5	NC2	18	NO3
6	COM3	19	NC3
7	NO4	20	COM4
8	NC4	21	NO5
9	COM5	22	NC5
10	NO6	23	COM6
11	NC6	24	NO7
12	COM7	25	NC7
13	n.c.		

Function Overview

Initialization and Configuration

- [MIO_M43_Config\(\)](#)

Set Relays

- [MIO_M43_Set\(\)](#)

MIO_M43_Config()

```
int MIO_M43_Config(int Slot)
```

Description

(see function description for [MIO_M27_Config\(\)](#))

MIO_M43_Set()

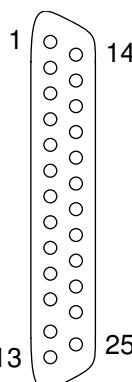
```
void MIO_M43_Set(int Slot, int Value)
```

Description

(see function description for [MIO_M27_Set\(\)](#))

10.4.8 M51: Quadruple CAN Interface

Connector assignment



	Pin	Signal		Pin	Signal
1	1	CAN0_low		14	CAN0_high
	2	CAN0_GND		15	CAN0_VCC
	3			16	
	4	CAN1_low		17	CAN1_high
13	5	CAN1_GND		18	CAN1_VCC
	6			19	
	7	CAN2_low		20	CAN2_high
	8	CAN2_GND		21	CAN2_VCC
	9			22	
	10	CAN3_low		23	CAN3_high
	11	CAN3_GND		24	CAN3_VCC
	12			25	
	13				

Function Overview

Initialization and Configuration

- `MIO_M51_Config()`
- `MIO_M51_SetBufSize()`
- `MIO_M51_SetCommParam()`
- `MIO_M51_SetTxTimeout()`
- `MIO_M51_EnableIds()`
- `MIO_M51_DisableIds()`

Polling Mode Operation

- `MIO_M51_Poll()`

Sequential Send/Receive Mode

- `MIO_M51_Send()`
- `MIO_M51_SendV()`
- `MIO_M51_SendBufClear()`
- `MIO_M51_SendStat()`
- `MIO_M51_Recv()`
- `MIO_M51_RecvBufClear()`
- `MIO_M51_RecvStat()`
- `MIO_M51_TxAbort()`

Mailboxes

- `MIO_M51_MBoxCreate()`
- `MIO_M51_MBoxDelete()`
- `MIO_M51_MBoxRead()`

Additional Functions

- `MIO_M51_SetTimestampFct()`
- `MIO_M51_SetTransMode()`
- `MIO_M51_UnlockTxBuf()`

CAN Trace

- `MIO_M51_TraceInit()`
- `MIO_M51_TraceStop()`

Diagnostics

- `MIO_M51_Status()`
- `MIO_M51_GetRxErrCnt()`
- `MIO_M51_GetTxErrCnt()`
- `MIO_M51_CANInfo()`

Debugging

- `MIO_M51_GetRegister()`
- `MIO_M51_ListAddr()`
- `MIO_M51_DumpPSCC()`

MIO_M51_Config()

```
int MIO_M51_Config(int Slot, int IRQlevel)
```

Description

By calling the function [MIO_M51_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M51 is specified. This function must be called one time, before any attempted access to the module.

`IRQlevel` specifies the priority that the interupts will be handled. The possible values are listed below:

- negative value (usually -1) - `IRQlevel` will be automatically chosen
- zero - if 0 is specified then the module will be configured in polling mode with interrupts disabled (the CAN controllers will be polled in the main task)
- one through 100 - sets the interrupt priority level, with one being the lowest priority and 100 being the highest priority (choose `IRQlevel < 40` for background polling with low priority)
- one through seven - a value in the range 1 .. 7 sets the interrupt priority level, with one being the highest priority and seven being the lowest priority (on older PowerPC based hardware platform, only)



All CAN controllers are set to the default state, listed below:

- Baud rate of 200 kBit/s, single sample point (see [MIO_M51_SetCommParam\(\)](#))
- Send and receive buffer size is 64 messages (see [MIO_M51_SetBufSize\(\)](#))
- No mailboxes are defined (see [MIO_M51_MBoxCreate\(\)](#))
- No message IDs are enabled (see [MIO_M51_EnableIds\(\)](#))
- Transparency mode is turned off (see [MIO_M51_SetTransMode\(\)](#))
- There is no timeout when sending a message (see [MIO_M51_SetTxTimeout\(\)](#))

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

MIO_M51_SetBufSize()

```
void MIO_M51_SetBufSize(int Slot, int Ch, int RxSize, int TxSize)
```

Description

The size of the send and receive buffers can be changed by calling [MIO_M51_SetBufSize\(\)](#).

`Slot` and `Ch` give the location and channel number of the CAN interface.

`RxSize` and `TxSize` indicate the number of messages which the receive and/or send buffers can hold. The following applies to both parameters: minimum buffer size is 2, the maximum size is not limited, and the default value is 64.

Realtime conditions will not be met when resizing the buffer to a larger size. Changing the buffer size to something smaller, however, will meet realtime conditions.

MIO_M51_SetCommParam()

```
int MIO_M51_SetCommParam(int Slot, int Ch, int Baudrate, int SamplePoint, int SynchJumpWidth,  
                         int Sample3)
```

Description

With the function [MIO_M51_SetCommParam\(\)](#) the CAN communication parameters are specified. The values needed for the programming of the CAN interface module SJA1000 are relatively complex, are subject to a number of restrictions and are not particularly meaningful as numerical values. For this reason more general and understandable parameters are used.

`Slot` and `Ch` give the location and channel number of the CAN interface.

`Baudrate` is the data transmission rate, e.g. 250000. Only values that can be derived by integral division from 8 MHz (the oscillator frequency is 16 MHz, divisor ≥ 3) can be used.

`SamplePoint` indicates the sample point (i.e. the point in time within the bounds of the bit period that the bit will be evaluated as (1) high or (0) low) for the value of a bit. The indication takes place in percent, relative to the bit period. Valid values: 0..100, typically: 65.

`SynchJumpWidth` specifies over how many internal clock units the time slot pattern per data bit can be adapted. This makes a resynchronisation possible on the bit stream of the transmitter with a slightly deviating clock rate. Range of values: 1-4, typically: 2.

`Sample3` is a flag that specifies whether the individual bits that are transmitted, are sampled one (`Sample3==0`) or three times (`Sample3==1`). Usually with high data transmission rates only one sample is necessary, but with slower transmission rates the three sample method is recommended.

If the requested baud rate is invalid or impossible, the function returns an error code, otherwise 0 is returned.

If the value of the selected bit period (dependent on the data transmission rate) is not known, [MIO_M51_SetCommParam\(\)](#) determines and uses the configuration with the smallest deviation.

More details are available with the documentation of the SJA1000 controller from Phillips.

MIO_M51_SetTxTimeout()

```
void MIO_M51_SetTxTimeout(int Slot, int Ch, int Timeout)
```

Description

Currently has no effect.

MIO_M51_EnableIds()

```
void MIO_M51_EnableIds(int Slot, int Ch, int StartId, int n)
```

Description

[MIO_M51_EnableIds\(\)](#) enables the receipt of the specified IDs, beginning with `StartId` and ending with `StartId + (n-1)`.

`Slot` und `Ch` give the location and channel number of the CAN interface.

The acceptance mask of the interface module is configured accordingly. See [MIO_M51_SetTransMode\(\)](#).



This function only affects the reception of standard CAN messages. It does not enable the reception of any extended CAN message.

The reception of extended CAN messages can only be enabled by calling the function [MIO_M51_SetTransMode\(\)](#), which disables both, the hardware and the software acceptance filter.

MIO_M51_DisableIds()

```
void MIO_M51_DisableIds(int Slot, int Ch, int StartId, int n)
```

Description

[MIO_M51_DisableIds\(\)](#) removes *n* IDs from the list of accepted IDs, beginning with *StartId* and ending with *StartId + n*. This does not influence messages that are already contained in the receive buffer.

Slot und *Ch* give the location and channel number of the CAN interface.

The acceptance mask of the interface module is adapted automatically in order to keep the CPU load to a minimum.



This function should only be used when only standard CAN messages are used. As a side effect, this function disables the reception of most extended CAN messages.

For reception of extended CAN messages, the hardware and the software acceptance filter should not be used. Use the function [MIO_M51_SetTransMode\(\)](#) instead, to enable or disable the reception of CAN messages generally.

MIO_M51_Poll()

```
int MIO_M51_Poll(void)
```

Description

This function is needed, if CAN interfaces are configured for Polling mode (see [MIO_M51_Config\(\)](#)). It has to be called every millisecond.

[MIO_M51_Poll\(\)](#) checks all registered M51 modules whether Polling mode is enabled or not. The CAN interfaces of the module are then checked for received CAN messages. All available messages will be copied out of the CAN controllers into the Rx buffer, ready to be read out by the functions [MIO_M51_Recv\(\)](#). [MIO_M51_Poll\(\)](#) will also try to send as many messages as possible available in the Tx buffer, where they are stored by the functions [MIO_M51_Send\(\)](#) and [MIO_M51_SendV\(\)](#).



Currently, there is a limitation when CAN controllers are used in Polling mode: The SJA1000 CAN controllers on the M51 module only have a Rx FIFO on-chip, but no Tx FIFO. As a consequence, in most cases only one CAN message can be sent with each call of [MIO_M51_Poll\(\)](#). Even if you would call [MIO_M51_Poll\(\)](#) several times per cycle, you could not be sure, that there will be no CAN messages left unsent in the Tx buffer. The Tx buffer may overflow, even if there is not much CAN traffic.

MIO_M51_Send()

```
int MIO_M51_Send(int Slot, int Ch, CAN_Msg *Msg)
```

Description

The indicated message is sent, i.e. written into the send buffer working as FIFO. The function works non-blocking, i.e. it returns in each case immediately. If the send buffer is full, the message is ignored and the error code -1 is returned.

Slot und Ch give the location and channel number of the CAN interface.

Msg is the address of a data structure CAN_Msg (see Listing 10.7), which contains the message to be sent.

The function returns 0 on success, and -1 otherwise.

Example

Listing 10.5: Example for [MIO_M51_Send\(\)](#)

```
1:  {
2:      CAN_Msg m;
3:
4:      m.MsgId    = 123;
5:      m.FrameFmt = 0;
6:      m.RTR      = 0;
7:      m.FrameLen = 2;
8:      m.Data[0]   = 0x18;
9:      m.Data[1]   = 0xFF;
10:     MIO_M51_Send(2, 0, &m);
11: }
```

MIO_M51_SendV()

```
int MIO_M51_SendV(int Slot, int Ch, int MsgId, void *Data, int DataLen)
```

Description

[MIO_M51_SendV\(\)](#) is a variant of the function [MIO_M51_Send\(\)](#) with same functionality, however in this case the components of the CAN Message are specified as separate parameters:

- the message Id is specified in MsgId,
- the pointer to the data block is specified in Data
- the number of data bytes is specified in DataLen (0 .. 8)

See the description of [MIO_M51_Send\(\)](#).

Example

Listing 10.6: Example for [MIO_M51_SendV\(\)](#)

```
1:  {
2:      static char Data1[] = {0x07, 0x00, 0x00, 0x00};
3:      static char Data2[] = {0x07, 0x00, 0xff, 0xff};
4:
5:      MIO_M51_SendV(2, 0, 0x400, Data1, 4);
6:      MIO_M51_SendV(2, 0, 0x401, Data2, 4)
7: }
```

MIO_M51_SendBufClear()

```
void MIO_M51_SendBufClear(int Slot, int Ch)
```

Description

The function empties the send buffer. A running transmission procedure (i.e. a message which is being evaluated by the interface module) will not be affected.

Slot and Ch give the location and channel number of the CAN interface.

MIO_M51_SendStat()

```
int MIO_M51_SendStat(int Slot, int Ch)
```

Description

The function returns the number of messages that have not yet been sent from the send buffer. The function can be used to recognize a possible overflow of the send buffer or to measure the time requirement to empty the send buffer.

Slot and Ch give the location and channel number of the CAN interface.

MIO_M51_Recv()

```
int MIO_M51_Recv(int Slot, int Ch, CAN_Msg *Msg)
```

Description

Reads the next Message from the receive buffer. The function works non-blocking, i.e., it returns immediately in each case. If the receive buffer is empty, the function returns the error code (-1). On success, the return value is 0.

Slot and Ch give the location and channel number of the CAN interface.

The received message is returned in *Msg. The data structure CAN_Msg has the following structure:

Listing 10.7: The CAN_Msg struct

```
1:  typedef struct {
2:      unsigned MsgId;
3:      unsigned char FrameFmt :1;
4:      unsigned char RTR :1;
5:      unsigned char FrameLen :4;
6:      unsigned char Data[8];
7:  } CAN_Msg;
```

In order to receive messages with certain ids the id must be enabled by calling the function [MIO_M51_EnableIds\(\)](#).

MIO_M51_RecvBufClear()

```
void MIO_M51_RecvBufClear(int Slot, int Ch)
```

Description

The function empties the receive buffer. The contents of the mailboxes are not affected.

Slot and Ch give the location and channel number of the CAN interface.

MIO_M51_RecvStat()

```
int MIO_M51_RecvStat(int Slot, int Ch)
```

Description

This status function returns the number of messages in the receive buffer.

Slot and Ch give the location and channel number of the CAN interface.

MIO_M51_TxAbort()

```
int MIO_M51_TxAbort(int Slot, int Ch)
```

Description

This status function returns the number of messages in the receive buffer.

Slot and Ch give the location and channel number of the CAN interface.

MIO_M51_MBoxCreate()

```
int MIO_M51_MBoxCreate(int Slot, int Ch, int Id)
```

Description

The function [MIO_M51_MBoxCreate\(\)](#) adds a mailbox to the message ID specified by the argument Id, assuming one does not already exist. The receipt of the messages that have the specified ID is activated with this call and a separate call to the function [MIO_M51_EnableIds\(\)](#) is not necessary.

Each message received will increment a counter and set a time stamp. Normally a default time stamp is set to the system time by using the function [SysGetUTime\(\)](#).

Slot and Ch give the location and channel number of the CAN interface.

The parameter Id gives the message ID for the mailbox.

The use of mailboxes can be an alternative to the functions [MIO_M51_Recv\(\)](#). Instead of reading the messages from the receive ring buffer they are read from the mailbox with the corresponding identifier.

Using the software implemented mailboxes, it is possible to simulate a full CAN implementation, which is usually done with on-chip hardware. Full CAN has an acceptance filter that is used to filter out all messages that are not desired. The M51 module CAN controllers have filters but they would not fulfill full CAN criteria since the filters are not perfect and allow some of the undesired messages to pass through. By using mailboxes it is possible to filter all undesired messages and therefore have what could be considered full CAN functionality, although it would not be as fast as a hardware implementation and would require CPU time.

CAN messages that are sent to a mailbox are not sent to the receive buffer and can therefore not be read using the functions [MIO_M51_Recv\(\)](#). Also, the mailbox holds only one single message, with a counter number and a time stamp that can be used to determine if new messages have been received and/or if there are messages that have been overwritten before being read.

MIO_M51_MBoxDelete()

```
int MIO_M51_MBoxDelete(int Slot, int Ch, int Id)
```

Description

The mailbox with the specified `Id` is deleted. The messages with the id `Id` will then be sent to the receive buffer.

`Slot` and `Ch` give the location and channel number of the CAN interface.

MIO_M51_MBoxRead()

```
long MIO_M51_MBoxRead(int Slot, int Ch, int Id, CAN_Msg *Msg, unsigned *pTimestamp)
```

Description

[MIO_M51_MBoxRead\(\)](#) reads the contents of the mailbox identified by `Id`.

`Slot` and `Ch` give the location and channel number of the CAN interface.

`Id` is the message identifier whose mailbox is to be read.

`pTimestamp` will contain a pointer to the timestamp of the last message upon return.

`Msg` contains the CAN message.

The function returns the number of messages that were put into the mailbox since its creation. If the mailbox does not exist, -1 is returned.

MIO_M51_SetTimestampFct()

```
MIO_TimestampFct MIO_M51_SetTimestampFct(int Slot, int Ch, unsigned (*Func) (void))
```

Description

With the help of this function a user defined function can be specified for the determination of the time stamps in place of the normally used system function `SysGetUTime()`. A typical application is the installation of a timer with a higher resolution.

`Func` is a pointer to a function, which returns the time stamp (type `unsigned int`). It is necessary that the function has a small time requirement – less than 2 micro seconds – and is callable from the interrupt level.

`Slot` and `Ch` give the location and channel number of the CAN interface.

A pointer to the old time stamp function is returned.

MIO_M51_SetTransMode()

```
void MIO_M51_SetTransMode(int Slot, int Ch, int OnOff)
```

Description

With [MIO_M51_EnableIds\(\)](#) the user can freely determine the messages which can be received. Normally with the help of the acceptance mask of the interface module the messages are pre-selected (hardware function, no CPU load). Since this selection is indistinct (some messages which were not selected are still received), additional selection by software is used.

In the transparency mode the preselection in hardware is deactivated, however the software selection remains active. The transparency mode makes it possible, for example, to log all CAN traffic in the background. Applications that read the messages with [MIO_M51_Recv\(\)](#) from the receive buffer will still only see the selected Messages.

The CPU load is substantially higher with transparency mode switched on due to the missing hardware preselection. In particular, this is to be considered during high data transmission rates and high bus utilization.

Slot and Ch give the location and channel number of the CAN interface.

OnOff = 1 activates, OnOff = 0 deactivates transparent mode.



In order to enable the reception of extended CAN messages, call this function with OnOff = 1. Using the functions [MIO_M51_EnableIds\(\)](#) or [MIO_M51_DisableIds\(\)](#) – as a side effect – will disable the reception of most extended CAN messages.

MIO_M51_UnlockTxBuf()

```
int MIO_M51_UnlockTxBuf(int Slot, int Ch)
```

Description

This function prevents the transmit buffer from overflowing, if some inconsistency of data between user process and MIO driver lead to a frozen CAN transmitter.

[MIO_M51_UnlockTxBuf\(\)](#) checks if the transmit buffer is at least half full and whether the CAN controller is still transmitting CAN messages or not. In case the CAN transmitter is locked up in some undesired state the transmission of CAN messages will be reinitiated.



Use this function carefully. Normally, the CAN transmitter never freezes. If you use this function, some essential CAN messages may get lost.

MIO_M51_SetSleepMode()

```
void MIO_M51_SetSleepMode (int Slot, int Channel, int EnSleepMode, int EnLOMode);
```

Description

This function enables the *sleep mode* or the *listen only mode* of the CAN controller.

To switch the controller into the *sleep mode* set `EnSleepMode` to “1”. The *sleep mode* can only be reached if no bus activity is recognized and no interrupt is pending. The controller will wake up if either `EnSleepMode` is set to “0” or the bus gets active.

If `EnLOMode` is set to “1” the controller is switched to the *listen only mode*. In the mode the controller also does not send acknowledges but is not sleeping. This mode can be used for analyzing the CAN bus.

MIO_M51_TraceInit()

```
int MIO_M51_TraceInit(int Slot, int BufSize, char ChMask)
```

Description

Enables a data trace for CAN messages received and transmitted by the M51 module at slot `Slot` on all channels specified by `ChMask`.

A buffer of `BufSize` bytes will be allocated to hold all CAN messages received and transmitted on the specified CAN channels. Note, that currently the CAN messages will be logged by the user process. So, there will always be a significant period of time between logging a message and being handled by the MIO driver (and therefore being really received or really transmitted).

MIO_M51_TraceStop()

```
int MIO_M51_TraceStop(int Slot, const int Abort, const char *FName)
```

Description

`MIO_M51_TraceStop()` stops logging of CAN messages and writes the contents of the previously allocated buffer (obtained by using `MIO_M51_UnlockTxBuf()`) to the file `FName`. If `Abort != 0`, then no bytes will be written. The CAN trace will be aborted.



This function does not meet real time conditions, as there is file I/O, and must not be called in real time context.

The format of the output file is binary. Each CAN message is stored as a block of variable length, depending on the length of the data field. The format of one CAN message is as followed:

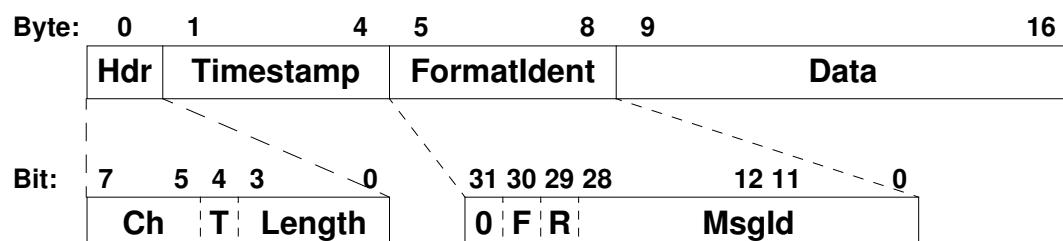


Figure 10.3: CAN message data record in Trace file

Each data record consists of one header byte, a 4-byte time stamp, a format identifier and up to 8 data bytes. Depending on the length of the CAN message, the data record has a size of 9, up to 17 bytes.

The format of the header byte `Hdr` is:

- Bits 0 .. 3:Length of the Data field of the CAN message in bytes (0 .. 8 data bytes)
- Bit 4:Type of CAN message, 0 for Rx message, 1 for Tx message
- Bits 5 .. 7:CAN Channel on which the message was sent or received

The **Timestamp** is an unsigned 32 bit integer value in network byte order (MSB first or Big Endian). It tells the number of micro second relative to the point of time when the CAN trace was started.

The **FormatIdent** field contains the message identifier **MsgId**, the remote transmission request flag **R** and the frame format flag **F**. It is written as 32 bit unsigned integer in network byte order (MSB first or Big Endian):

- Bits 0 .. 28:**MsgId** of CAN message. Bits 12 .. 28 are only valid for extended CAN messages
- Bit 29:**RTR** bit, 0 for normal data frames, 1 for remote requests
- Bit 30:**Frame Format**, 0 for standard messages, 1 for extended CAN messages with 29 bit identifier

The **Data** field contains the payload data of the CAN message. This part has a variable length, exactly the number of bytes as specified in the **Length** field of the header **Hdr**.

MIO_M51_Status()

```
int MIO_M51_Status(int Slot, int Ch, CAN_Info *Info)
```

Description

Returns some status information of the CAN controller located on the M51 module at slot **Slot** and channel **Ch**. The information will be written to a **CAN_Info** struct, which is defined as listed below:

Listing 10.8: The **CAN_Info** struct

```

1:  typedef struct CAN_Info {
2:      unsigned char Mode;           /* Mode register */
3:      unsigned char Status;        /* Status register */
4:      unsigned char IntrEnable;    /* Interrupt Enable register */
5:      unsigned char BusTiming[2];   /* Bus Timing registers */
6:      unsigned char ArbLostCapt;   /* Arbitration Lost Capture register */
7:      unsigned char ErrCodeCapt;   /* Error Code Capture register */
8:      unsigned char ErrWarnLmt;    /* Error Warning Limit */
9:      unsigned char RxErrCnt;      /* Receive Error Counter */
10:     unsigned char TxErrCnt;      /* Transmit Error Counter */
11:     unsigned char RxMsgCnt;      /* No of Messages available in RxFIFO */
12:     unsigned char ClockDivider;  /* Clock Divider register */
13:     unsigned char AcceptCode[4];  /* Acceptance Code registers */
14:     unsigned char AcceptMask[4];  /* Acceptance Mask registers */
15:     unsigned     TxClearCnt[5];   /* No of Auto Clears for TxErrCnt */
16: } CAN_Info;
```

The data field **TxClearCnt** contains additional information about how often the transmit buffer was cleared, e.g. automatically by the MIO driver when there was an Bus Error interrupt.

MIO_M51_GetRxErrCnt()

```
int MIO_M51_GetRxErrCnt(int Slot, int Ch)
```

Description

Returns the value of the receive error counter register of the CAN controller at `Slot` and `Ch`.

MIO_M51_GetTxErrCnt()

```
int MIO_M51_GetTxErrCnt(int Slot, int Ch)
```

Description

Returns the value of the transmit error counter register of the CAN controller at `Slot` and `Ch`.

MIO_M51_CANInfo()

```
char *MIO_M51_CANInfo(int Slot, int Ch, char *Buf, char *LinePrefix)
```

Description

Writes some useful and descriptive information about the CAN controller at `Slot` and `Ch` into buffer `Buf`, or directly to standard out.

If `Buf` is not NULL, then the information will be written into `Buf`, otherwise it is written directly to `stdout`. `LinePrefix` will be prepended each written line. This is useful, if you want to log `Buf` with the Log functions of CarMaker.

The printed information looks like:

```
Bus Status:           Bus-On
Transmitter Status: idle, last Tx completed, TxBuf released
Receiver Status:    idle, RxFIFO empty
Chip State:          Active / TxErrCnt = 51, RxErrCnt = 50
```

`Buf` should have a size of at least 300 bytes.

MIO_M51_GetRegister()

```
int MIO_M51_GetRegister(int Slot, int Ch, unsigned addr, int reset)
```

Description

Returns the value of register at `addr` on CAN controller at `Slot` and `Ch`. If `reset` = 1, then the CAN controller will be requested to enter reset mode first before reading the register value.

MIO_M51_ListAddr()

```
void MIO_M51_ListAddr(int Slot, int Ch)
```

Description

Prints a table with all registers and addresses of the CAN controller at `Slot` and `Ch` to standard out.

MIO_M51_DumpPSCC()

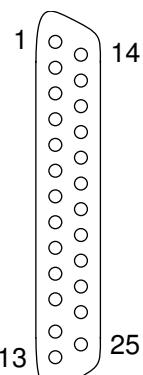
```
void MIO_M51_DumpPSCC(int Slot, int Ch)
```

Description

Dumps all register values of the CAN controller at `Slot` and `Ch` to standard out.

10.4.9 M62 / M62N: Analog Outputs (16 channels)

Connector assignment



Pin	Signal	Pin	Signal
1	Out_8	14	Out_0
2	Out_9	15	Out_1
3	Out_10	16	Out_2
4	Out_11	17	Out_3
5	Out_12	18	Out_4
6	Out_13	19	Out_5
7	Out_14	20	Out_6
8	Out_15	21	Out_7
9	GND	22	GND
10	GND	23	+15V
11	GND	24	-15V
12	GND	25	
13	GND		

Function Overview

Initialization and Configuration

- [MIO_M62_Config\(\)](#)
- [MIO_M62_SetMode\(\)](#)

Set Output Signals

- [MIO_M62_Set\(\)](#)
- [MIO_M62_SetV\(\)](#)

MIO_M62_Config()

```
int MIO_M62_Config(int Slot)
```

Description

By calling the function [MIO_M62_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M62, is specified. This function must be called one time, before any attempted access to the module.

Initial state of all output channels:

Output level: 0V



On M62 modules (hardware revision < 1.0), jumper J1/J2 is used to specify bipolar or unipolar mode.

On M62N modules (hardware revision 1.0 and newer), this feature is realized in software (see function [MIO_M62_SetMode\(\)](#)). The initial output mode is unipolar.

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned

MIO_M62_SetMode()

```
void MIO_M62_SetMode(int Slot, int Mode)
```

Description

This function configures if the modul works in unipolar or bipolar mode.

Parameters

Mode Value	Working Mode	Voltage Range
0	Unipolar	0 .. +10V
1	Bipolar	-10V .. +10V



This function is only supported by M62N modules with hardware revision 1.0 and newer. On older M62 modules, the bipolar mode is configured by jumpers on the module, Therfore this function does not have any effect. See datasheet of the older M62 module.

MIO_M62_Set()

```
void MIO_M62_Set(int Slot, int Ch, int Value)
```

Description

Sets the value of an output channel.

Slot and Ch indicate the card location and the channel number.

Parameters

Value contains the output value left justified in a 16bit short integer (unsigned).

Voltage	Unipolar Mode ^a	Bipolar Mode ^a
-10V	not available	0x0000
0V	0x0000	0x8000
+10V	0xFFFF	0xFFFF

a. see function [MIO_M62_SetMode\(\)](#)

The D/A conversion is started immediately after the output value is written to the module. Due to the hardware structure of the M62N module, it takes up to 13.5 micro seconds until the target voltage level appears at the output (M62 module: 15 micro seconds).

MIO_M62_SetV()

```
void MIO_M62_SetV(int Slot, int ChMask, int *Values)
```

Description

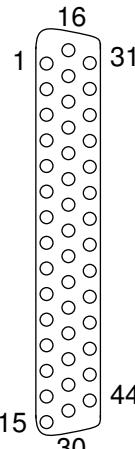
Similar to [MIO_M62_Set\(\)](#), however several output channels can be set at the same time. This function is particularly efficient for setting several channels.

Parameters

- The channels concerned are represented by the bit list ChMask (bit 0 .. 15 used): Bit n is set to 1 if channel n is to be set. Therefore, if ChMask==0xffff then all 16 channels are to be set.
- Values is a pointer to an array (vector), which contains the desired values in concise form, i.e. without gaps for unselected channels.

10.4.10 M66: Binary Inputs/Outputs (32 channels)

Connector assignment



Pin	Signal	Pin	Signal	Pin	Signal
1	IO_0	16	GND_1	31	IO_1
2	IO_3	17	IO_2	32	IO_4
3	IO_6	18	IO_5	33	IO_7
4	+24V_1	19	+24V_1	34	GND_2
5	IO_9	20	IO_8	35	IO_10
6	IO_12	21	IO_11	36	IO_13
7	IO_15	22	IO_14	37	+24V_2
8	GND_3	23	+24V_2	38	IO_16
9	IO_18	24	IO_17	39	IO_19
10	IO_21	25	IO_20	40	IO_22
11	+24V_3	26	IO_23	41	+24V_3
12	IO_24	27	GND_4	42	IO_25
13	IO_27	28	IO_26	43	IO_28
14	IO_30	29	IO_29	44	IO_31
15	+24V_4	30	+24V_4		

Function Overview

Initialization and Configuration

- [MIO_M66_Config\(\)](#)
- [MIO_M66_SetDirection\(\)](#)
- [MIO_M66_SetDirectionV\(\)](#)

Set Output Signals

- [MIO_M66_Set\(\)](#)
- [MIO_M66_SetV\(\)](#)

Get Input Signals

- [MIO_M66_Read\(\)](#)
- [MIO_M66_ReadV\(\)](#)

MIO_M66_Config()

```
int MIO_M66_Config (int Slot)
```

Description

By calling the function [MIO_M66_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M66, is specified. This function must be called one time, before any attempted access to the module.

Initial state of all channels: configured as input (output switch open).

MIO_M66_SetDirection()

```
void MIO_M66_SetDirection (int Slot, int Channel, int IsOutput)
```

Description

Selects the operation mode of the given channel. A channel can be either used as input (`IsOutput = 0`, default) or as output (`IsOutput = 1`).

MIO_M66_SetDirectionV()

```
void MIO_M66_SetDirectionV (int Slot, unsigned int IsOutputMask)
```

Description

Similar to [MIO_M66_SetDirection\(\)](#), except several channels can be configured simultaneously.

`IsOutputMask` is a bit field (bit 0..31 used), which specifies, which channels are used as outputs, and which are used as inputs. If bit n has the value 1, the channel n is configured as output, else the channel is used as input.

Example

```
IsOutputMask = 0x1f8;  
MIO_M66_SetDirectionV(s, IsOutputMask);
```

Configures channel 3 .. 8 as outputs, while channels 0 .. 2 and 9 .. 31 are used as inputs.

MIO_M66_Set()

```
int MIO_M66_Set (int Slot, int Channel, int Value)
```

Description

Sets the value of the selected channel (e.g. sets the switch position) to the specified value (`Value = 0`: switch open, `Value = 1`: switch closed). If `Channel` is configured as input, or if the requested value would not change the state, this function does not access the M module (e.g. no operation is performed).

MIO_M66_SetV()

```
void MIO_M66_SetV (int Slot, unsigned int BitMask, unsigned int EnableMask)
```

Description

Similar to [MIO_M62_Set\(\)](#), except several channels can be set simultaneously. The function is very efficient when many channels are being set.

`EnableMask` is a bit field (bit 0..31 used), which specifies the channels which shall be set. If bit n has the value 1, the channel n is set.

`BitMask` is a bit field (bit 0..31 used), which specifies the requested switch positions. If bit n has the value 1, the channel n is set to 1 (switch open), if bit n has the value 0, the channel is set to 0 (switch closed).

Example

```
EnableMask = 0x1f8;
BitMask = 0x0f0;
MIO_M66_SetV(s, BitMask, EnableMask);
```

Sets channel 4 .. 7 to 1 (switch closed), while channels 3 and 8 are set to 0 (switch open). All other channels are left untouched.

Channels, that are configured as inputs, will not be touched by this function. For output channels, where the requested value is already set (state does not change), nothing is written to the M module.

MIO_M66_Read()

```
unsigned int MIO_M66_Read (int Slot, int Channel)
```

Description

Reads the value of the specified channel and returns the current value. This function can also be used for output channels. Refer also to [Performance aspects](#), for information on how to achieve lowest possible access times.

Return Value

Bitwise combination of the current state (low / high), and if the state changed since the last reading:

Bit Position	Meaning
0	Current state: <ul style="list-style-type: none">• 0: low (input) / open (output)• 1: high (input) / closed (output)
1	Rising Edge: 1, if state changed from 0 to 1 since last read
2	Falling Edge: 1, if state changed from 1 to 0 since last read

MIO_M66_ReadV()

```
unsigned int MIO_M66_ReadV (int Slot, unsigned int EnableMask)
```

Description

Similar to [MIO_M66_ReadV\(\)](#), except several input channels can be read simultaneously. The function is very efficient when many channels are being read (s. [Performance aspects](#) for information on how to achieve lowest possible access times).

EnableMask is a bit field (bit 0..31 used), which specifies the channels which can be queried. If bit n has the value 1, the channel n is queried.

Example

```
EnableMask = 0x1f8;  
States = MIO_M66_ReadV(s, EnableMask, 0, v);
```

Causes the inquiry of channel 3 .. 8.



This function only reads the state of input channels. Channels, that are configured as outputs, but are selected with EnableMask, are not read.

Return Value

Bitwise combination of the state of all read input channels. Bit n corresponds to channel n.

Performance aspects

Reading the state of one input channel, takes about 1.7 micro seconds. Thus, reading all 32 channels with [MIO_M66_Read\(\)](#), takes about 55 micro seconds. This is a consequence of the address layout on the M66 module, where each channel occupies an register of 16 bits.

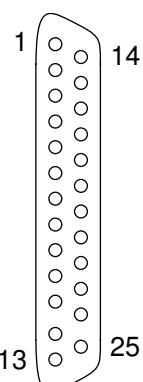
However, the function [MIO_M66_ReadV\(\)](#) reads multiple channels up to 2 times faster, than reading the same channels with [MIO_M66_Read\(\)](#): e.g. all 32 input channels are read in only 27 micro seconds. This is possible with highly optimized 64-bit reads, which allows to read the state of up to 4 input channels at once.

In order to have the lowest possible access times, we recommend to consider the following aspects, when designing the electrical layout of the TestSystem:

- Group inputs and outputs together (e.g. use channels 0 .. 3 as inputs and 4 .. 7 as outputs, rather than 0, 2, 4, 6 as inputs and 1, 3, 5, 7 as outputs)
- Choose a multiple of 4 as channel number for the first channel of a group (e.g. use channels 0 .. 4 as inputs and 8 .. 12 as outputs, rather than 5 .. 9 as outputs)

10.4.11 M77: Quadruple RS232/423 - RS422/485 UART

Connector assignment



Pin	Signal	Pin	Signal
1	TXD/RXD[0]+	14	TXD/RXD[0]-
2	TXD[0]+	15	TXD[0]-
3	TERM[0]	16	I-GND[0]
4	TXD/RXD[1]+	17	TXD/RXD[1]-
5	TXD[1]+	18	TXD[1]-
6	TERM[1]	19	I-GND[1]
7	TXD/RXD[2]+	20	TXD/RXD[2]-
8	TXD[2]+	21	TXD[2]-
9	TERM[2]	22	I-GND[2]
10	TXD/RXD[3]+	23	TXD/RXD[3]-
11	TXD[3]+	24	TXD[3]-
12	TERM[3]	25	I-GND[3]
13			

Function Overview

Initialization and Configuration

- [MIO_M77_Config\(\)](#)
- [MIO_M77_UART_Reset\(\)](#)
- [MIO_M77_UART_Flush\(\)](#)
- [MIO_M77_SetBaud\(\)](#)
- [MIO_M77_SetMode\(\)](#)
- [MIO_M77_GetInputClock\(\)](#)
- [MIO_M77_SetInputClock\(\)](#)

Data Transfer

- [MIO_M77_Write\(\)](#)
- [MIO_M77_Read\(\)](#)

Diagnostics

- [MIO_M77_RxFillLvl\(\)](#)
- [MIO_M77_TxFillLvl\(\)](#)

MIO_M77_Config()

```
int MIO_M77_Config(int Slot)
```

Description

By calling the function [MIO_M77_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M77 is specified. This function must be called one time, before any attempted access to the module.

The transmit trigger levels of all UARTs will be set to maximum (127), the receive trigger levels to minimum (1). No other configuration is done on the UARTs, in particular the baud rate is not modified.



The input clock, which is important for the baud rate generation, is assumed to be 32 MHz. This does not meet the default configuration of a virgin M77 module. For CarMaker/HIL applications, the default oscillator with 18.432 MHz is replaced by an oscillator with 32 MHz. If your M77 module uses an oscillator with a frequency different from 32 MHz, you should change the input clock with [MIO_M77_SetInputClock\(\)](#).

If the module cannot be configured-initialized, the function returns an error code, otherwise 0 is returned.

The function resets all UARTs to Power-Up state and sets the baud rate to maximum. The resulting baud rate depends on the oscillator frequency:

Table 10.1: Maximum possible baud rates

Oscillator frequency	Maximum baud rate
1.8432 MHz	115.200 kbaud
7.3728 MHz	460.800 kbaud
14.7456 MHz	921.600 kbaud
18.432 MHz	1.152 Mbaud
32.000 MHz	2.000 Mbaud
33.000 MHz	2.0625 Mbaud
40.000 MHz	2.500 Mbaud
50.000 MHz	3.125 Mbaud
60.000 MHz	3.750 Mbaud

Both FIFOs (Rx-FIFO and Tx-FIFO) are flushed. Any remaining data is lost.

The UARTs are configured as followed:

- 950 mode trigger levels enabled
- all interrupts disabled
- transmitter FIFO and receiver FIFO enabled

MIO_M77_GetInputClock()

```
int MIO_M77_GetInputClock(int Slot)
```

Description

Returns the input clock in Hz, which is needed to program the baud rate generator.

MIO_M77_SetInputClock()

```
void MIO_M77_SetInputClock(int Slot, int InputClock)
```

Description

Sets the input clock in Hz, which is needed to program the baud rate generator.

The input clock should always match the frequency of the oscillator, which is mounted on the M77 module. If not, it is not possible to set accurate baud rates.



MIO_M77_UART_Reset()

```
void MIO_M77_UART_Reset(int Slot, int Ch)
```

Description

This function resets the UART at slot `Slot` and channel `Ch` to Power-Up state. The UART results in the same state as after a hardware reset. For a proper use, the UART has to be reinitialized.

MIO_M77_UART_Flush()

```
void MIO_M77_UART_Flush(int Slot, int Ch)
```

Description

Flushes the transmitter FIFO and receiver FIFO of the UART located at `Slot` and `Ch`. Any remaining data bytes in the FIFOs get lost.

MIO_M77_SetBaud()

```
void MIO_M77_SetBaud(int Slot, int Ch, int Baud)
```

Description

Sets the desired baud rate by programming the divisor latch registers of the baud rate generator. If the input clock is not divisible by the baud rate, the resulting baud rate may be different from the requested. The divisor is determined through integer divisions with the following formula:

$$\text{divisor} = (\text{InputClock}/\text{RequBaudrate} + 8) / 16 \quad (\text{EQ 1})$$

Thus, the resulting baud rate need not to be exactly the same value as the requested baud rate.



If the requested baud rate is near to the maximum and the input clock is not an integer multiple of it, then the resulting baud rate may differ significantly. For example: if you want to set a baud rate of 1.5 Mbaud (32 MHz input clock!), the divisor will be 1, which results in the maximum baud rate of 2 Mbaud.

MIO_M77_SetMode()

```
void MIO_M77_SetMode(int Slot, int Ch, int Mode)
```

Description

This function allows to set the operating mode of the UART, located at `Slot` and `Ch`.

The M77 module supports the following operating modes:

Table 10.2: Operating modes of M77 module

Operating mode	Mode (bits 0..2)	Constants defined in <code>mio.h</code>
RS423	000	<code>MIO_M77_MODE_RS423</code>
RS422 half-duplex	001	<code>MIO_M77_MODE_RS422half</code>
RS422 full-duplex	010	<code>MIO_M77_MODE_RS422full</code>
RS485 half-duplex	011	<code>MIO_M77_MODE_RS485half</code>
RS485 full-duplex	100	<code>MIO_M77_MODE_RS485full</code>
RS232	111	<code>MIO_M77_MODE_RS232</code>

If you are using RS422 half-duplex mode or RS485 half-duplex mode, setting `RX_EN` (bit 3) enables echoing effect, clearing this bit disables the echoing effect. In the modes RS232, RS423, RS422 full-duplex and RS485 full-duplex, setting `RX_EN` has no effect.

MIO_M77_Write()

```
int MIO_M77_Write(int Slot, int Ch, char *Data, int Len)
```

Description

Writes up to `Len` bytes of the character array `Data` to the Transmitter FIFO of the UART located at `Slot` and `Ch`.

Returns the number of written bytes. Depending on the fill level of the Transmitter FIFO and the baud rate, the number of written bytes can be about up to 128 bytes.

MIO_M77_Read()

```
int MIO_M77_Read(int Slot, int Ch, char *Data, int MaxLen)
```

Description

Reads all available or `MaxLen` bytes from the Receiver FIFO of the UART located at `Slot` and `Ch` and copies them to the character array `Data`.

Returns the number of copied bytes. Depending on the fill level of the Receiver FIFO and the baud rate, about up to 128 bytes can be copied from the UART to `Data`.

MIO_M77_RxFillLvl()

```
int MIO_M77_RxFillLvl(int Slot, int Ch)
```

Description

Returns the number of available bytes in the Receiver FIFO of the UART located at `Slot` and `Ch`.

MIO_M77_TxFillLvl()

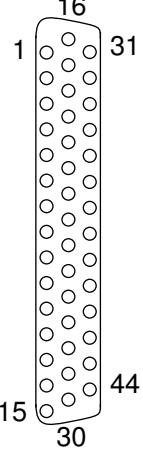
```
int MIO_M77_TxFillLvl(int Slot, int Ch)
```

Description

Returns the number of bytes in the Transmitter FIFO of the UART located at `Slot` and `Ch`, that have not been transferred.

10.4.12 M81: Binary Outputs (16 channels)

Connector assignment



Pin	Signal	Pin	Signal	Pin	Signal
1	DA_1	16	DA_0	31	DA_2
2	DB_1	17	-	32	DB_2
3	DA_3	18	DB_0	33	DA_4
4	DB_3	19	-	34	DB_4
5	DA_5	20	-	35	DA_6
6	DB_5	21	-	36	DB_6
7	DA_7	22	-	37	DA_8
8	DB_7	23	-	38	DB_8
9	DA_9	24	-	39	DA_10
10	DB_9	25	-	40	DB_10
11	DA_11	26	-	41	DA_12
12	DB_11	27	-	42	DB_12
13	DA_13	28	-	43	DA_14
14	DB_13	29	-	44	DB_14
15	DA_15	30	DB_15		

Function Overview

Initialization and Configuration

- [MIO_M81_Config\(\)](#)

Set Output Signals

- [MIO_M81_Set\(\)](#)

MIO_M81_Config()

```
int MIO_M81_Config(int Slot)
```

Description

(see function description for [MIO_M27_Config\(\)](#))

MIO_M81_Set()

```
void MIO_M81_Set(int Slot, int Value)
```

Description

(see function description for [MIO_M27_Set\(\)](#))

10.4.13 M400: Wheelspeed Generator, 6 Channels

Connector Assignment

Pin	Signal	Pin	Signal
1	Wheel Speed 0 (+)	14	Wheel Speed 0 (-)
2		15	
3	Wheel Speed 1 (+)	16	Wheel Speed 1 (-)
4		17	
5	Wheel Speed 2 (+)	18	Wheel Speed 2 (-)
6		19	
7	Wheel Speed 3 (+)	20	Wheel Speed 3 (-)
8		21	
9	Wheel Speed 4 (+)	22	Wheel Speed 4 (-)
10		23	
11	Wheel Speed 5 (+)	24	Wheel Speed 5 (-)
12		25	
13			

For pinning of 50 pin connector see hardware manual.

Function Overview

If the A24D32 address space is available on the used carrier board this address space is used. Otherwise the A8D16 address space is used.

Initialization and Configuration

- `MIO_M400_Config()`
- `MIO_M400_SetMode()`
- `MIO_M400_SetSignalsPerTurn()`

Set Output Signals

- `MIO_M400_SetFrequency()`
- `MIO_M400_SetWheelspeed()`
- `MIO_M400_SetDutyCycle()`
- `MIO_M400_SetPulseWidth()`

Since Revision 2.0 OH4x wheel speed sensors are supported which contain a serial data protocol. The following functions only work with revisions >=2.0. Furthermore it is possible to generate errors - missing teeth and jittering teeth - with revisions >=2.0.

OH4x specific functions

- `MIO_M400_SetMode_OH44()`
- `MIO_M400_SetData()`

Generating Errors

- `MIO_M400_SetSignalsPerTurn()`
- `MIO_M400_ConfigTooth()`

MIO_M400_Config()

```
int MIO_M400_Config(int Slot)
```

Description

By calling the function `MIO_M400_Config()` the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M400 is specified. This function must be called one time, before any attempted access to the module.

Initial state of the module:

- The output mode is set to “Binary Out”.
- The duty cycle is set to 0x00. This means a high:low relationship of 0%. The output is always low, independent of the adjusted frequency.
- The frequency is set to 0.0Hz

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

MIO_M400_SetMode()

```
void MIO_M400_SetMode (int Slot, int Channel, int OutputMode, int PulseWidthMode)
```

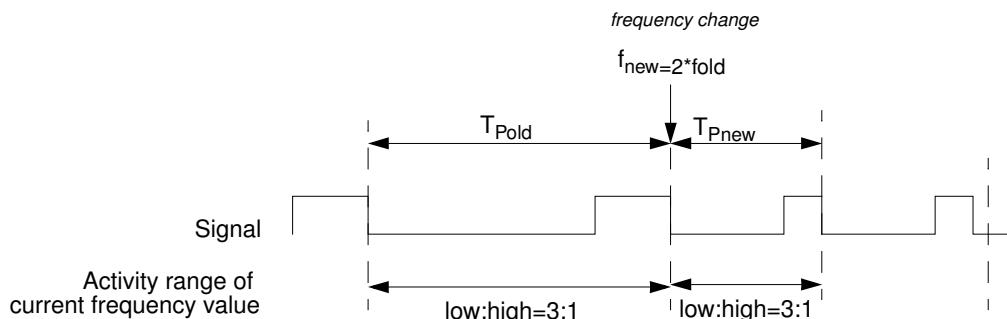
Description

This function configures the behavior of each channel of the M400 in the normal mode. The module offers 2 different frequency update modes.

If OH4x specific sensors are used use `MIO_M400_SetMode_OH44()` instead of `MIO_M400_SetMode()`.

Parameters

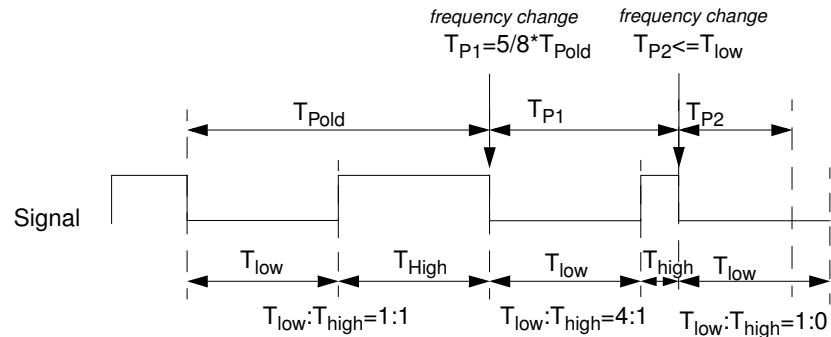
- `OutputMode` configures the currents of the wheelspeed channel:
0 - Disabled
1 - 7/14mA
2 - 7/20mA
3 - 7/28mA
- `PulseWidthMode`
0 - Duty Cycle Mode
With the Duty Cycle Mode, the relationship between high and low stays the same, even if the frequency is changed.



1 - Fix low duration during the period

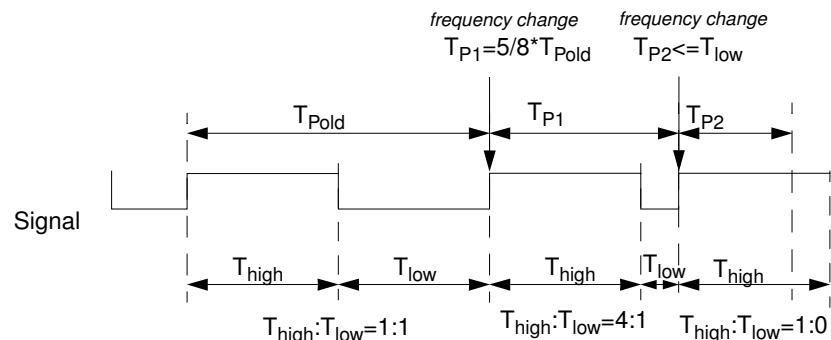
The duration of the low level is fix. So the low level duration stays the same even if the

frequency changes while the high duration is adapted to the remaining cycle duration. Thus if the period is smaller than the low level duration the signal output is low all the time.



2 - Fix high duration during the period

The duration of the high level is fix. So the high level duration stays the same even if the frequency is changed while the low duration is adapted to the remaining cycle duration.. Thus if the period is smaller than the high level duration the signal output is high all the time.



MIO_M400_SetSignalsPerTurn()

```
void MIO_M400_SetSignalsPerTurn (int Slot, int Channel, int SignalsPerTurn)
```

Description

[MIO_M400_SetSignalsPerTurn\(\)](#) configures how many pulses happen on the wheel speed sensor, when the wheel is turned 1 full rotation.

If signals per turn is configured, you can use the function [MIO_M400_SetWheelspeed\(\)](#) to adjust the frequency directly from the CarMaker's wheel speed value where the unit is rad/s.

MIO_M400_SetDutyCycle()

```
void MIO_M400_SetDutyCycle (int Slot, int Channel, double DutyCycle)
```

Description

Configures the relationship between high duration and low duration within one period. If `DutyCycle` is 0.0, the output signal is all high, if `DutyCycle` is 1.0, the output signal is all low.

The value of `DutyCycle` must be in the range of 0.0 ... 1.0.

This function only works correctly if the “Pulse Width-Mode” is set to “Duty Cycle Mode” in [MIO_M400_SetMode\(\)](#).

MIO_M400_SetPulseWidth()

```
void MIO_M400_SetPulseWidth (int Slot, int Channel, double PulseWidth)
```

Description

With this function the value for the fixed low duration or the fixed high duration (depending on the pulse width mode) is set. This duration stays the same, even if the frequency is changed.

The unit of `PulseWidth` is [sec]. It must be in the range 0sec to 20sec.

This function only works correctly if the “Pulse Width Mode” is set to “Fix low duration” or “Fix high duration” in [MIO_M400_SetMode\(\)](#).

MIO_M400_SetFrequency()

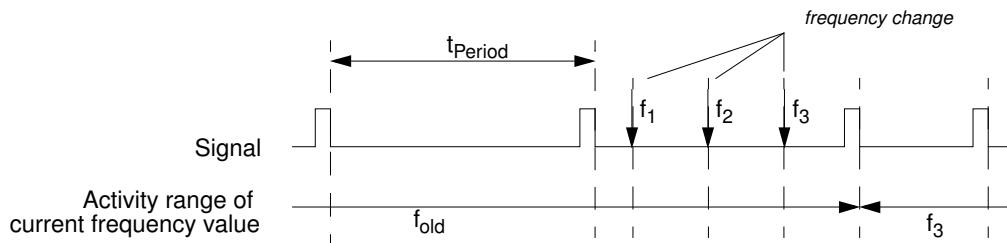
```
void MIO_M400_SetFrequency (int Slot, int Channel, double Freq, int ImmediateUpdate);
```

Description

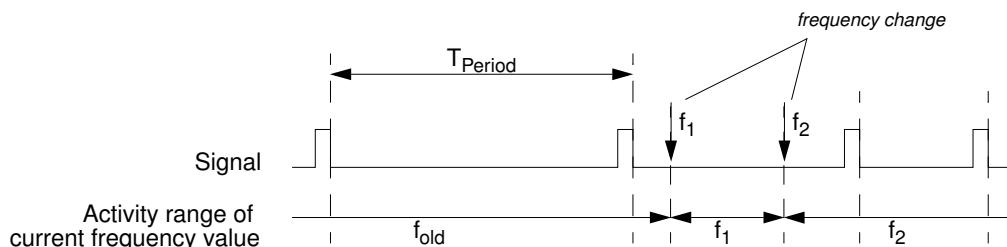
Set the frequency for 1 channel. The unit of the frequency value is Hz. For use with Car-Maker, it is easier to use [MIO_M400_SetWheelspeed\(\)](#).

If the frequency value is negative the function internally does an `fabs()`. So internally always positive frequency values are used.

- `ImmediateUpdate`
- 0 - Frequency will be updated with the beginning of the next period.



1 - Frequency will be updated immediately



MIO_M400_SetWheelspeed()

```
void MIO_M400_SetWheelspeed (int Slot, int Channel, double Wheelspeed)
```

Description

There are 2 ways to adjust the correct frequency for the module. One is to calculate the frequency yourself and use the function [MIO_M400_SetFrequency\(\)](#).

The other - much easier - way for usage with CarMaker is to call [MIO_M400_SetWheelspeed\(\)](#) where the unit of `Wheelspeed` is [rad/s] as calculated from CarMaker. Therefore you first have to configure the signals per turn via [MIO_M400_SetSignalsPerTurn\(\)](#).

When using this function, always the immediate update mode is used.

If the wheel speed value is negative the function internally does an `fabs()`. So internally always positive wheel speed values are used.

MIO_M400_SetMode_OH44()

```
void MIO_M400_SetMode_OH44 (int Slot, int Channel, int nDataBits,
                           int SensorType /*1-OH42, 3-OH44 */,
                           int ActiveEdge /* 0-Rising, 1-Both */,
                           int DataPeriod /* 10..240us */,
                           double MinFreq,double StandStillRepeat);
```

Description

This function configures the module for use with OH42 or OH44 sensors. Both sensor types have a serial protocol but they differ amongst others in the handling of the serial data during higher frequencies.

If no sensors with serial protocols are used use [MIO_M400_SetMode\(\)](#) instead of [MIO_M400_SetMode_OH44\(\)](#).

Parameters

- `nDataBits`

This is the number of ALL bits, the serial protocol contains, including the parity. By the OH4x specifications this value should be set to 9. Other values are for fail-safe testing issues only. Valid range: 0 .. 15.

- `SensorType`

1 - OH42, 3 - OH44

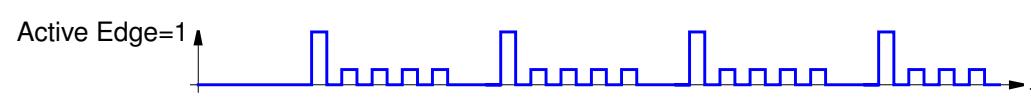
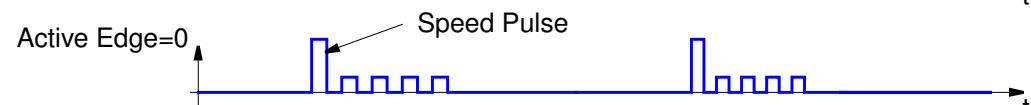
- `ActiveEdge`

If this parameter is set to "0" the serial protocol is triggered on the rising edge of each tooth, otherwise it is triggered on the rising and the falling edge of the tooth (default)

Wheel Tooth

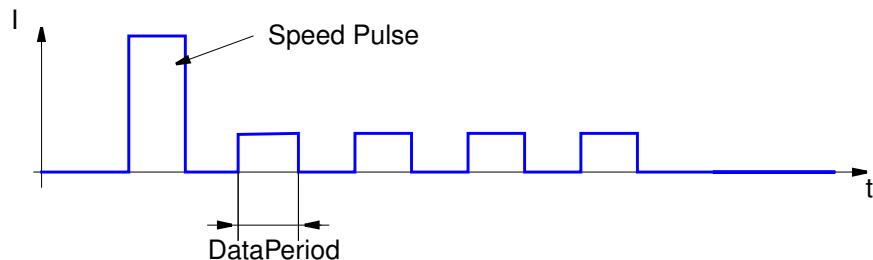


Speed Pulse



- DataPeriod

This is the cycle duration of one bit of the serial protocol in microseconds. Valid range 10..240 (us). Specification: 40..60us.



- MinFreq

This is the minimum frequency at which the speed pulse is generated as an 28mA signal. Below this frequency a speed replacement pulse with 14mA and a fix repetition rate is generated. The repetition rate of the speed replacement pulse is set by StandStillRepeat. Specification: 3.33Hz

- StandStillRepeat

This is the repetition frequency of the stand still signal. In standstill condition the serial protocol is repeated with a fix repetition rate and a speed replacement pulse instead of the speed pulse. Specification: 6.66Hz (=150ms)

MIO_M400_SetData()

```
void MIO_M400_SetData (int Slot, int Channel, unsigned short SerialData);
```

Description

This function sets the data bits of the serial protocol. The parity must be handled as a normal data bit.

Parameters

- SerialData
- The serial data contains the information that is transmitted on the bus. Bit 0 is the bit which is send as the first. Bit 1 is send as the second bit and so on.

MIO_M400_ConfigTooth()

```
void MIO_M400_ConfigTooth (int Slot, int Channel,
                           int ToothNo, int IsMissing, float JitterAngle);
```

Description

This function is used to configure erroneous teeth on the tooth wheel. Before this function can be used, the number of teeth on the wheel must be configured by [MIO_M400_SetSignalsPerTurn\(\)](#), because with this function one special tooth of one wheel turn is specified. Any number of teeth can be configured as erroneous. There is no reset function for the errors. Thus every tooth must be set to its No-Error-State individually.

Parameters

- ToothNo

This is the number of the tooth to configure. The valid range is 0 to “signals per turn”-1.

Example: If the module is configured to have 48 teeth per turn, 47 erroneous teeth can be configured. A maximum of 128 teeth per turn is possible.

- **IsMissing**

If this parameter is set to “1”, the specified tooth is simulated as missing, which means no data is generated on OH4xx sensors and the signal stays 7mA on common sensors.

Note: If a tooth is missing, of course both edges of the tooth are missing. This means if OH4xx sensors are used and ActiveEdge is set to “both”, one missing tooth results in 2 missing speed pulses on the data line of the sensor!

- **JitterAngle**

JitterAngle configures the angle by which a tooth is shifted. A positive value means that the speed signal comes more early (hurries ahead) while a negative value means that the speed signal comes later (it lags).

Each tooth can be shifted a maximum of a quarter of the angle between 2 teeth in both directions.

Example:

The tooth wheel has 48 teeth. The angle between the middle of one tooth to the middle of the next tooth is $360\text{deg} / 48\text{teeth} = 7.5\text{deg}$. Therefore the maximum possible displacement angle is $\pm 1.875\text{deg}$. This sounds few, but remember that the width of a tooth in this case is about 3.75deg . Therefore it is enough to shift 2 teeth together with almost no gap between them.

Note: A missing tooth cannot jitter!

Note: If a tooth is displaced, both edges of the tooth are displaced in the same way. This means if OH4xx sensors are used and ActiveEdge is set to “both”, one displaced tooth results in 2 displaced speed pulses on the data line of the sensor!

10.4.14 M401: Frequency Generator and Frequency Meter (3 Units)

Connector assignment

Pin	Name	Pin	Name	Unit	General Function	Function with Quadrature Encoder / Decoder
1	Out 2-0(-)	26	Out 2-0(+)	2	RS422-Out(0)	Output Signal "A"
2	Out 2-1(-)	27	Out 2-1(+)		RS422-Out(1)	Output Signal "B"
3	In 2-0(-)	28	In 2-0(+)		RS422-In(0)	Input Signal "A"
4	In 2-1(-)	29	In 2-1(+)		RS422-In(1)	Input Signal "B"
5	In 2-2(-)	30	In 2-2(+)		RS422-In(2)	Input Signal "Ref"
6	In 2-3(-)	31	In 2-3(+)		RS422-In(3)	
7	GND2	32	VCC2		Supply (+5V / 300mA)	
8	n.c.	33	n.c.			
9	n.c.	34	n.c.			
10	n.c.	35	n.c.			
11	n.c.	36	n.c.			
12	GND1	37	VCC1	1	Supply (+5V / 300mA)	
13	Out 1-0(-)	38	Out 1-0(+)		RS422-Out(0)	Output Signal "A"
14	Out 1-1(-)	39	Out 1-1(+)		RS422-Out(1)	Output Signal "B"
15	In 1-0(-)	40	In 1-0(+)		RS422-In(0)	Input Signal "A"
16	In 1-1(-)	41	In 1-1(+)		RS422-In(1)	Input Signal "B"
17	In 1-2(-)	42	In 1-2(+)		RS422-In(2)	Input Signal "Ref"
18	In 1-3(-)	43	In 1-3(+)		RS422-In(3)	
19	GND0	44	VCC0		Supply (+5V / 300mA)	
20	Out 0-0(-)	45	Out 0-0(+)	0	RS422-Out(0)	Output Signal "A"
21	Out 0-1(-)	46	Out 0-1(+)		RS422-Out(1)	Output Signal "B"
22	In 0-0(-)	47	In 0-0(+)		RS422-In(0)	Input Signal "A"
23	In 0-1(-)	48	In 0-1(+)		RS422-In(1)	Input Signal "B"
24	In 0-2(-)	49	In 0-2(+)		RS422-In(2)	Input Signal "Ref"
25	In 0-3(-)	5	In 0-3(+)		RS422-In(3)	

Function Overview

Initialization and Configuration

- `MIO_M401_Config()`
- `MIO_M401_SetUnitMode()`
- `MIO_M401_SetInPortMode()`
- `MIO_M401_SetInPortMode_CV()`
- `MIO_M401_SetOutPortMode()`

Set Output Signals

- `MIO_M401_SetOutPort()`
- `MIO_M401_SetFrequency()`

- [MIO_M401_SetDutyCycle\(\)](#)
- [MIO_M401_SetPulseWidth\(\)](#)
- [MIO_M401_SetQuadEnc\(\)](#)

Get Input Signals

- [MIO_M401_GetInPort\(\)](#)
- [MIO_M401_GetFrequency\(\)](#)
- [MIO_M401_GetFrequency_CV\(\)](#)
- [MIO_M401_GetDutyCycle\(\)](#)
- [MIO_M401_GetQuadDec\(\)](#)
- [MIO_M401_GetEdges\(\)](#)
- [MIO_M401_GetPeriods\(\)](#)
- [MIO_M401_GetTimeStamp\(\)](#)

MIO_M401_Config()

```
int MIO_M401_Config(int Slot)
```

Description

By calling the function [MIO_M401_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M401 is specified. This function must be called one time, before any attempted access to the module.

Initial state of the module:

- The Unit Mode of each unit is set to "Binary (In/Out)".
- The duty cycle is set to 0x00. This means a high:low relationship of 0%. The output is always low, independent of the adjusted frequency.
- The frequency is set to 0.0Hz

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

The module is set back to it's initial state.

MIO_M401_SetUnitMode()

```
void MIO_M401_SetUnitMode (int Slot, int Unit, tM401_UnitMode InputMode,  
                           tM401_UnitMode OutputMode);
```

Description

This function configures the input mode and the output mode of each unit.

Possible Values for `InputMode` and `OutputMode` are:

- `M401_Binary`
Sets the input mode of the selected unit to "binary in" or the output mode to "binary out".
- `M401_Frequency`
If the input mode is set to `M401_Frequency`, the 4 input channels of the selected unit are used as 4 independent frequency meters.
If the output mode is set to `M401_Frequency`, the 2 output channels of the selected unit are used as 2 independent frequency generators.
- `M401_Quadrature`
If the input mode is set to `M401_Quadrature`, the first 3 input channels are used as the "A", then "B" and the reference input of a quadature decoded signal. The forth input channel is used as an independent frequency meter.
If the output mode is set to `M401_Quadrature`, the 2 ouputs of the selected unit are used as the "A" and the "B" output of a quadature encoded signal.

MIO_M401_SetInPortMode()

```
void MIO_M401_SetInPortMode (int Slot, int Unit, int InPort, tM401_EdgeMode EdgeMode,  
                            int AvgPeriods, double TimeSlot, double ZeroDetectTime);
```

Description

With this function the frequency meter mode of each input port is configured. The module has 3 frequency units with 4 input ports and 2 output ports. The mode of each of the ports is configurable independently of the modes of other ports. However, in quadrature decoder mode only configuring the first and the forth input channel of the unit makes sense.

Parameters

- EdgeMode

With this parameter you can configure if rising, falling or all edges are used to measure the frequency. This perhaps is important if one of the edges of the measured signal is not as accurate as the other one, e.g. because it is a sawtooth signal where one edge is very flat. Possible values:

- M401_Falling

The frequency is measured within the last `AvgPeriods` periods starting with the last falling edge.

- M401_Rising

The frequency is measured within the last `AvgPeriods` periods starting with the last rising edge.

- M401_Both

The frequency is measured within the last `AvgPeriods` periods starting with the last edge, no matter, if the it was rising or falling.

- AvgPeriods

The number of periods to average over. The value must be in the range from 1 up to 128. The higher the value, the lower is the influence of the jitter in the measurement result. Anyway, high value works as low-pass filter. Thus if the value is chosen too high fast frequency changes cannot be detected.

- TimeSlot [s]

This is the maximum duration the frequency is averaged over. If the duration of the last `AvgPeriods` periods is higher than this value, the frequency is averaged over the periods that are within the `TimeSlot`, but not less than 1 period. This value must not be higher than 20 seconds. In the CarMaker Environment, the value should be between half the cycle time and 10 times the cycle time, depending on the frequency to be measured. If `TimeSlot` is set to "0.0", always the number specified in `AvgPeriods` is used and no maximum duration for the last `AvgPeriods` periods is considered.

With `TimeSlot` set to 0.0, reading one frequency value will be up to 5 times faster as with a value unequal 0.0. However, in this case the calculation of the frequency is not as accurate as if the value is not set to 0.0.

- ZeroDetectTime [s]

If no signal edges are detected for a duration higher than `ZeroDetectTime` seconds, a frequency of 0 is returned. This value must not be higher than 20seconds to avoid measurement errors in consequence of overruns of the circular buffer.

Please be aware that small values can falsify measurement results.

If `ZeroDetectTime` is set to "0.0", the zero detection is switched off. With `ZeroDetectTime` set to "0.0", reading one frequency value is up to 2 times faster as if the value is unequal 0.0. The acceleration also depends from if `TimeSlot` is set to 0 or not.

Setting this value to 0.0 only should be done, if the frequency is never set to 0 during simulation.

Example:

For faster detection of a frequency of "0", you will set this value to 0.1s. Now you can not measure frequencies lower than 10Hz. If you have a frequency of 5Hz, the measurement result always toggles between 0Hz and 5Hz.



MIO_M401_SetInPortMode_CV()

```
void MIO_M401_SetInPortMode_CV (int Slot, int Unit, int InPort,
                                tM401_EdgeMode EdgeMode, int AvgPeriods, int AvgCycles, double ZeroDetectTime);
```

Description

This function must be used together with the [MIO_M401_GetFrequency_CV\(\)](#).

With this function, the frequency meter mode of each input port is configured. The module has 3 frequency units with 4 input ports and 2 output ports. The mode of each of the ports is configurable independently of the modes of other ports. However, in quadrature decoder mode only configuring the first and the forth input channel of the unit makes sense.

Parameters

- EdgeMode

With this parameter you can configure if rising, falling or all edges are used to measure the frequency. This perhaps is important, if one of the edges of the measured signal is not as accurate as the other one, e.g. because it is a sawtooth signal where one edge is very flat. Possible values:

- M401_Falling

The frequency is measured within the last AvgPeriods periods starting with the last falling edge.

- M401_Rising

The frequency is measured within the last AvgPeriods periods starting with the last rising edge.

- M401_Both

The frequency is measured within the last AvgPeriods periods starting with the last edge, no matter, if it was rising or falling.

- AvgPeriods

The number of periods to average over. The value must be in the range from 1 up to 128. The higher the value, the lower is the influence of the jitter in the measurement result. Anyway, high value works as low-pass filter. Thus if the value is chosen too high, fast frequency changes cannot be detected.

- AvgCycles

The number of real time cycles to avarage over. The value must be in the range from 1 up to 128. The higher the value, the lower is the influence of the jitter in the measurement result. Anyway, high value works as low-pass filter. Thus if the value is chosen too high, fast frequency changes cannot be detected.

This value should be set to 1 or 2 but not higher than 5.

- ZeroDetectTime [s]

If no signal edges are detected for a duration higher than *ZeroDetectTime* seconds a frequency of 0 is returned. This value must not be higher than 20seconds to avoid measurement errors in consequence of overruns of the circular buffer.

Please be aware that small values can falsify measurement results.

The ZeroDetectTime must not be 0 for [MIO_M401_GetFrequency_CV\(\)](#).



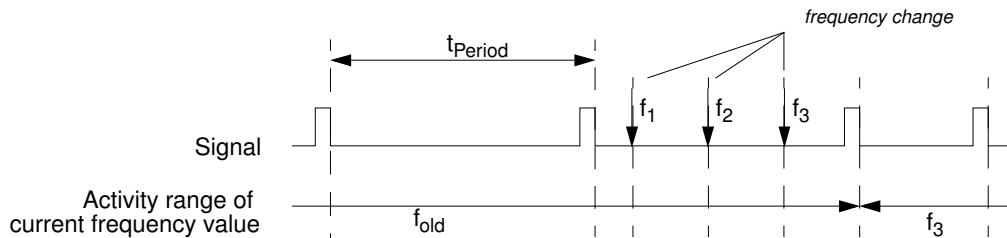
MIO_M401_SetOutPortMode()

```
void MIO_M401_SetOutPortMode (int Slot, int Unit, int OutPort, int ImmediateUpdate);
```

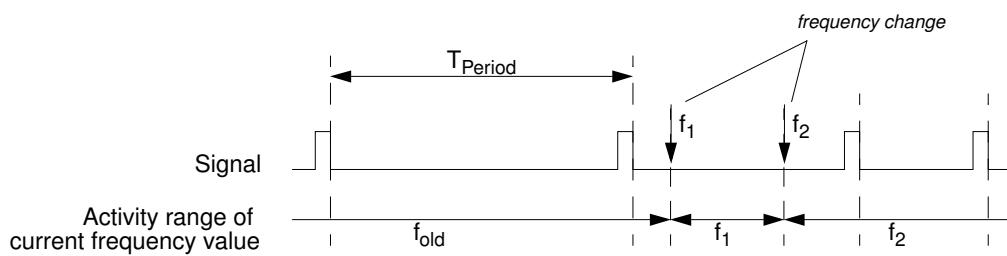
Description

With this function the frequency generator mode of each output port is configured. The module has 3 frequency units with 4 input ports and 2 output ports. The mode of each of the ports is configurable independently of the modes of other ports. However, in quadrature encoder mode only configuring the first output channel of the unit makes sense.

- ImmediateUpdate
0 - Frequency will be updated with the beginning of the next period.



- 1 - Frequency will be updated immediately



MIO_M401_SetOutPort()

```
void MIO_M401_SetOutPort (int Slot, int Unit, unsigned int Value);
```

Description

Set the output ports of the specified unit if the units output mode is M401_Binary. Bit 0 of Value is the output port 0 of the unit, bit 1 is the output port 1.

MIO_M401_SetFrequency()

```
void MIO_M401_SetFrequency (int Slot, int Unit, int OutPort, double Frequency);
```

Description

Set the frequency of one output port of the specified unit if the units output mode is M401_Frequency.

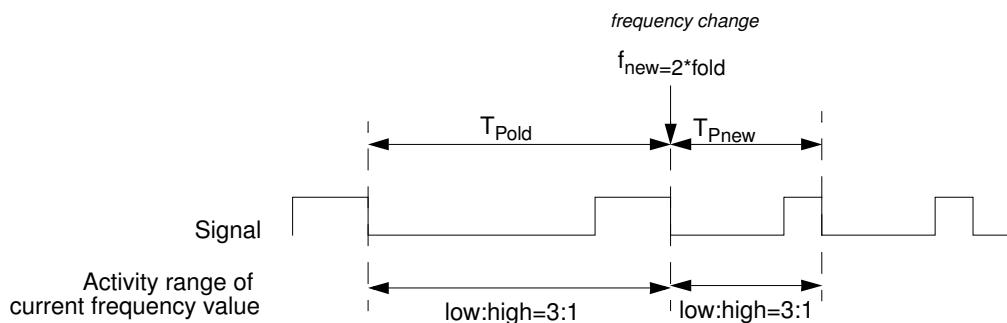
MIO_M401_SetDutyCycle()

```
void MIO_M401_SetDutyCycle (int Slot, int Unit, int OutPort, double DutyCycle);
```

Description

Set the duty cycle of one output port of the specified unit if the units output mode is M401_Frequency. The duty cycle is the relationship of the duration of the high level and the low level within on period. The relationship stays the same, even if the frequency is changed.

The range is 0.0 ... 1.0, where 0.0 means the output is all high and 1.0 means the output is all low.

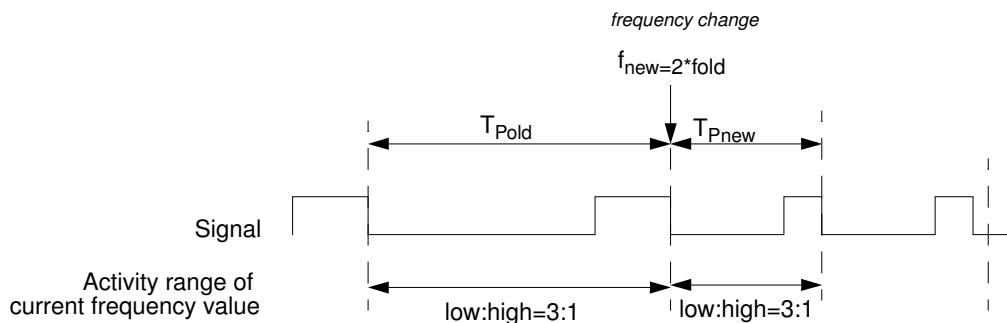


MIO_M401_SetPulseWidth()

```
void MIO_M401_SetPulseWidth (int Slot, int Unit, int OutPort, int FixPulseWidthLevel,
                             double PulseWidth);
```

Description

Here the duration of the low level or the high level as specified with `FixPulseWidthLevel` is fix. So the duration of the specified level stays the same even if the frequency changes while the other level's duration is adapted to the remaining cycle duration. Thus if the cycle duration is smaller than the specified level duration the signal output is low all the time. With the relative pulse width, the relationship between high and low stays the same, even if the frequency is changed.



MIO_M401_SetQuadEnc()

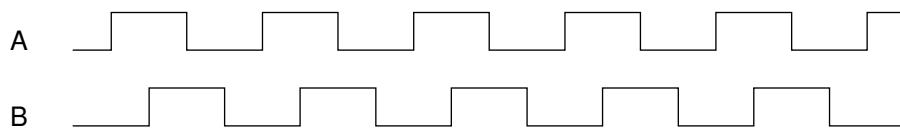
```
void MIO_M401_SetQuadEnc (int Slot, int Unit, double Frequency, int Direction);
```

Description

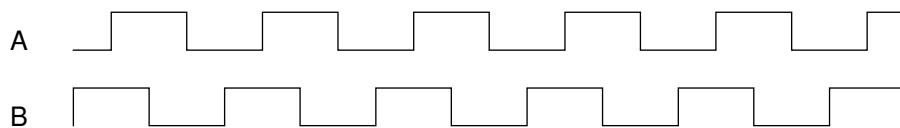
Set the frequency and the direction (=phase shift) of the quadrature encoder of the selected unit.

- Direction

0 - Turning Right, B legs 90deg after A



1 - Turning Left, B leads 90deg before A



MIO_M401_GetInPort()

```
unsigned int MIO_M401_GetInPort (int Slot, int Unit);
```

Description

Returns the current state of the input ports of the specified unit if the unit's input mode is M401_Binary. Bit 0 of the return value is the input port 0 of the unit, bit 3 is the input port 3.

MIO_M401_GetFrequency()

```
int MIO_M401_GetFrequency (int Slot, int Unit, int InPort, double *Frequency );
```

Description

This function returns the frequency of the specified input port of the selected unit.

This function only works correctly, if [MIO_M401_SetInPortMode\(\)](#) is used to configure the input ports. In the opposite to [MIO_M401_SetInPortMode_CV\(\)](#) this function always reads any needed timestamp directly from the M401 hardware and thus is not depending on the timestamps of earlier cycles. This function should be used if periodical calls with equidistant timing is not possible or if the duration of the function is not important.

Parameters

- Frequency

After returning from this function this parameter contains the calculated frequency value in [Hz].

Return Value

- -1 means that not at least 3 edges happened since the last reset and therefor it is not possible to calculate the frequency.
- 0 means that for a duration of more than `ZeroDetectTime` seconds no edges happened and thus the frequency is set to 0Hz.
- >0 reflects the number of periods, the frequency is averaged over.

Execution time

Depending on the configuration specified with `MIO_M401_SetInPortMode()` this function needs between 4 μ s (ZeroDetectTime==0 && TimeSlot==0) and up to about 15 μ s (ZeroDetectTime!=0 && TimeSlot!=0) for each channel.

MIO_M401_GetFrequency_CV()

```
void MIO_M401_GetFrequency_CV (int Slot, int PortMask, double* Frequency, int* Periods);
```

Description

`MIO_M401_GetFrequency_CV()` must only be called 1 time to return all the frequencies of the selected input ports of one M401. The desired channels are selected by the `PortMask`.

This function only works correctly, if `MIO_M401_SetInPortMode_CV()` is used to configure the input ports.

`MIO_M401_GetFrequency_CV()` reads the timestamp of the newest specified edge for the specified ports each time it is called. The values are stored within an internal circular buffer. Thus the function should be called only one time each real time cycle for each M401. To calculate the frequency, the function uses the current timestamp and the timestamp `<AvgCycles>` ago where `AvgCycles` is specified within `MIO_M401_SetInPortMode_CV()`.

If `AvgPeriods` is set to a value unequal “0” the function checks if more than `<AvgPeriods>` signal periods happened since the timestamp `<AvgCycles>` ago. If yes an additional access to the M401 is done to get the timestamp `<AvgPeriods>` signal periods ago and the frequency is calculated using this value and the current timestamp. This is done to avoid averaging over e.g. 1000 signal periods if the signal frequency is 1MHz and the real time cycle time is 1ms. With low frequencies (signal period `< cycle time * <AvgCycles>`) a minimum of 1 signal period is used to calculate the frequency.

Because the function only reads the newest time stamp, it must be called at least 2 times to return a real frequency value!

Parameters

- `PortMask`
is the mask of channel that should be read.
Bit0: Unit[0] InputPort[0]
Bit1: Unit[0] InputPort[1]
Bit2: Unit[0] InputPort[2]
Bit3: Unit[0] InputPort[3]
Bit4: Unit[1] InputPort[0]
and so on. Thus the lowest 12 bits are used for the mask.
- `Frequency`
A pointer to an array of doubles of the returned frequency values. The values are bottom justified. This means if only the channels 1 and 5 are selected by the `PortMask`, the value of channel 1 is returned in `Frequency[0]` and the value of channel 5 is returned in `Frequency[1]`.
- `Periods`
A pointer to an array of integer values of the number of periods the calculated frequency is averaged over. The values are bottom justified. This means if only the channels 1 and 5 are selected by the `PortMask`, the value of channel 1 is returned in `Period[0]` and the value of channel 5 is returned in `Period[1]`.
If not needed set the Value to “NULL”.

Execution time

The execution time is $t = 1.448\mu s * (1 + 2 * nChannels)$ if AvgPeriods is set to "0" at [MIO_M401_SetInPortMode_CV\(\)](#) else it has a maximum of $t = 1.448\mu s * (1 + 3 * nChannels)$ if AvgPeriods is set to a value unequal "0".

MIO_M401_GetDutyCycle()

```
int MIO_M401_GetDutyCycle (int Slot, int Unit, int InPort, double *DutyCycle);
```

Description

[MIO_M401_GetDutyCycle\(\)](#) returns the relationship of the duration of the low level and the high level within the last period of the specified input port of the selected unit.

Parameters

- DutyCycle

After returning from this function this parameter contains the calculated duty cycle value as a range between 0.0 ... 1.0. 0.0 means the input signal was all high, 1.0 means the input signal was all low.

Return Value

- 1 means that not at least 3 edges happened since the last reset and therefore it is not possible to calculate the duty cycle.
- 0 means that the duty cycle is calculated correctly.

MIO_M401_GetQuadDec()

```
void MIO_M401_GetQuadDec (int Slot, int Unit, tM401_QuadDec *QuadDecSignal);
```

Description

This function returns the angle, the frequency and the direction of the selected unit if the unit's input mode is set to [M401_Quadrature](#).

QuadDecSignal is a pointer to a struct, in which the results of the function are returned. The structure of [tM401_QuadDec](#) is shown below.

[MIO_M401_GetQuadDec\(\)](#) uses [MIO_M401_GetFrequency\(\)](#) to calculate the Frequency!

[MIO_M401_GetQuadDec\(\)](#) should no longer be used because it reads all the data about the quadrature decoding channels, even unneeded values and therefore wastes execution time.

```
struct {
    unsigned int Angle;
    unsigned int AngleZero;
    unsigned int DirChange;
    unsigned int Direction;
    double Frequency;
} tM401_QuadDec;
```

- Angle
Absolute Angle since the module was configured / reset
- AngleZero
Absolute Angle since last reference signal (signal "0")

- **DirChange**
This is the level of the edge counter of signal "A" when the direction changed for the last time
- **Direction**
Current Direction of the quadrature decoder, 0 - turning right, 1 - turning left
- **Frequency**
Is the Frequency of the signal "A" of the quadrature decoder.

MIO_M401_GetDirection()

```
unsigned int MIO_M401_GetDirection (int Slot, int Unit);
```

Description

Returns the current direction of the quadrature decoder.

0 - turning right

1 - turning left

MIO_M401_GetAngle()

```
unsigned int MIO_M401_GetAngle (int Slot, unsigned int Unit);
```

Description

Returns the absolute angle of the quadrature decoder.

MIO_M401_GetAngleZero()

```
unsigned int MIO_M401_GetAngleZero (int Slot, int Unit);
```

Description

Returns the absolute Angle since last reference signal (signal "0").

MIO_M401_GetDirectionChange()

```
unsigned int MIO_M401_GetDirectionChange (int Slot, int Unit);
```

Description

Returns the level of the edge counter of signal "A" when the direction changed for the last time.

MIO_M401_GetEdges()

```
int MIO_M401_GetEdges (int Slot, int Unit, int InPort);
```

Description

[MIO_M401_GetEdges\(\)](#) returns the content of the 32bit-signal-edge-counter of the specified input port of the selected unit.

MIO_M401_GetPeriods()

```
int MIO_M401_GetPeriods (int Slot, int Unit, int InPort);
```

Description

[MIO_M401_GetPeriods\(\)](#) returns the number of periods happened on the specified input port of the selected unit since the module is configured or [MIO_ResetModules\(\)](#) was called. The number of periods is the number of edges divided by 2. The return value of this function can be used to set the Parameter Index of [MIO_M401_GetTimeStamp\(\)](#).

MIO_M401_GetTimeStamp()

```
int MIO_M401_GetTimeStamp (int Slot, int Unit, int InPort, unsigned int Index, int Rising);
```

Description

This function can be used to read any rising or falling edge from the circular buffer of the specified input port of the selected unit.

Parameters

- **Index**
This is the index of the period from which the timestamp should be returned. The index of the last period is returned by [MIO_M401_GetPeriods\(\)](#). Because of the size of the circular buffer, the last 254 periods can be indexed.
- **Rising**
0 - Return the timestamp of the falling edge of the period defined by index.
1- Return the timestamp of the rising edge of the period defined by index.

Return Value

The function returns a 32bit value that reflects the content of the time base counter when the corresponding edge happened. To convert the value into a time value it must be multiplied by 0.000,000,01 seconds. Be aware that the time base is a 32bit-value which runs over every 42 seconds. Thus if the duration between the desired timestamps is longer than 42 seconds the result would be ambiguous.

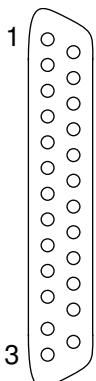
Example

Get the duration between the rising edge of the current period and the falling edge of the previous period.

```
Index = MIO_M401_GetPeriods (0 /* Slot 0 */, 0 /* Unit 0 */, 0 /* InPort 0 */);
t1 = MIO_M401_GetTimeStamp (0, 0, 0, Index, 1 /* Rising edge */);
t2 = MIO_M401_GetTimeStamp (0, 0, 0, Index-1, 0/* Falling edge */);
Duration = (t1-t2)*0.00000001;
```

10.4.15 M402: Engine Signal Generator, 8 Channels

Connector assignment M402



Pin	Signal Description	Pin	Signal Description
1	Crank/Cam0	14	AGND
2	Crank/Cam1	15	AGND
3	Crank/Cam2	16	AGND
4	Crank/Cam3	17	AGND
5	Knock0	18	AGND
6	Knock1	19	AGND
7	Knock2	20	AGND
8	Knock3	21	AGND
9	n.c.	22	AGND
10	n.c.	23	AGND
11	n.c.	24	AGND
12	-15V	25	AGND
13	+15V		

Function Overview

Structs and Enums

- `tM402_Waveform {}`
- `tM402CfgBin {}`

Utility Functions

- `MIO_M402_GenerateSampleWaveform()`
- `MIO_M402_GenerateKnockWaveform()`
- `MIO_M402_GenerateNoiseWaveform()`

Initialization and Configuration

- `MIO_M402_Config()`
- `MIO_M402_SetWaveform()`
- `MIO_M402_SetEngineStartAngle()`
- `MIO_M402_CCChannelConfig()`
- `MIO_M402_CCBinaryConfig()`
- `MIO_M402_CCBinarySetTriggerAngles()`
- `MIO_M402_ConfigureAPU()` (only for unsynchronized usage)
- `MIO_M402M403_Synchronize()` (only for synchronized usage)
- `MIO_M402_EnableChannels()`
- `MIO_M402_KnockSetTriggerAngle()`
- `MIO_M402_KnockSensorEventConfig()`
- `MIO_M402_KnockSensorNoiseEnable()`

Runtime Functions

- `MIO_M402_SetRotationSpeed()` (only for unsynchronized usage)

- [MIO_M402M403_SyncSetRotationSpeed\(\)](#) (only for synchronized usage)
- [MIO_M402_SetPhaseShift\(\)](#)
- [MIO_M402_GetBaseAngle\(\)](#)
- [MIO_M402_EnableKnocks\(\)](#)
- [MIO_M402_KnockSensorNoiseGain\(\)](#)

Example

- [Configuration Example](#)

tM402_Waveform {}

```
tM402_Waveform {
    M402_Square=1,
    M402_Sine,
    M402_Cosine,
    M402_Triangle
} tM402_Waveform;
```

Description

This enum is used for the function [MIO_M402_GenerateSampleWaveform\(\)](#) to define the kind of the waveform to be generated.

tM402CfgBin {}

```
typedef struct tM402CfgBin {
    double PulseWidth; /* sec */
    double PhaseDelay; /* rad */
    double TimeDelay; /* sec */
} tM402CfgBin;
```

Description

[tM402CfgBin {}](#) is used inside the parameter list of [MIO_M402_CCBinaryConfig\(\)](#) to configure the Binary Hallsensor Mode of the M402. The Binary Hallsensor Mode is available since revision 1.7.

MIO_M402_GenerateSampleWaveform()

```
void MIO_M402_GenerateSampleWaveform (tM402_Waveform WF, float Umax, float Umin,
                                         float PPRatio, int nTeeth, int MissingStart,
                                         int nMissing, float *Buf, int BufSize);
```

Description

This function generates several wave forms for the M402 or M404 which can be directly downloaded into the modules by using [MIO_M402_SetWaveform\(\)](#). Thereby also unsymmetrical wave forms, e.g. due to missing teeth, are possible. This function is a simple way to generate some often needed wave forms. Of course the user also can define his own function without using this function.

Parameters

- WF
This is the kind of the waveform. The values must be taken from the enumerator [tM402_Waveform {}](#)
- Umax, Umin
The maximum and minimum voltage values of the waveform. If Umin > Umax the result waveform is inverted.
- PPRatio
Pulse Pause Relationship. Usually set to 0.5. 1.0 means the value is Umax all the Time, 0.0 means the value is Umin all the Time.
- nTeeth
The number of teeth/periods within the buffer including the missing ones.

- MissingStart
The number of the first missing tooth. The first tooth has the number 0.
- nMissing
The number of the missing teeth at all.
- Buf
For the M402 and the M404 this must be an array of 65536 float values.
- BufSize
For the M402 and the M404 the BufSize must be 65536

Example

A 60-2Z sine wave should be generated. The teeth #58 and #59 are missing. The minimum voltage is 0V and the maximum voltage is 7.5V.

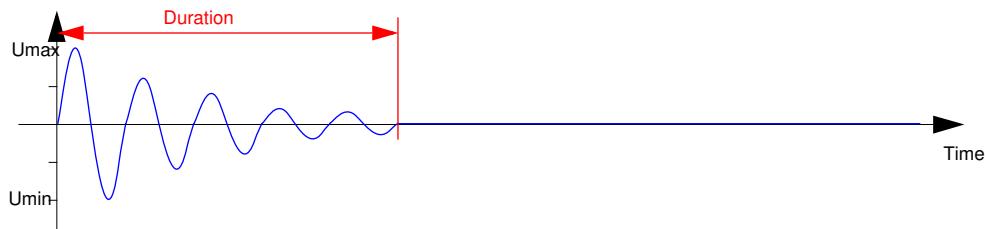
```
float buf[65536];
MIO_M402_GenerateSampleWaveform(M402_Sine, 7.5, 0.0, 0.5, 60, 58, 2, buf, 65536);
```

MIO_M402_GenerateKnockWaveform()

```
void MIO_M402_GenerateKnockWaveform (float Umax, float Umin,
                                      float Frequency, float Duration,
                                      float *Buf, int BufSize);#
```

Description

This function generates a typical knock waveform as shown below.



$$U(t) = U_0 + A \cdot e^{\left(\frac{-2 \cdot \pi \cdot f \cdot t}{Duration}\right)} \sin(2 \cdot \pi \cdot f \cdot t)$$

Parameters

- Umax, Umin
The maximum and minimum voltage values of the waveform. If Umin > Umax the result waveform is inverted.
- Frequency
The frequency of the knock signal. A typical knock frequency is between 5kHz and 20kHz.
- Duration
The time until the knocking has completely faded out in microseconds.
- Buf
For the M402 and the M404 this must be an array of 65536 float values.

- BufSize
For the M402 and the M404 the BufSize must be 65536

MIO_M402_GenerateNoiseWaveform()

```
void MIO_M402_GenerateNoiseWaveform (float Umax, float Umin, float *Buf, int BufSize);
```

Description

This function generates a noise with random values between Umax and Umin.

Parameters

- Umax, Umin
The maximum and minimum voltage values of the waveform. If Umin > Umax the result waveform is inverted.
- Buf
For the M402 and the M404 this must be an array of 65536 float values.
- BufSize
For the M402 and the M404 the BufSize must be 65536

MIO_M402_Config()

```
int MIO_M402_Config (int Slot);
```

Description

By calling the function [MIO_M402_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M402 is specified. This function must be called one time, before any attempted access to the module.

Initial state of the module:

- No waveforms are defined
- All gain values are 1.0 (0x1000)
- The first weighting register of each channel is set to 1.0 (0x1000) while any other weighting value is set to 0 (0x0000)
- No knock events are defined
- The Angular Processing Unit is disabled and set as master
- The rotation speed is set to “0”
- No synchronization is configured

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

The module is set back to its initial state.

MIO_M402_SetWaveform()

```
void MIO_M402_SetWaveform (int Slot, int BufferNo, float *Waveform0, float *Waveform1,  
                           float *Waveform2, float *Waveform3 );
```

Description

This function loads the waveforms from the buffers into the M402.

Parameters

- BufferNo
 - 0 - Waveform for is written the buffer of crank/cam channel 0
 - 1 - Waveform for is written the buffer of crank/cam channel 1
 - 2 - Waveform for is written the buffer of crank/cam channel 2
 - 3 - Waveform for is written the buffer of crank/cam channel 3
 - 4 - Waveform for is written the buffer of the knock signals
 - 5 - Waveform for is written the buffer of the noise signal.
- Waveform0, Waveform1, Waveform2, Waveform3
 - For the 4 crank/cam-Channels up to 4 waveforms can be used for each channel. The knock channels together have four waveforms. The noise channel has only 1 waveform. Therefore writing values to Waveform1, Waveform2 and Waveform3 does not make sense for the noise channel. If waveforms are unused, the parameter should be NULL.

MIO_M402_SetEngineStartAngle()

```
void MIO_M402_SetEngineStartAngle (int Slot, double Angle);
```

Description

With this function a predefined start angle of the crankshaft can be defined. This probably is useful to don't start the engine at the upper dead center (UDC) at the first time the module is used after switching it on.

Parameters

- Angle
 - Engine start angle. The unit is rad.

MIO_M402_CCChannelConfig()

```
void MIO_M402_CCChannelConfig (int Slot, int channel,
                               int Prescale, float PhaseSR,
                               float Weighting0, float Weighting1,
                               float Weighting2, float Weighting3,
                               float Gain);
```

Description

With this function the crankshaft and camshaft channels are configured.

Parameters

- Prescale
 - Must be set to "1" for camshaft signals and to "0" for crankshaft signals because the camshaft turns half the speed of the crankshaft.
- PhaseSR
 - This parameter specifies the change in phaseshift of camshafts per ms (if it changes). Unit is rad/ms.

Value range: 0.0 .. 73.6 rad/ms (0.0..4218deg/ms)

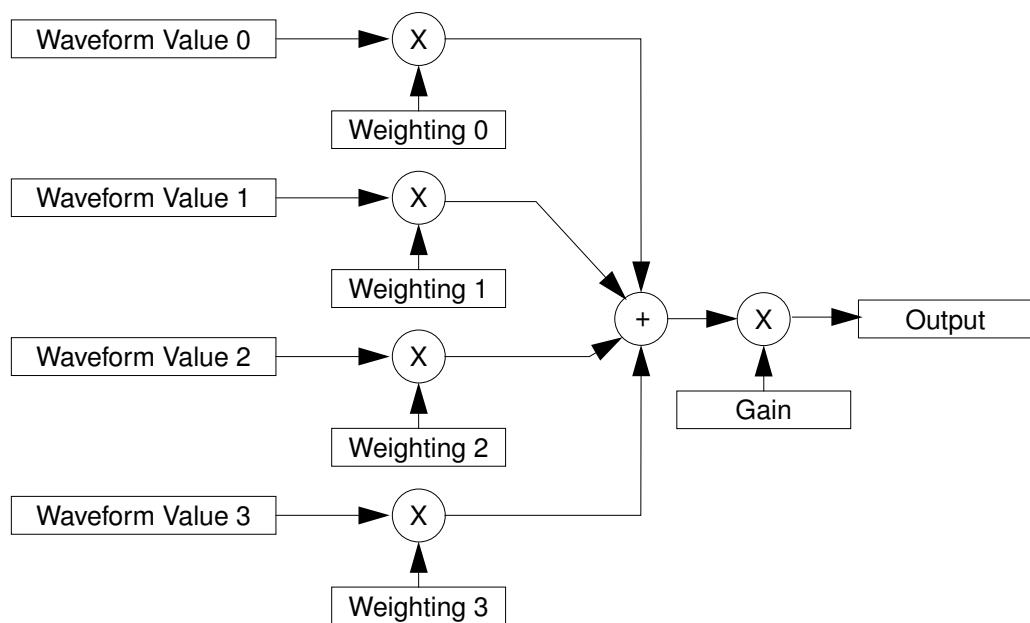
Resolution: 1.125×10^{-3} rad/ms (0.0645deg/ms)

If PhaseSR is set to “0.0” the phaseshift is adapted immediately.

- Weighting0, Weighting1, Weighting2, Weighting3, Gain

For each of the 4 waveforms it is configurable how strong they are considered for the output voltage. Therefore each waveform is multiplied with the corresponding weighting value.

The value range of the weighting is between 0.0 and 1.0 where 0.0 means the current waveform is not considered for the calculation of the current output value and 1.0 means that the value is considered with 100%. Then all weighted waveform values are added together and then are multiplied by the gain. The value range of the gain is 0 to 3.9.



MIO_M402_CCBinaryConfig()

```
void MIO_M402_CCBinaryConfig (int Slot, int Channel,
                             int Prescale, float PhaseSR,
                             int nTeeth,
                             tM402CfgBin BinForward, tM402CfgBin BinBackward,
                             float VoltageActive, float VoltageInactive);
```

Description

With this function the crankshaft and camshaft channels are configured in the Analog Hallsensor Mode.

Parameters

- Prescale
Must be set to “1” for camshaft signals and to “0” for crankshaft signals because the camshaft turns half the speed of the crankshaft.
- PhaseSR
This parameter specifies the change in phaseshift of camshafts per ms (if it changes). Unit is rad/ms.

- **nTeeth**
nTeeth is the real number of pulses to be generated with one turn of the crank/cam.
This value must be the same as the number of angles configured by
[MIO_M402_CCBinarySetTriggerAngles\(\)](#).
For example if a 60-2Z-signal is used, set this value to 58!
- **BinForward, BinBackward**
These parameters are from type [tM402CfgBin {}](#).
 - **PulseWidth** is the duration of the pulse. It's unit is seconds.
 - **PhaseDelay** delayes the pulse-generation by some degrees. It's unit is rad. Positive as well as negative values are possible for the Phase Delay
 - **TimeDelay** delayes the pulse-generation by some seconds. The unit is seconds.
 - As the following opicture shows, first the PhaseDelay and thereafter the TimeDelay is considered.

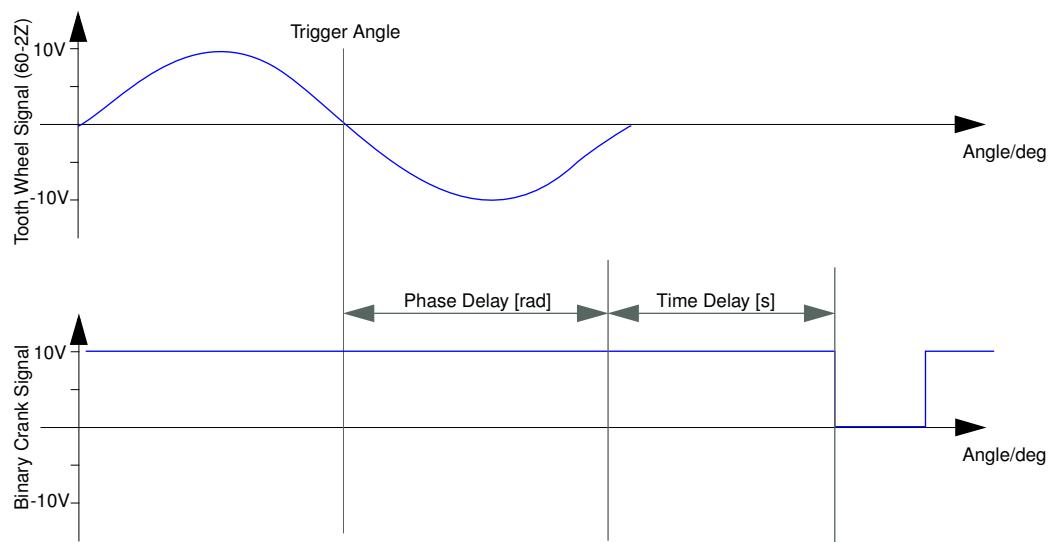


Figure 10.4: Delayed Pulse Generation (example)

- **VoltageActive**
This is the signal voltage as long as the pulse is active.
- **VoltageInactive**
This is the signal voltage as long as the pulse is NOT active.

MIO_M402_CCBinarySetTriggerAngles()

```
void MIO_M402_CCBinarySetTriggerAngles (int Slot, int Channel,
                                         int AngleID, float Angle /*rad*/);
```

Description

With this function the crankshaft and camshaft channels are configured.

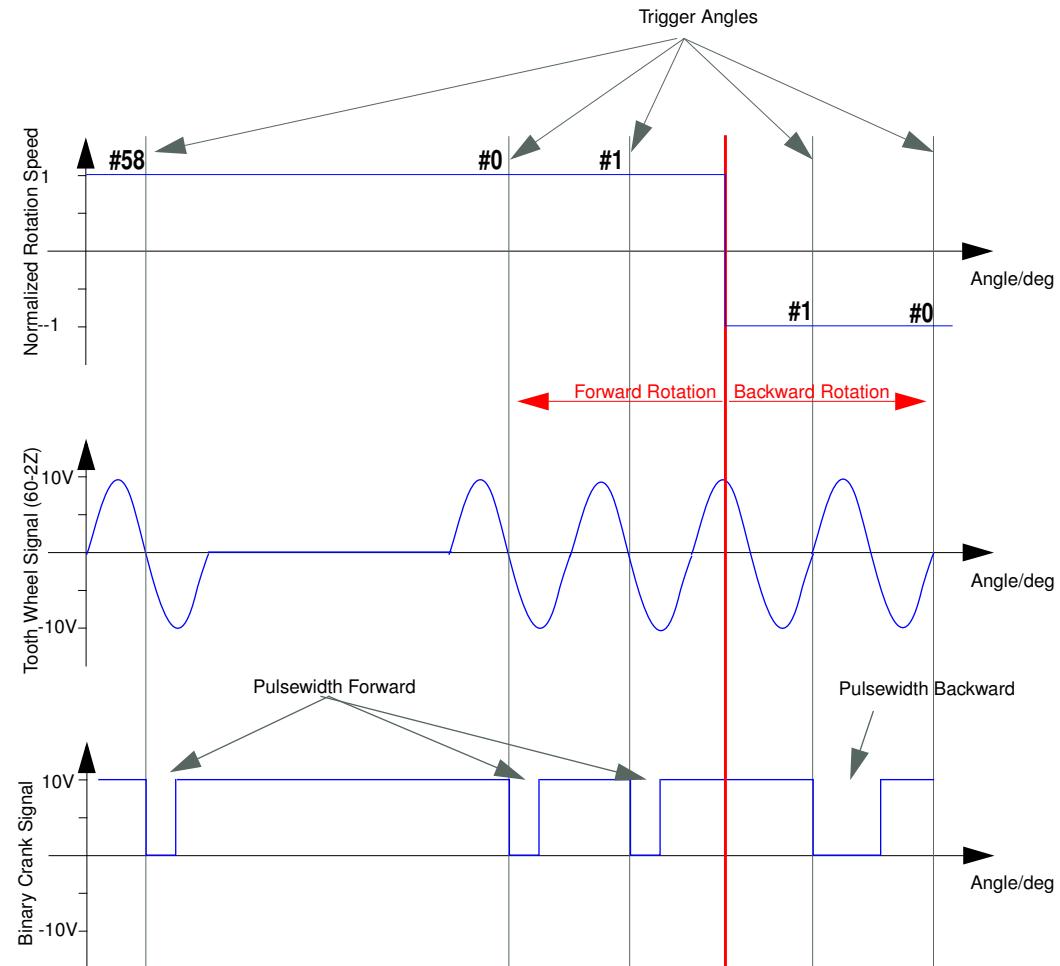


Figure 10.5: Binary Output Waveforms (60-2Z example)

Parameters

- **AngleID**
The AngleID is the index of the angle to be configured. It can also be used to re-configure an angle.
- **Angle**
This is the angle where a pulse is generated.

Example:

```
tM402CfgBin fw = {0.000054, 0, 0}; /* Forward-Pulse = 54us, no delay*/
tM402CfgBin bw = {0.000108, 0, 0}; /* Backward-Pulse = 108us, no delay */
int i, nTeeth=60, nMissing=2;
MIO_M402_CCBinaryConfig (Slot_M402, 0, 0, 0,
                         nTeeth - nMissing,
                         fw, bw,
                         0.0 /* Active Voltage */,
                         5.0 /* Inactive Voltage */);
for (i = nMissing; i < nTeeth; i++) {
    float angle = i * (2 * M_PI) / nTeeth; /*rad*/
    MIO_M402_CCBinarySetTriggerAngles (Slot_M402, 0, i-nMissing, angle );
}
```

MIO_M402_ConfigureAPU()

```
void MIO_M402_ConfigureAPU (int Slot, int Enable, int IsMaster);
```

Description

This function enables the calculation of the angular and defines, if the APU is independent from other M402/M403-APUs.

If several M402/M403 modules should be synchronized, don't use this function. Use [MIO_M402M403_Synchronize\(\)](#) instead.

Parameters

- **Enable**
Enables and disables the calculation of the angular.
- **IsMaster**
If the module is not synchronized with other modules set this parameter to "1".
If IsMaster is set to "0" the address calculation stops after one turn and waits for the synchronisation trigger of another module.

MIO_M402_EnableChannels()

```
void MIO_M402_EnableChannels (int Slot, int EnableMask, int ModeMask);
```

Description

This function enables the output DACs of the channels and configures the mode of the crank/cam channels to either binary or analog mode.

Parameters

- **EnableMask**
This parameter is a bitmask for enabling the output channels.
 - bit0 enables the crank/cam channel0
 - bit1 enables the crank/cam channel1
 - bit2 enables the crank/cam channel2
 - bit3 enables the crank/cam channel3
 - bit4 enables the knock channel0
 - bit5 enables the knock channel1
 - bit6 enables the knock channel2
 - bit7 enables the knock channel3
- **ModeMask**
If the new Binary HallSensor Mode should be used for a channel, set this bit to "1", otherwise if the out-dated Analog Hallsensor Mode is used, set it to "0".
 - bit0 configures the crank/cam channel0
 - bit1 configures the crank/cam channel1
 - bit2 configures the crank/cam channel2
 - bit3 configures the crank/cam channel3

MIO_M402_KnockSetTriggerAngle()

```
void MIO_M402_KnockSetTriggerAngle (int Slot, int EventNo, double Angle, int WaveFormNo);
```

Description

This function configures the start angle and the waveform of a knock event. Up to 12 knock events (=cylinder). For each cylinder one of the 4 knock waveforms is selectable.

The function must be called for every cylinder.

Parameters

- **EventNo**
EventNo is the same as cylinder number. Value range: 0..11.
- **Angle**
This is the angle in rad of the crankshaft where the knocking of the cylinder starts. Value range: 0..4PI (0..720deg)
- **WaveFormNo**
As shown at the description of [MIO_M402_SetWaveform\(\)](#) you can load up to 4 different knocking waveforms into the M402. With Waveform No, you select one of those 4 waveforms for each cylinder.

Example

```
int cyl;  
/* 12 cylinders knock evenly distributed over 720deg (4PI) */  
for (cyl=0;cyl<12; cyl++) {  
    MIO_M402_KnockSetTriggerAngle (Slot_M402, cyl, 4 * M_PI * cyl / 12, 0);  
}  
•
```

MIO_M402_KnockSensorEventConfig()

```
void MIO_M402_KnockSensorEventConfig (int Slot, int Sensor, int EventNo, int Enabled,  
                                      float Gain);
```

Description

This function configures which knock sensor can “hear” which cylinders knock and if so, how “loud” it can “hear” it.

This makes it possible to simulate that the knock sensors are tightened at different cylinder banks where the knock sensors of the one bank don’t “hear” the knocking cylinders of the other bank. Also the different distances between the cylinders and the several knock sensors can be simulated.

The function [MIO_M402_KnockSensorEventConfig\(\)](#) must be called for each knock sensor which should be simulated and for each cylinder of the engine.

By default the knocking for all 4 sensors and all 12 cylinders is disabled.

Parameters

- **Sensor**
The number of the knock sensor (0..3).
- **EventNo**
The number of the cylinder (0..11).

- Enabled
“1” if the specified sensor “hears” the specified cylinder, “0” if not.
- Gain
The amplification of the knock signal of the specified cylinder for the specified sensor.
This defines, how “loud” the sensor can “hear” the knocking of the cylinder. (0.0 .. 1.0)

Example

```
int cyl, s;
for (cyl=0;cyl<12; cyl++) { /* 12 cylinders */
    for (s=0;s<4;s++) { /* 4 sensors */
        MIO_M402_KnockSensorEventConfig (Slot_M402, s, cyl, 1, 1.0);
    }
}
```

MIO_M402_KnockSensorNoiseEnable()

```
void MIO_M402_KnockSensorNoiseEnable (int Slot, int Sensor, int EnableNoise);
```

Description

This function enables/disables the noise for the knock sensors. If noise is enabled the content of the noise waveform is added to the output of the specified knock sensor.

Parameters

- Sensor
The number of the knock sensor (0..3).
- EnableNoise
“1” if noise is enabled for the specified sensor, “0” if not.

MIO_M402_SetRotationSpeed()

```
void MIO_M402_SetRotationSpeed (int Slot, double Frequency);
```

Description

This function specifies the rotation speed of the engine.

If several M402/M403 modules are synchronized, use

[MIO_M402M403_SyncSetRotationSpeed\(\)](#) **to set the engine rotation speed instead of** [MIO_M402_SetRotationSpeed\(\)](#).

Parameters

- Frequency
This is the rotation speed of the engine in rounds per second. The value might be positive or negative.

MIO_M402_SetPhaseShift()

```
void MIO_M402_SetPhaseShift (int Slot, int Channel, float Phase);
```

Description

This function defines the phase shift of each crank/cam channel of the module compared to the base angle of the module.

Parameters

- Phase
Value Range: -4PI4PI
Negative values mean that the specified channel trails the crank/cam0 channel.
Positive values mean that the specified channel leads the crank/cam0 channel.

MIO_M402_GetBaseAngle()

```
double MIO_M402_GetBaseAngle (int Slot);
```

Description

This function returns the base angle of the module.

Please notice that the base angle is the current angle of the angular processing unit. The angle of each crank/cam channel of the module can differ from the angle returned by this function due to a phase shift defined by [MIO_M402_SetPhaseShift\(\)](#).

MIO_M402_EnableKnocks()

```
void MIO_M402_EnableKnocks (int Slot, int KnockEnableMask);
```

Description

With [MIO_M402_EnableKnocks\(\)](#) it is specified, which cylinder is knocking at the moment. This function is used to enable/disable knocking for all cylinders at once. See also [MIO_M402_KnockSensorEventConfig\(\)](#) to configure which sensor “hears” which cylinders knocking.

Parameters

- KnockEnableMask
This parameter is a bitmask for enabling the knocking.
 - bit0 enables/disables knocking of cylinder 0
 - bit1 enables/disables knocking of cylinder 1
 - bit2 enables/disables knocking of cylinder 2
 - bit3 enables/disables knocking of cylinder 3
 - bit4 enables/disables knocking of cylinder 4
 - bit5 enables/disables knocking of cylinder 5
 - bit6 enables/disables knocking of cylinder 6
 - bit7 enables/disables knocking of cylinder 7
 - bit8 enables/disables knocking of cylinder 8
 - bit9 enables/disables knocking of cylinder 9
 - bit10 enables/disables knocking of cylinder 10
 - bit11 enables/disables knocking of cylinder 11

MIO_M402_KnockSensorNoiseGain()

```
void MIO_M402_KnockSensorNoiseEnable (int Slot, int Sensor, int NoiseGain);
```

Description

This function configures how load the specified sensor can hear the noise if noise is enabled. The content of the noise waveform multiplied by gain and is added to the output of the specified knock sensor.

Parameters

- **Sensor**
The number of the knock sensor (0..3).
- **NoiseGain**
The value range of the gain is between 0.0 and 1.0 where 0.0 means the current noise waveform is not considered for the calculation of the current output value and 1.0 means that the value is considered with 100%.

Examples

Configuration Example

```
#define ENGINE_BUFSIZE 65536

int IO_Init (void) /* RTMaker: IO_Start */
{
    int s /* sensor */, c /* channel */;
    float buf[4][65536];
    tM402CfgBin fw = {0.000054, 0, 0}, bw = {0.000108, 0, 0};
    int nTeeth=60, nMissing=2;
    int i;

    if (MIO_M402_Config (Slot_EngineOut) < 0) return RT_Failure;

    /* Configure Crankshaft: 60-2Z, Sine, 0..5V, Binary Hallsensor Mode, 5us delay in both
     directions*/
    MIO_M402_CCBinaryConfig (Slot_M402, 0, 0, 0, nTeeth-nMissing,
                             fw, bw, 0.0 /* Active Voltage */, 5.0 /* Inactive Voltage */);
    for (i = nMissing; i < nTeeth; i++) {
        float angle = i * (2 * M_PI) / nTeeth;
        MIO_M402_CCBinarySetTriggerAngles (Slot_M402, 0, i-nMissing, angle /*rad*/);
    }

    /* Configure CAM Shaft 0 and 1: 1 Period, SquareWave, 0..5V */
    MIO_M402_GenerateSampleWaveform (
        M402_Square, /* tM402_Waveform WF = Kind of waveform, defined in mio.h */
        5.0, /*Umax*/0.0, /*Umin*/
        0.5, /*Duty Cycle*/
        1, /*nTeeth*/
        0, /*The first tooth that is missing*/
        0, /*nMissing Teeth*/
        buf[0], ENGINE_BUFSIZE);
    for (c=1; c<3; c++) {
        MIO_M402_SetWaveform (Slot_EngineOut, c /* CAM Channel */, buf[0], NULL, NULL, NULL);
        MIO_M402_SetPhaseShift (Slot_EngineOut, c, c * 1.62); /* c*PI/2 = c* +90 deg*/
        MIO_M402_CCChannelConfig (Slot_EngineOut, c /* CAM Channel */,
                                  1, /* Prescaler=1 means half the rotation speed of the base angle */
                                  0, /* Change speed of phaseshift (if it changes) rad/ms */
                                  1.0, /* Weighting of buffer 0 */

```

```
    0.0, /* Weighting of buffer 1 */
    0.0, /* Weighting of buffer 2 */
    0.0, /* Weighting of buffer 3 */
    1.0); /* Gain of the intermixed signal */
}

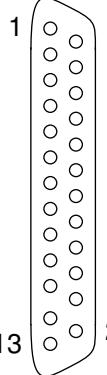
/* ... */

/* Knock Signals */
MIO_M402_GenerateKnockWaveform (
    5.0, /*Umax*/
    0.0, /*Umin*/
    5000.0, /* Frequency = 5000Hz*/
    1000, /* Duration in Micro seconds */
    buf[0], ENGINE_BUFSIZE);
MIO_M402_SetWaveform (Slot_EngineOut, 4 /* = Knock Channel 0 */, buf[0], NULL, NULL, NULL);

/* Noise */
MIO_M402_GenerateNoiseWaveform (0.25/*Umax*/, -0.25/*Umin*/, buf[0], 65536);
MIO_M402_SetWaveform (Slot_EngineOut, 5 /* Noise */, buf[0], NULL, NULL, NULL);
/* Enable / Disable Knocking and noise */
for (c=0; c<4; c++) {
    MIO_M402_KnockSensorNoiseEnable (Slot_EngineOut, c, 0);
    MIO_M402_KnockSensorNoiseGain (Slot_EngineOut, c, 1.0);
}
MIO_M402_EnableKnocks (Slot_EngineOut, 0);
for (c = 0;c < 12; c++) {
/* Set Start Angle of Knocking for each cylinder */
    MIO_M402_KnockSetTriggerAngle (Slot_EngineOut, e, 4 * M_PI * e / 12, 0);
    for (s=0; s<4; s++) {
        /* Enable the knocking for all the sensors and the cylinders */
        MIO_M402_KnockSensorEventConfig (Slot_EngineOut,
            s /* sensor*/,
            e /* zylinder*/,
            0 /* disabled*/,
            1.0 /* Gain*/);
    }
}
MIO_M402_EnableChannels (Slot_EngineOut, 0x1f);
}
```

10.4.16 M403: Engine Signal Detector, 8 Channels

Connector assignment



Pin	Signal Description	Pin	Signal Description
1	Input channel 0 (+)	14	Input channel 0 (-)
2	Input channel 1 (+)	15	Input channel 1 (-)
3	n.c.	16	n.c.
4	Input channel 2 (+)	17	Input channel 2 (-)
5	Input channel 3 (+)	18	Input channel 3 (-)
6	n.c.	19	n.c.
7	Input channel 4 (+)	20	Input channel 4 (-)
8	Input channel 5 (+)	21	Input channel 5 (-)
9	n.c.	22	n.c.
10	Input channel 6 (+)	23	Input channel 6 (-)
11	Input channel 7(+)	24	Input channel 7 (-)
12	n.c.	25	n.c.
13	GND		

Function Overview

Structs and Enums

- [tMIO_M403_WindowState](#)
- [tMIO_M403_WindowData](#)
- [eMIO_M403_WindowData](#)

Initialization and Configuration

- [MIO_M403_Config\(\)](#)
- [MIO_M403_SetEngineStartAngle\(\)](#)
- [MIO_M403_ConfigureAPU\(\)](#) (only for unsynchronized usage)
- [MIO_M402M403_Synchronize\(\)](#) (only for synchronized usage)
- [MIO_M403_SetThresholdVoltage\(\)](#)
- [MIO_M403_SetCaptureWindow\(\)](#)
- [MIO_M403_SetReferenceAngle\(\)](#)

Runtime Functions

- [MIO_M403_SetRotationSpeed\(\)](#) (only for unsynchronized usage)
- [MIO_M402M403_SyncSetRotationSpeed\(\)](#) (only for synchronized usage)
- [MIO_M403_GetBaseAngle\(\)](#)
- [MIO_M403_GetVoltage\(\)](#)
- [MIO_M403_GetnEvents\(\)](#)
- [MIO_M403_GetEvStartAngle\(\)](#)
- [MIO_M403_GetEvEndAngle\(\)](#)
- [MIO_M403_GetEvAngles\(\)](#)
- [MIO_M403_GetEvStartTime\(\)](#)

- [MIO_M403_GetEvEndTime\(\)](#)
- [MIO_M403_GetEvDuration\(\)](#)
- [MIO_M403_GetWindowState\(\)](#)
- [MIO_M403_GetWindowData\(\)](#)

tMIO_M403_WindowState

```
typedef struct tMIO_M403_WindowState {  
    int IsActive;  
    int IsEnabled;  
    int Counter;  
    int IndexFirst;  
    int nEvents;  
} tMIO_M403_WindowState;
```

Description

Use a variable of this type as the return value of [MIO_M403_SetCaptureWindow\(\)](#). The struct contains information about the Window:

Struct Members

- **IsActive**
Is set to "1" if the angle is within the window.
- **IsEnabled**
Is set to "1" if the window is enabled.
- **Counter**
This is the activation counter of the window. It is incremented by 1 each time the window is left. This can be used to check if the window was left once again since the last access.
- **IndexFirst**
The `IndexFirst` is the index of the first event within the circular event buffer of the channel that happened when the window was active the last time.
- **nEvents**
`nEvents` is the number of events that happened when the window was active the last time.

tMIO_M403_WindowData

```
typedef struct tMIO_M403_WindowData {  
    double StartAngle;  
    double EndAngle;  
    int nCycles;  
    double Duration;  
    double StartTime;  
    double EndTime;  
} tMIO_M403_WindowData;
```

Description

Use an array of this type with [MIO_M403_GetWindowData\(\)](#) to return the event data of all events that happened during a window was active.

Struct Members

- **StartAngle**
The start angle of an event related to the reference angle.
- **EndAngle**
The end angle of an event related to the reference angle.

- nCycles
The number of cycles the event kept active.
- Duration
The duration the event kept active in seconds.
- StartTime
The start time of the event related to the internal 32bit time base. Can be used to calculate time based relationships between events.
- EndTime
The end time of the event related to the internal 32bit time base. Can be used to calculate time based relationships between events.

eMIO_M403_WindowData

```
enum {
    MIO_M403_StartAngle = 0x01,
    MIO_M403_EndAngle = 0x02,
    MIO_M403_StartEndAngle = 0x03,
    MIO_M403_StartTime = 0x10,
    MIO_M403_EndTime = 0x20,
    MIO_M403_Duration = 0x30
} eMIO_M403_WindowData;
```

Description

This is used with the function [MIO_M403_SetCaptureWindow\(\)](#) to define which data is returned when calling [MIO_M403_GetWindowData\(\)](#).

The DataMask of [MIO_M403_SetCaptureWindow\(\)](#) is created by oring the values of this enumerator.

MIO_M403_Config()

```
int MIO_M403_Config (int Slot);
```

Description

By calling the function [MIO_M403_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M403 is specified. This function must be called one time, before any attempted access to the module.

Initial state of the module:

- The event buffers are cleared
- The indices of the event buffers are empty
- The Angular Processing Unit is disabled and set as master
- The rotation speed is set to "0"
- No synchronization is configured
- The capturing is disabled

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

MIO_M403_SetEngineStartAngle()

```
void MIO_M403_SetEngineStartAngle (int Slot, double Angle);
```

Description

With this function a predefined start angle of the crankshaft can be defined. This probably is useful to don't start the engine at the upper dead center (UDC) at the first time the module is used after switching it on.

Parameters

- Angle
Engine start angle. The unit is rad.

MIO_M403_ConfigureAPU()

```
void MIO_M403_ConfigureAPU (int Slot, int Enable, int IsMaster);
```

Description

This function enables the calculation of the angular and defines, if the APU is independent from other M402/M403-APUs.

If several M402/M403 modules should be synchronized, don't use this function. Use [MIO_M402M403_Synchronize\(\)](#) instead.

Parameters

- Enable
Enables and disables the calculation of the angular.
- IsMaster
If the module is not synchronized with other modules set this parameter to "1".
If IsMaster is set to "0" the address calculation stops after one turn and waits for the synchronisation trigger of another module.

MIO_M403_SetRotationSpeed()

```
void MIO_M403_SetRotationSpeed (int Slot, double Frequency);
```

Description

This function specifies the rotation speed of the engine.

If several M402/M403 modules are synchronized, use [MIO_M402M403_SyncSetRotationSpeed\(\)](#) to set the engine rotation speed [MIO_M403_SetRotationSpeed\(\)](#).

Parameters

- Frequency
This is the rotation speed of the engine in rounds per second.

MIO_M403_SetThresholdVoltage()

```
void MIO_M403_SetThresholdVoltage(int Slot, int Channel,
                                    float V_PreOn, int Vin_gt_VPreOn,
                                    float V_On, int Vin_gt_VOn,
                                    float V_PreOff, int Vin_gt_VPreOff,
                                    float V_Off, int Vin_gt_VOff);
```

Description

The module is used to detect ignition events or injection events. For each event the start angle, end angle, start time and time for each event are stored.

Each event is defined by 4 conditions:

- a pre-start condition
- a start condition
- a pre-stop condition
- and a stop condition

An event begins, if first the pre-start condition was fulfilled and then start condition is met. The same is true for the end of an event. It ends, if first the pre-stop condition was fulfilled and then the stop condition is met.

If the pre-start/pre/stop condition was not fulfilled meeting the start/stop condition has no effect.

Each condition consists of a threshold voltage and a relation flag which specifies if the input voltage must be higher or lower than the specified threshold voltage to fulfill the condition. The threshold voltages are configurable in the range from -24.5V to + 24.5V with a resolution of 12.2mV.

Parameters

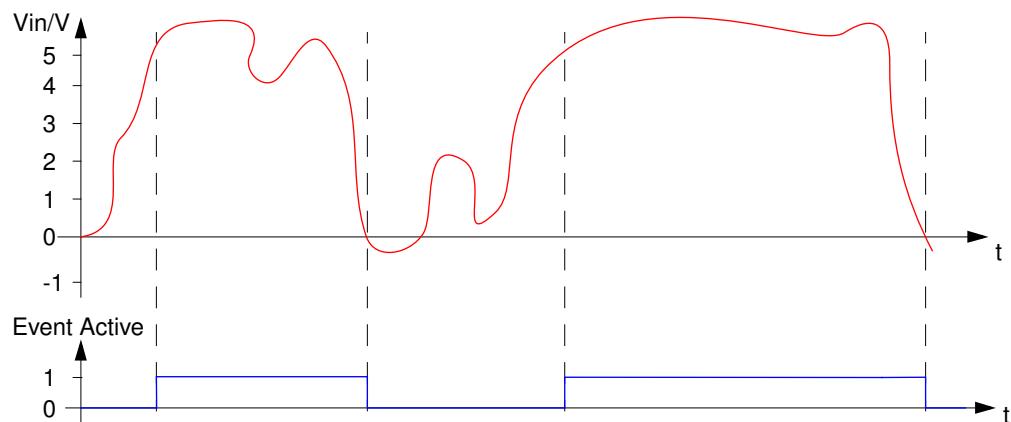
- **V_PreOn**
Threshold voltage of the condition that must be fulfilled before the start condition can get true.
- **Vin_gt_VPreOn, Vin_gt_VOn, Vin_gt_VPreOff, Vin_gt_VOff**
 - “1” if the Input voltage must be higher than the specified threshold voltage to fulfill the condition
 - “0” if the Input voltage must be lower than the specified threshold voltage to fulfill the (pre-)condition
- **V_On**
Threshold voltage for the start condition of the event.
- **V_PreOff**
Threshold voltage of the condition that must be fulfilled before the stop condition can get true.
- **V_Off**
Threshold voltage for the end conditionbegin of the event.

Example

1. The event begins if the input voltage was lower 1V and is now higher 5V and stops, if the input voltage was higher than 4V and is now lower than 0V.

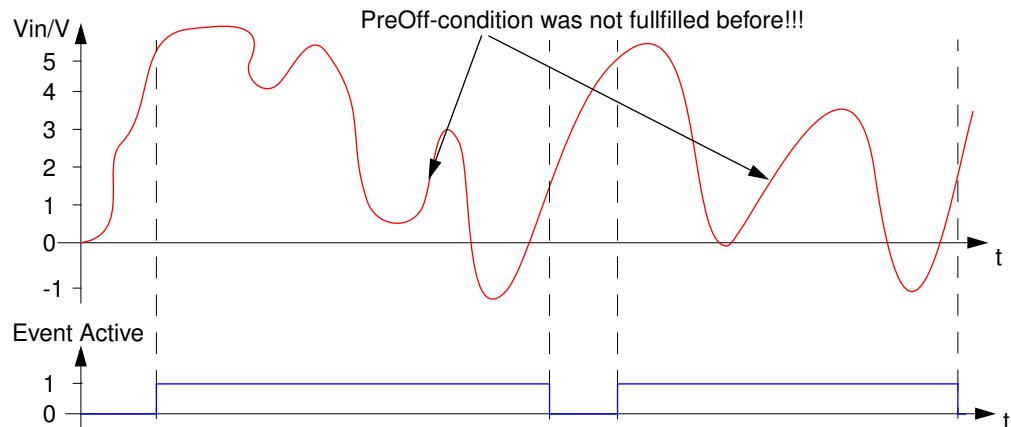
```
void MIO_M403_SetThresholdVoltage(Slot_M403, DesiredChannel,
                                   1.0, 0, /* lower than 1V */
                                   5.0, 1, /* Higher than 5V */
                                   4.0, 1, /* Higher than 4V */
```

```
0.0, 0, /* Lower than 0V */ );
```



2. The event begins if the input voltage was higher 4V and is now higher 5V and stops, if the input voltage was lower than -1 V and is now higher than 1.5V.

```
void MIO_M403_SetThresholdVoltage(Slot_M403, DesiredChannel,
        4.0, 1, /* Higher than 4V */
        5.0, 1, /* Higher than 5V */
        -1.0, 0, /* Lower than -1V */
        1.5, 1, /* Higher than 1.5V */ );
```



MIO_M403_SetReferenceAngle()

```
void MIO_M403_SetReferenceAngle (int Slot, int Channel, double Angle);
```

Description

Since the hardware revision 1.4 the M403 supports a reference angle for each channel. The reference angle is a phase shift of the specified angle against the base angle of the angular processing unit. Mostly this is the angle of the upper dead point (UDP) of the corresponding cylinder. The advantage of configuring the reference angle of each channel to its UDP is, that configuring the angles of capture windows is done in the same range for all channels and the start and end angles of an event then are also related to the UDP. If a reference angle is specified for a channel any information read from this channel by the functions

- [MIO_M403_GetnEvents\(\)](#)
- [MIO_M403_GetEvStartAngle\(\)](#)

- [MIO_M403_GetEvEndAngle\(\)](#)
- [MIO_M403_GetEvAngles\(\)](#)
- [MIO_M403_GetEvStartTime\(\)](#)
- [MIO_M403_GetEvEndTime\(\)](#)
- [MIO_M403_GetEvDuration\(\)](#) and
- [MIO_M403_GetWindowData\(\)](#)

is related to the reference angle.

Parameters

- Angle

This configures the phase shift of the specified channel against the base angle of the angular processing unit. The value range is between -4PI to +4PI.

MIO_M403_GetBaseAngle()

```
double MIO_M403_GetBaseAngle (int Slot);
```

Description

This function returns the current base angle of the angular processing unit. The angle of each M402 crank/cam channel of the module can differ from the angle returned by this function due to a phase shift defined by [MIO_M402_SetPhaseShift\(\)](#).

MIO_M403_GetVoltage()

```
int MIO_M403_GetVoltage(int Slot, int Channel, double* Voltage);
```

Description

This function returns the current voltage value and if the voltage is in the valid range.

Parameters

- Voltage is a pointer to the double variable in which the measured voltage is returned.

Return Value

“0” - Voltage is in the valid range

“1” - Over voltage / under voltage condition, depending on the voltage value.

MIO_M403_GetnEvents()

```
unsigned int MIO_M403_GetNoEvents(int Slot, int Channel);
```

Description

This function returns the number of events that occurred on this channel. If the return value must be decremented by one to use it as index of the newest event for the functions

- [MIO_M403_GetEvStartAngle\(\)](#)
- [MIO_M403_GetEvEndAngle\(\)](#)

- [MIO_M403_GetEvAngles\(\)](#)
- [MIO_M403_GetEvStartTime\(\)](#)
- [MIO_M403_GetEvEndTime\(\)](#)
- [MIO_M403_GetEvDuration\(\)](#)

The last 256 events are available.

Return Value

This function returns the number of events that occurred on this channel

Example

```
int i;
double StartAngle[8], EndAngle[8];
double StartTime[8], EndTime[8];
double Duration[8];
for (i=0;i<8;i++) {
    IO.Index[i] = MIO_M403_GetNoEvents(Slot_M403, i);
    IO.Index[i]--;
    MIO_M403_GetEvStartAngle(Slot_M403, i, IO.Index[i], &StartAngle[i]);
    MIO_M403_GetEvEndAngle(Slot_M403, i, IO.Index[i], &EndAngle[i]);
    StartTime[i] = MIO_M403_GetEvStartTime(Slot_M403, i, IO.Index[i]);
    EndTime[i] = MIO_M403_GetEvEndTime(Slot_M403, i, IO.Index[i]);
    /* oder */
    MIO_M403_GetEvAngles(Slot_M403, i, IO.Index[i], &StartAngle[i], &EndAngle[i]);
    Duration[i] = MIO_M403_GetEvDuration(Slot_M403, i, IO.Index[i]);
}
```

MIO_M403_GetEvStartAngle()

```
int MIO_M403_GetEvStartAngle(int Slot, int Channel, unsigned int Index, double *StartAngle);
```

Description

This function can be used to get the angle at which the event defined by the index started.

Parameters

- **Index**
This is the index of the event from which the start angle should be returned. The index of the last event is returned by [MIO_M403_GetnEvents\(\)](#). Because of the size of the circular buffer, the last 256 events can be indexed.
- **StartAngle**
Is a pointer to the double variable in which the start angle is returned.
The angle value's unit is rad.

Return Value

The function returns the content of the 15bit engine crankshaft cycle counter at the moment the event occurred. This can be used to detect if an event starts in one cycle and end in another one.

Example

See [MIO_M403_GetnEvents\(\)](#).

MIO_M403_GetEvEndAngle()

```
int MIO_M403_GetEvEndAngle(int Slot, int Channel, unsigned int Index, double *EndAngle)
```

Description

This function can be used to get the angle at which the event defined by the index ended.

Parameters

- **Index**
This is the index of the event from which the end angle should be returned. The index of the last event is returned by [MIO_M403_GetnEvents\(\)](#). Because of the size of the circular buffer, the last 256 events can be indexed.
- **EndAngle**
Is a pointer to the double variable in which the end angle is returned.
The angle value's unit is rad.

Return Value

The function returns the content of the 15bit engine crankshaft cycle counter at the moment the event occurred. This can be used to detect if an event starts in one cycle and end in another one.

Example

See [MIO_M403_GetnEvents\(\)](#).

MIO_M403_GetEvAngles()

```
int MIO_M403_GetEvAngles(int Slot, int Channel, unsigned int Index, double *StartAngle, double *EndAngle, int* nCycles);
```

Description

This function can be used to get both the angle at which the event defined by the index started and the angle at which the event defined by the index ended. If both values are needed, this function is much faster than calling [MIO_M403_GetEvStartAngle\(\)](#) and [MIO_M403_GetEvEndAngle\(\)](#) one after the other.

Parameters

- **Index**
This is the index of the event from which the angles should be returned. The index of the last event is returned by [MIO_M403_GetnEvents\(\)](#). Because of the size of the circular buffer, the last 256 events can be indexed.
- **StartAngle**
Is a pointer to the double variable in which the start angle is returned.
The angle value's unit is rad.
- **EndAngle**
Is a pointer to the double variable in which the end angle is returned.
The angle value's unit is rad.
- **nCycles**
Is a pointer to the number of crank shaft rotation cycles the event kept active. This can be used to detect if an event starts in one cycle and ends in another one.

Return Value

The function returns the content of the 15bit engine crankshaft cycle counter at the moment the event started.

Example

See [MIO_M403_GetnEvents\(\)](#).

MIO_M403_GetEvStartTime()

```
double MIO_M403_GetEvStartTime(int Slot, int Channel, unsigned int Index);
```

Description

This function can be used to get the time at which the event defined by the index started.

Parameters

- Index

This is the index of the event from which the start time should be returned. The index of the last event is returned by [MIO_M403_GetnEvents\(\)](#). Because of the size of the circular buffer, the last 256 events can be indexed.

Return Value

[MIO_M403_GetEvStartTime\(\)](#) returns the time at which the indexed event started in seconds. The time is normalized to the internal 32bit time base which has a resolution of 62.5ns.

Example

See [MIO_M403_GetnEvents\(\)](#).

MIO_M403_GetEvEndTime()

```
double MIO_M403_GetEvEndTime(int Slot, int Channel, unsigned int Index);
```

Description

This function can be used to get the time at which the event defined by the index ended.

Parameters

- Index

This is the index of the event from which the end time should be returned. The index of the last event is returned by [MIO_M403_GetnEvents\(\)](#). Because of the size of the circular buffer, the last 256 events can be indexed.

Return Value

[MIO_M403_GetEvEndTime\(\)](#) returns the time at which the indexed event ended in seconds. The time is normalized to the internal 32bit time base which has a resolution of 62.5ns.

Example

See [MIO_M403_GetnEvents\(\)](#).

MIO_M403_GetEvDuration()

```
double MIO_M403_GetEvDuration(int Slot, int Channel, unsigned int Index);
```

Description

This function returns the duration of the activity of the indexed event.

Parameters

- Index

This is the index of the event from which the duration should be returned. The index of the last event is returned by [MIO_M403_GetnEvents\(\)](#). Because of the size of the circular buffer, the last 256 events can be indexed.

Return Value

[MIO_M403_GetEvDuration\(\)](#) returns the duration of the indexed event in seconds.

Example

See [MIO_M403_GetnEvents\(\)](#).

MIO_M403_SetCaptureWindow()

```
void MIO_M403_SetCaptureWindow(int Slot, int Channel, int WindowNumber,
                                float StartAngle, float EndAngle,
                                int nMaxEvents, int DataMask);
```

Description

By default capturing occurs over the whole combustion cycle of the crankshaft (720deg). With this functions the capturing can be limited to defined angle ranges, the “capture windows”, e.g from 120deg to 180deg. Up to 4 windows are definable for each channel.

Parameters

- WindowNumber

This is the number of the window to be configured. Up to 4 (0..3) windows are definable for each channel.

- StartAngle

The start angle of the capture window. Unit: rad. Value range is -4PI..4PI.

- EndAngle

The end angle of the capture window. Unit: rad. Value range is -4PI..4PI.

The active angle (EndAngle - StartAngle) must be less than 4PI!

- nMaxEvents

This is the maximum number of events that can happen within one window. In [MIO_M403_GetWindowData\(\)](#) a maximum of nMaxEvents events is returned.

- DataMask

The DataMask configures which data is read from the module when calling [MIO_M403_GetWindowData\(\)](#). This is to avoid unneeded accesses to the hardware and thus not to waste I/O-time. The mask can be composed by oring the values from [eMIO_M403_WindowData](#).

If MIO_M403_Duration is selected MIO_M403_StartTime and MIO_M403_EndTime are selected automatically!



Example

See [MIO_M403_GetWindowState\(\)](#).

MIO_M403_GetWindowState()

```
int MIO_M403_GetWindowState(int Slot, int Channel, int WindowNo,  
                           tMIO_M403_WindowState *WindowState);
```

Description

This function returns if the capture window was left once again since the last call of this function. Additionally some information is returned about the events during the window was active the last time.

This function is only for debugging purposes.

Use [MIO_M403_GetWindowData\(\)](#) instead of doing everything on your own.

MIO_M403_GetWindowState() and MIO_M403_GetWindowData() cannot be used at the same time!



Parameters

- WindowNumber

This is the number of the window to be configured. Up to 4 (0..3) windows are definable for each channel.

- WindowState

This is a pointer to a variable of the type [tMIO_M403_WindowState](#). This contains the information about the events that happened when the window was active the last time.

Return Value

The function returns

- “-1” if the revision of the M403 does not supports capture windows and must be updated.
- “0” if the window was NOT left another time since the last call of [MIO_M403_GetWindowState\(\)](#).
- “1” if the window was left another time since the last call of [MIO_M403_GetWindowState\(\)](#).

MIO_M403_GetWindowData()

```
int MIO_M403_GetWindowData(int Slot, int Channel, int WindowNo,  
                           tMIO_M403_WindowData EventBuf[]);
```

Description

This function returns the desired data of the specified number of events that happened when the window was active the last time.

MIO_M403_GetWindowState() and MIO_M403_GetWindowData() cannot be used at the same time!



Parameters

- **WindowNumber**
This is the number of the window to be configured. Up to 4 (0..3) windows are definable for each channel.
- **EventBuf**
This variable is an array of type `tMIO_M403_WindowData` and after calling this function it contains the data of the events that happened when the window was active the last time.
`EventBuf` only contains data if
 - the window was active and left once again since the last call of `MIO_M403_GetWindowData()`
 - at least one event happened during the time the window was active.

Only the data configured by the `DataMask` in `MIO_M403_SetCaptureWindow()` contains data. This means if e.g. the `DataMask` is configured to return only `MIO_M403_StartAngle` and `MIO_M403_Duration` then `EventBuf[x].EndAngle` is not initialized. If more events happened in the window as configured by `MIO_M403_SetCaptureWindow()` only the configured number of events is returned.

Return Value

- “-1”: No new data available, window was not left once again since last call of `MIO_M403_GetWindowData()`.
In this case the content of `EventBuf` shall be ignored.
- “0” - No events happened when the window was active the last time.
In this case the content of `EventBuf` shall be ignored.
- Otherwise the return value contains the number of events that happened when the window was active the last time. In this case move the returned data to the model.

Example

```
/* IO.h */
#define M403_MAX_EVENTS     8
#define M403_NWINDOWS       4
#define M403_NCHANNELS      4
#define M_PI                  3.14
tMIO_M403_WindowData DataToModel[M403_NCHANNELS][M403_NWINDOWS][ENGINE_IN_MAXEVENTS];

/* Initialisation */
int IO_Init(void)
{
    int c, datamask;

    /* only start angle and duration needed! */
    datamask = MIO_M403_StartAngle | MIO_M403_Duration; /* from eMIO_M403_WindowData */
    for (c = 0; c < M403_NCHANNELS; c++) {
        /* Reference Angle for Cylinders */
        MIO_M403_SetReferenceAngle (Slot_M403, c, c * M_PI);
        MIO_M403_SetThresholdVoltage(Slot_M403, c, 3.0, 0, 5.0, 1, 5.2, 1, 3.0, 0);
        /* example angles for the windows */
        MIO_M403_SetCaptureWindow(Slot_M403, c, 0, 0, 2 * M_PI, 8, datamask);
        MIO_M403_SetCaptureWindow(Slot_M403, c, 1, 2 * M_PI, 4 * M_PI, 8, datamask);
        MIO_M403_SetCaptureWindow(Slot_M403, c, 2, 2.5 * M_PI, 3.5 * M_PI, 8, datamask);
        MIO_M403_SetCaptureWindow(Slot_M403, c, 3, -0.5 * M_PI, 0.5 * M_PI, 256, datamask);
    }
}

/* Runtime */
void IO_In  (void)
```

```
{  
    ** If new data available, copy it to the buffer that contains the data for the engine model  
    ** else ignore it!  
    */  
    tMIO_M403_WindowData tmpData[ENGINE_IN_MAXEVENTS];  
  
    int c, w;  
  
    for (c=0; c<M403_NCHANNELS; c++) {  
        for (w=0; w<M403_NWINDOWS; w++) {  
            if (t = MIO_M403_GetWindowData(Slot_M403, c, w, tmpData) < 0)  
                continue;  
            else if (t == 0)  
                memset (&DataToModel[c][w], 0, sizeof(tMIO_M403_WindowData) * M403_MAX_EVENTS)  
            else  
                memcpy (&DataToModel[c][w], &tmpData, sizeof(tMIO_M403_WindowData) *  
                        M403_MAX_EVENTS);  
        }  
    }  
}
```

10.4.17 M402 & M403: synchronized Use of Several M402 / M403

MIO_M402M403_Synchronize()

```
int MIO_M402M403_Synchronize(int SyncMasterSlot, int nSlaves, char *SyncSlaveSlots);
```

Description

`MIO_M402M403_Synchronize()` must be used to synchronize several M402/M403 with each other. This function synchronizes the modules by configuring the angular processing units of the modules in the correct way and routes the trigger lines in the whole system. Synchronization can be done between the M-Module slots of one carrier board as well as between any number of M-Modules on several carrier boards.



Don't use `MIO_M402_ConfigureAPU()` and `MIO_M403_ConfigureAPU()` together with this function because the synchronization gets lost in this case.

If several M402/M403 modules are synchronized, use `MIO_M402M403_SyncSetRotationSpeed()` to set the engine rotation speed instead of `MIO_M402_SetRotationSpeed()` and `MIO_M403_SetRotationSpeed()`.

Parameters

- SyncMasterSlot
There is one slot that must be configured as the sync master. The sync master can be any of the slots that are to be synchronized. The sync master must not always be configured as sync slave. It's APU is configured to synchronize and control the APUs of all the slaves.
- nSlaves
This is the number of slaves that is to be synchronized with the master slot. Up to 256 slaves are possible.
- SyncSlaveSlots
This is the address of a user defined array which contains the slot numbers of the modules that should be synchronized by the sync master. The slots can be on any carrier board that is plugged into the real time unit. Thereby the order of the slots is unimportant.

Return Value

This function returns the following values:

- 0 - Synchronization done correctly
- 1 - Sync master is neither an M402 nor an M403.
- 2 - Trigger routing not possible because all trigger lines are in use
- 3 - One of the sync slaves is neither an M402 nor an M403.

MIO_M402M403_SyncSetRotationSpeed()

```
void MIO_M402M403_SyncSetRotationSpeed (int SyncMasterSlot, double Frequency);
```

Description

`MIO_M402M403_SyncSetRotationSpeed()` **must** be used to set the engine rotation speed of all synchronized M402/M403 modules. This function must be called only one time for all synchronized modules on one carrier board if the rotation speed changes.



Don't use `MIO_M402M403_SyncSetRotationSpeed()` together with `MIO_M402_SetRotationSpeed()` and `MIO_M403_SetRotationSpeed()` because the synchronization gets lost in this case.

Parameters

- `SyncMasterSlot`

This must be the slot that is configured as synchronisation master by `MIO_M402M403_Synchronize()`.

- `Frequency`

This is the rotation speed of the engine in rounds per second.

10.4.18 M404: Waveform Generator, 8 Channels

Connector assignment

	Pin	Signal Description	Pin	Signal Description
1	1	Channel 0	14	AGND
2	2	Channel 1	15	AGND
3	3	Channel 2	16	AGND
4	4	Channel 3	17	AGND
5	5	Channel 4	18	AGND
6	6	Channel 5	19	AGND
7	7	Channel 6	20	AGND
8	8	Channel 7	21	AGND
9	9	n.c.	22	AGND
10	10	n.c.	23	AGND
11	11	n.c.	24	AGND
12	12	-15V	25	AGND
13	13	+15V		

Function Overview

Utility Functions

- [MIO_M404_GenerateSampleWaveform\(\)](#)

Initialization and Configuration

- [MIO_M404_Config\(\)](#)
- [MIO_M404_SetWaveform\(\)](#)
- [MIO_M404_EnableChannels\(\)](#)

Runtime Functions

- [MIO_M404_OutputConfig\(\)](#)
- [MIO_M404_SetFrequency\(\)](#)

MIO_M404_GenerateSampleWaveform()

```
void MIO_M404_GenerateSampleWaveform (tM402_Waveform WF, float Umax, float Umin,
                                      float PPRatio, int nTeeth, int MissingStart,
                                      int nMissing, float *Buf, int BufSize);
```

Description

see [MIO_M402_GenerateSampleWaveform\(\)](#)

MIO_M404_Config()

```
int MIO_M404_Config (int Slot);
```

Description

By calling the function [MIO_M404_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M404 is specified. This function must be called one time, before any attempted access to the module.

Initial state of the module:

- No waveforms are defined
- All gain values are 1.0 (0x1000)
- The first weighting register of each channel is set to 1.0 while any other weighting value is set to 0.0 (0x0000)
- The frequency is set to “0”
- All channels are disabled

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

MIO_M404_SetWaveform()

```
void MIO_M404_SetWaveform (int Slot, int Channel, float *Waveform0, float *Waveform1);
```

Description

This function loads the waveforms from the buffers into the M404.

Parameters

- Channel
0..7 are the waveforms for the channels 0..7
- Waveform0, Waveform1
For the 8 output channels up to 2 waveforms can be specified for each channel. If only one waveform is used, this should be Waveform0 and the parameter WaveForm1 should be *NULL*.

Example

```
float buf[2][65536];
int i;
MIO_M404_GenerateSampleWaveform(M402_Square, 7.5, 0.0, 0.5, 60, 58, 2, buf[0], 65536);
MIO_M404_GenerateSampleWaveform(M402_Sine, 7.5, 0.0, 0.5, 60, 58, 2, buf[1], 65536);
for (i=0;i<8;i++) {
```

```

    MIO_M404_SetWaveform (Slot_M404, i, buf[0], buf[1]);
}

```

MIO_M404_OutputConfig()

```
void MIO_M404_OutputConfig (int Slot, int channel, float Weighting0, float Weighting1,
                           float Gain);
```

Description

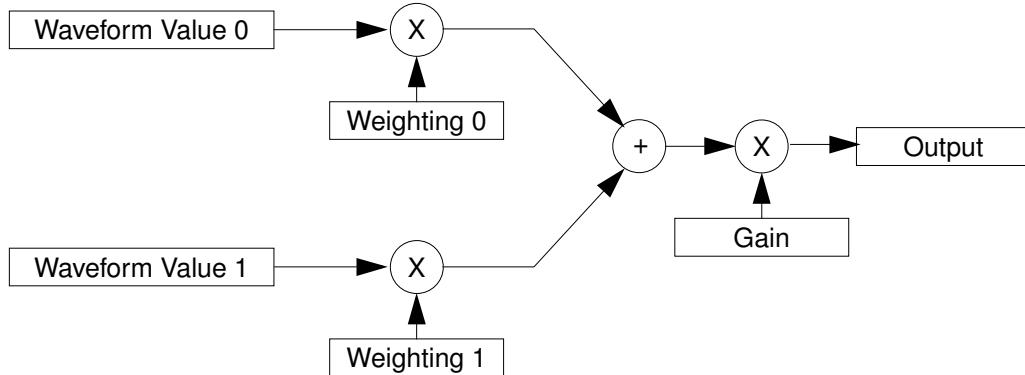
With this function the amplification and the intermixing of the waveforms of each channel is configured. After power on the weighting of channel 0 is 1.0 while the weighting of channel 1 is 0.0 and the gain is set to 1.0.

Parameters

- Weighting0, Weighting1, Gain

For each of the 2 waveforms it is configurable how strong they are considered for the output voltage. Therefore each waveform is multiplied with the corresponding weighting value.

The value range of the weighting is between 0.0 and 1.0 where 0.0 means the current waveform is not considered for the calculation of the current output value and 1.0 means that the value is considered with 100%. Then all weighted waveform values are added together and then are multiplied by the gain. The value range of the gain is 0 to 3.9.



MIO_M404_EnableChannels()

```
void MIO_M404_EnableChannels (int Slot, int EnableMask);
```

Description

This function enables the output DACs of the channels. Disabled channels keep the last output value. After switching on all channels are disabled.

Parameters

- EnableMask

This parameter is a bitmask for enabling the output channels.

- bit0 enables channel 0
- bit1 enables channel 1

- bit2 enables channel 2
- bit3 enables channel 3
- bit4 enables channel 4
- bit5 enables channel 5
- bit6 enables channel 6
- bit7 enables channel 7

MIO_M404_SetFrequency()

```
void MIO_M404_SetFrequency (int Slot, int Channel, double Frequency);
```

Description

This function specifies how often per second the whole waveform is replayed.

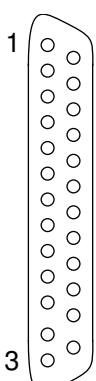
Parameters

- Frequency
This is the number of iterations per second.

10.4.19 M405: LIN Interface, 12 Channels

Connector assignment

Female SubD-25 Connector



Pin	Signal Description	Pin	Signal Description
1	LIN Channel 0	14	LIN Channel 1
2	LIN Channel 2	15	LIN Channel 3
3	Supply (+12V) for Channel 0..3	16	GND for Channel 0..3
4	Supply (+12V) for Channel 0..3	17	GND for Channel 0..3
5	LIN Channel 4	18	LIN Channel 5
6	LIN Channel 6	19	LIN Channel 7
7	Supply (+12V) for Channel 4..7	20	GND for Channel 4..7
8	Supply (+12V) for Channel 4..7	21	GND for Channel 4..7
9	LIN Channel 8	22	LIN Channel 9
10	LIN Channel 10	23	LIN Channel 11
11	Supply (+12V) for Channel 8..11	24	GND for Channel 8..11
12	Supply (+12V) for Channel 8..11	25	GND for Channel 8..11
13			

Function Overview

Initialization and Configuration

- [MIO_M405_Config\(\)](#)
- [MIO_M405_ConfigureChannel\(\)](#)
- [MIO_M405_SetMsgConfig\(\)](#)

Runtime Functions

- [MIO_M405_TriggerMessage\(\)](#)
- [MIO_M405_SetTXData\(\)](#)
- [MIO_M405_GetMessage\(\)](#)

Query the State of the Module

- [MIO_M405_GetDataAvailableMask\(\)](#)
- [MIO_M405_GetBufferFullMask\(\)](#)
- [MIO_M405_GetErrorMask\(\)](#)
- [MIO_M405_GetError\(\)](#)
- [MIO_M405_GetShortedGNDMask\(\)](#)
- [MIO_M405_GetBusIdleMask\(\)](#)
- [MIO_M405_GetLinLineState\(\)](#)

Change the State of the Module

- [MIO_M405_RstError\(\)](#)
- [MIO_M405_RstFIFO\(\)](#)
- [MIO_M405_DoWakeup\(\)](#)

- [MIO_M405_DoWakeup\(\)](#)

struct tlinmsg

```
typedef struct tlinmsg {  
    unsigned char ProtectedID;  
    unsigned char Length;  
    unsigned char Data[256];  
    unsigned char Checksum;  
    unsigned char Status;  
} tlinmsg;
```

Description

A variable of type tlinmsg contains the content of the oldest message of the reception FIFO after MIO_M405_GetMessage is called.

MIO_M405_Config()

```
int MIO_M405_Config (int Slot);
```

Description

By calling the function [MIO_M405_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M405 is specified. This function must be called one time, before any attempted access to the module.

Initial state of the module:

- Message FIFOs are empty
- All Channels are disabled
- All LIN transceivers are disabled
- All message IDs are disabled

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

The module is set back to its initial state.

MIO_M405_ConfigureChannel()

```
void MIO_M405_ConfigureChannel (int Slot, int Channel, float Baudrate, unsigned short  
                                BaudrateDetection, int IsMaster,int Enable, int nSyncBreak, int nSyncDel,  
                                int DisableBusIdleDetection, intGenParErr, int TraceMode);
```

Description

With this function the basic configuration of a channel is done.

Parameters

- Baudrate

The baudrate can be set in a range from 2400 Baud to 30 kBaud. The baudrate is set for the master and the slaves of each channel.

- BaudrateDetection

BaudRateDetection switches the baud rate detection of the channel on for each slave of a channel as specified in the LIN specification ≥ 2.0 . If BaudRateDetection is switched on, the very first message is neglected.

- IsMaster

If IsMaster is set to “1” the master node of the channel is enabled (as well as the slave node). That means that the channel is able to send frame headers. Switching this flag on does not affect the slave functionality of the channel.

- Enable

Enables “1” or disables “0” the transceiver of the channel.

- nSyncBreak

This is the number of bit times the master pulls the LIN line low as sync break.

To conform to the LIN specification this value should be set to 13. Others values are only for fail-safe testing issues. Valid range: 5..63 (bit times).

- nSyncDel

This is the number of bit times the master releases the LIN line (“high”) after the sync break and before the sync byte is send.

To conform to the LIN specification this value should be set to 1. Others values are only for fail-safe testing issues. Valid range: 1..15 (bit times).

- DisableBusIdleDetection

Set this to “1” if the disable the bus idle detection should be disabled. By default after 4 seconds without any activity on the bus the module switches to the bus idle mode. In this case a bus wake-up is needed to transmit an data again.

If the bus idle detection is disabled no wake-up is needed even if the delay between 2 messages is more the 4 seconds. This is useful for testing purposes where transmission of messages is triggered manual.

Disabling the bus idle detection is possible since hardware revision 1.2.

- GenParErr

If the flag is set to “1” an erroneous parity is generated by the master. Set this flag to “1” for fail-safe testing issues.

- TraceMode

By default only message are stored in the Rx-FIFO, if they are received completely and without any error.

Enabling the trace mode causes the receiver to store any received data byte in the Rx-FIFO even if a timeout happens or if the checksum is wrong. The only information that must be received correctly is the message ID.

Therefore this mode can be used to trace a LIN bus and to analyse it for erroneous data.

MIO_M405_SetMsgConfig()

```
void MIO_M405_SetMsgConfig(int Slot, int Channel, int ID, int Length, int IsRead, int IsWrite,  
                           int IsEventTriggered, int UseEnhancedChecksum);
```

Description

This function is used to configure what the channels slave has to do with received IDs. It must be called once for each ID to be used.

Parameters

- ID

This is the to be configured ID of the specified channel. Valid range: 0..61

- Length
Number of data bytes in the message defined by “ID”. Valid range: 0..8
- IsRead
Should the data of the message with the specified ID be received? If yes the received data can be read by calling [MIO_M405_GetMessage\(\)](#).
Note: It is also possible to receive AND transmit the message data for the specified ID on the same channel!
- IsWrite
Should the data of the message with the specified ID be transmitted? If yes the to be transmitted data must be set by calling [MIO_M405_SetTXData\(\)](#).
Note: It is also possible to receive AND transmit the message data for the specified ID on the same channel!
- IsEventTriggered
If IsEventTriggered is set to “1” the transmitting slave does only answer to an ID if its to be transmitted data has changed. Setting this parameter to “1” does only take affect if IsWrite for the message is set to “1”.
- UseEnhancedChecksum
With the LIN specification 2.0 the enhanced checksum was introduced. Now slaves can use 2 possibilities of checksums to be used. The classic checksum is calculated over all the data bytes while the enhanced checksum is calculated over all the data bytes and the protected ID. By setting UseEnhancedChecksum to “1” the enhanced checksum is generated / verified.

Example

If messages with the LIN ID 0x23 have 4 data bytes, are to be received and they use the classic checksum use the following function call:

```
MIO_M405_SetMsgConfig(M405_Slot, 0, 0x23, 4, 1, 0, 0, 0);
```

If messages with the LIN ID 0x30 have 7 data bytes, are to be transmitted and they use the enhanced checksum use the following function call:

```
MIO_M405_SetMsgConfig(M405_Slot, 0, 0x30, 7, 0, 1, 0, 1);
```

Don't forget to specify the to be transmitted data with [MIO_M405_SetTXData\(\)](#)

MIO_M405_TriggerMessage()

```
void MIO_M405_TriggerMessage (int Slot, int Channel, char ID);
```

Description

The scheduler for the LIN masters must be specified in software. To trigger the transmission of the master frame with the desired ID call [MIO_M405_TriggerMessage\(\)](#).

Note: If the master stops transmitting master frames the slaves go to the bus idle state. Also see [MIO_M405_GetBusIdleMask\(\)](#) and [MIO_M405_DoWakeups\(\)](#).

MIO_M405_GetMessage()

```
int MIO_M405_GetMessage (int Slot, int Channel, tlinmsg* linmsg);
```

Description

This function reads the received data from the receiver FIFO of the channel (if any). Data is only put to the receiver FIFO if the associated ID is configured as “IsRead” message.

The data is returned in the struct from type tlinmsg. The structure of tlinmsg can be seen here [struct tlinmsg](#).

If no data is available in the FIFO the function returns -1 otherwise the number of data bytes of the message is returned.

MIO_M405_SetTXData()

```
void MIO_M405_SetTXData(int Slot, int Channel, int ID, unsigned char* Data, int Checksum);
```

Description

This function specifies the data bytes for messages with the specified ID. This only must be done if the ID is configured as transmitting ID.

Parameters

- ID
This is the ID of the message which is to be configured.
- Data
This is a buffer of up to 8 data bytes. The byte on the lowest address is sent first of all. The size of the data buffer must be at least the number of data bytes in the message (see [MIO_M405_SetMsgConfig\(\)](#))
- Checksum
If the checksum is set to -1 the checksum is calculated by [MIO_M405_SetTXData\(\)](#) (Recommended). Otherwise the lowest 8 Bits of Checksum are sent as the checksum what is only recommended for fail-safe testing issues.

Example

```
#define MSGLEN 8
int MySetTxData(...)

{
    unsigned char buf[MSGLEN];
    MIO_M405_SetMsgConfig(M405_Slot, 0 /*Channel*/, 0x35, MSGLEN, 0, 1, 0, 1);
    data[0] = 0x67;
    data[1] = 0xfd;
    .
    .
    MIO_M405_SetTXData(M405_Slot, 0 /*Channel*/, 0x35 /*ID*/, data, -1);
}
```

MIO_M405_GetBufferFullMask()

```
int MIO_M405_GetBufferFullMask(int Slot);
```

Description

This function returns a bit mask for each receiver fifo which is full. It is recommended to call [MIO_M405_RstFIFO\(\)](#) after a buffer is full to ensure that the data is consistent.

Bit 0 corresponds to channel 0
Bit 1 corresponds to channel 1
Bit 2 corresponds to channel 2
and so on.

MIO_M405_GetDataAvailableMask()

```
int MIO_M405_GetDataAvailableMask(int Slot);
```

Description

This function returns a bit mask for each receiver fifo which contains at least one complete message.

Bit 0 corresponds to channel 0
Bit 1 corresponds to channel 1
Bit 2 corresponds to channel 2
and so on.

MIO_M405_GetErrorMask()

```
int MIO_M405_GetErrorMask(int Slot);
```

Description

This function returns a bit mask where the corresponding bit for each channel is set to “1” if the last received message was erroneous.

Bit 0 corresponds to channel 0
Bit 1 corresponds to channel 1
Bit 2 corresponds to channel 2
and so on.

For further information also see [MIO_M405_GetError\(\)](#).

To reset the error counter call [MIO_M405_RstError\(\)](#).

MIO_M405_GetError()

```
int MIO_M405_GetError(int Slot, int Channel, int *ErrorID, int *ErrorCounter, int *MsgID)
```

Description

This function returns if errors happened on the bus, if an error is active and which kind of error happened at last with which message ID.

Parameters

Set the ErrorID, ErrorCounter or MsgID to NULL if the value is not needed.

- ErrorID
 - If ErrorID is not NULL the ID of the error is returned that happened last.
 - 0 - No error happened
 - 1 - Received wrong Checksum
 - 2 - Parity error in identifier field
 - 3 - Timeout

- 4 - Error in synchronization
 - 5 - Collision detected. Collision detection is only possible by a transmitting slave.
 - 7 - Short to GND
- ErrorCounter
The error counter is incremented by “1” each time an error happens. If the error counter reaches “0xffff” it keeps its value.
To reset the error counter call [MIO_M405_RstError\(\)](#).
 - MsgID
If possible this contains the message ID of the erroneous message.

Return Value

The function returns “1” if the last message on the bus was erroneous. As soon as a correct message is received “0” is returned.

MIO_M405_GetShortedGNDMask()

```
int MIO_M405_GetShortedGNDMask(int Slot);
```

Description

This function returns a bit mask for each channel whose LIN line is set low for more than 5 seconds. A shorted channel is disabled and can only be re-enabled by calling [MIO_M405_DoWakeup\(\)](#).

MIO_M405_GetBusIdleMask()

```
int MIO_M405_GetBusIdleMask(int Slot);
```

Description

This function returns a bit mask for each channel that is in bus idle state. As specified in the LIN specification >= 2.0 the idle state is reached if the bus was high for more than 4 seconds without any frames.

Bit 0 corresponds to channel 0
Bit 1 corresponds to channel 1
Bit 2 corresponds to channel 2
and so on.

Note: A sleeping channel neither transmit or receive any data nor send the master part of a frame until it is woken up by calling [MIO_M405_DoWakeup\(\)](#).

MIO_M405_GetLinLineState()

```
int MIO_M405_GetLinLineState(int Slot);
```

Description

This function returns a bit mask for each LIN channel which represents the current state of the LIN line. A “1” means that the LIN line is in recessive (high) state, a “0” that it is pulled low.

Bit 0 corresponds to channel 0
Bit 1 corresponds to channel 1
Bit 2 corresponds to channel 2
and so on.

MIO_M405_RstError()

```
void MIO_M405_RstError(int Slot, int Mask);
```

Description

This function resets the error counter - and thus the error flag - of the channels specified by mask.

Parameters

- Mask

Bit 0 corresponds to channel 0
Bit 1 corresponds to channel 1
Bit 2 corresponds to channel 2
and so on.

To reset all set error flags the return value of [MIO_M405_GetErrorMask\(\)](#) can be used as mask.

MIO_M405_RstFIFO()

```
void MIO_M405_RstFIFO(int Slot, int Mask);
```

Description

This function resets the fifo of the channels specified by mask. This should be done if the fifo full flag is set for a channel.

Parameters

- Mask

Bit 0 corresponds to channel 0
Bit 1 corresponds to channel 1
Bit 2 corresponds to channel 2
and so on.

To reset the fifos of all channels whose fifos are full the return value of [MIO_M405_GetBufferFullMask\(\)](#) can be used as mask.

MIO_M405_DoWakeups()

```
void MIO_M405_DoWakeups(int Slot, int Mask);
```

Description

Performs a wake-up on the LIN busses of the specified channels which means it pulls the bus low for about 4ms.

Note: A sleeping channel neither transmit or receive any data nor send the master part of a frame until it is woken up. After the wake-up is executed the master should wait for about 100ms before it starts sending requests.

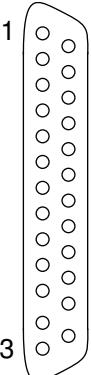
Parameters

- Mask
 - Bit 0 corresponds to channel 0
 - Bit 1 corresponds to channel 1
 - Bit 2 corresponds to channel 2
 - and so on.

To wake-up all channels which are idle (“sleeping”) use the return value of [MIO_M405_GetBusIdleMask\(\)](#) as mask.

10.4.20 M406: PSI5 Interface, 8 Slaves + 1 Master

Connector assignment



	Pin	Signal Description	Pin	Signal Description
1	1	PSI5 Channel 0 (+)	14	PSI5 Channel 0 (-)
13	2	PSI5 Channel 0 - Daisy Chain	15	PSI5 Channel 1 (-)
	3	PSI5 Channel 1 (+)	16	PSI5 Channel 1 - Daisy Chain
	4	PSI5 Channel 2 (+)	17	PSI5 Channel 2 (-)
	5	PSI5 Channel 2 - Daisy Chain	18	PSI5 Channel 3 (-)
	6	PSI5 Channel 3 (+)	19	PSI5 Channel 3 - Daisy Chain
	7	PSI5 Channel 4 (+)	20	PSI5 Channel 4 (-)
	8	PSI5 Channel 4 - Daisy Chain	21	PSI5 Channel 5 (-)
	9	PSI5 Channel 5 (+)	22	PSI5 Channel 5 - Daisy Chain
	10	PSI5 Channel 6 (+)	23	PSI5 Channel 6 (-)
	11	PSI5 Channel 6 - Daisy Chain	24	PSI5 Channel 7 (-)
	12	PSI5 Channel 7 (+)	25	GND for Master (=PSI5 Channel 7 (-))
	13	Ubat for Master (Channel 7)		

Function Overview

If the A24D32 address space is available on the used carrier board it is used. Otherwise the A8D16 address space is used.

Initialization and Configuration

- [MIO_M406_Config\(\)](#)
- [MIO_M406_SlaveConfigureSensor\(\)](#)
- [MIO_M406_SlaveConfigureChannel\(\)](#)
- [MIO_M406_MasterConfigure\(\)](#)
- [MIO_M406_SlaveSetInitData\(\)](#)
- [MIO_M406_SlaveEnableSensor\(\)](#)
- [MIO_M406_MasterEnableTS\(\)](#)
- [MIO_M406_MasterConfigure\(\)](#)

Runtime Functions

- [MIO_M406_SlaveSetTxData\(\)](#)
- [MIO_M406_MasterGetData\(\)](#)
- [MIO_M406_MasterSendData\(\)](#)
- [MIO_M406_MasterGetError\(\)](#)
- [MIO_M406_MasterGetSensorState\(\)](#)

Examples

- [Loopback, PSI5-P10CRC-500/4H, Master channel, channels 0..3 as Slave](#)

enum eM406_SensorState

```
typedef enum {
    M406_SensorReady =      0x1e7,
    M406_SensorDefect =     0x1f4,
    M406_SensorBusy =       0x1e8,
    M406_SensorDiagnostic = 0x1e9,
    M406_SensorUnlocked =   0x1e6
} eM406_SensorState;
```

Description

No longer used since CarMaker 4.0.6.

enum eM406_SensorManufacturer

```
enum {
    M406_Unknown =      0x00,
    M406_Autoliv =      0x40,
    M406_Bosch =        0x10,
    M406_Continental =  0x80,
    M406_SiemensVDO =   0x20,
    M406_TRW =          0x50,
    M406_Freescale =   0x46
} eM406_SensorManufacturer;
```

Description

This is a number of IDs of PSI5 sensor manufacturers.

struct tM406_SensorIDData

```
typedef struct {
    unsigned int ProtVersion;
    unsigned int nInitNibbles;
    unsigned int ManufacturerCode;
    unsigned int SensorType;
    unsigned int SensorParameter;
    unsigned int SensorManCode;
    unsigned int VehicleManCode;
    unsigned int SensorProdYear;
    unsigned int SensorProdMonth;
    unsigned int SensorProdDay;
    unsigned int SensorTraceInfo[14];
} tM406_SensorIDData;
```

Description

This struct describes the ID data which is sent during initialization phase “2”. For further informations see the PSI5 specification.

- **nInitNibbles**

This value has a double function. On the one hand this value is part of the ID data. on the other hand this value specifies how many block identifiers are sent.
please realize that this value is “nibbles” but not characters or integers!

According to the PSI5 specification the *ProtVersion*, *nInitNibbles*, *ManufacturerCode*, *SensorType* and *SensorParameter* - which in effect are 9 nibbles - is mandatory to send. Thus if the value is lower than 9 the M406 sends 9 nibbles anyway.

MIO_M406_Config()

```
int MIO_M406_Config (int Slot);
```

Description

By calling the function [MIO_M406_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M406 is specified. This function must be called one time, before any attempted access to the module.

Initial state of the module:

- Message FIFOs are empty
- All Channels are disabled

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

MIO_M406_SlaveConfigureChannel()

```
int MIO_M406_SlaveConfigureChannel (int Slot, int Channel, char* CfgString,  
double *TimeSlotStart );
```

Description

[MIO_M406_SlaveConfigureChannel\(\)](#) configures the PSI5 mode. This function analyzes the PSI5 string and configures the sensors based on this string.

Parameters

- CfgString
CfgString is the text for the PSI5 mode you use. The text can be found in the PSI5 specification.
- TimeSlotStart
This is an array of doubles. The number of elements of the array must be the same as the number of sensors on the channel. Each value stands for the time between the beginning of the sync pulse and the beginning of the time slot of the corresponding channel.
TimeSlotStart[0] corresponds to sensor 0.
TimeSlotStart[1] corresponds to sensor 1.
TimeSlotStart[2] corresponds to sensor 2.
TimeSlotStart[3] corresponds to sensor 3.
The unit is seconds. The valid value range is 100ns .. 1.3ms.

MIO_M406_SlaveConfigureSensor()

```
void MIO_M406_SlaveConfigureSensor (int Slot, int Channel, int nSensors, double Baudrate,  
double TimeSlotOffset, double CycleTime, int nBits, int IsDaisyChainMode,  
int IsSynchronous, int IsTimeMultiplexed, int UseCRC );
```

Description

`MIO_M406_SlaveConfigureSensor()` configures one sensor. If you don't need any special settings which are not supported by the PSI5 string, prefer `MIO_M406_SlaveConfigureChannel()` instead of `MIO_M406_SlaveConfigureSensor()`.

The configuration of each sensor can differ in each detail from each other sensor.

Parameters

- nSensors
The number of the sensor of the channel which is to be configured.
- Baudrate
The baudrate must be in the range from 800Baud to 1MBaud.
- TimeSlotOffset
This is the time offset from the beginning of the first time slot to the beginning of the time slot of the specified sensor. The unit is seconds. The valid value range is 100ns .. 1.3ms.
- CycleTime
In synchronous mode this is the expected cycle time. That means it is the expected time from one sync-pulse to the subsequent one. In asynchronous mode the CycleTime is the time from one data frame to the subsequent one.
The unit is seconds. The valid value range is 100ns .. 1.3ms.
- nBits
The number of data bits, messages from this sensor contain.
- IsDaisyChainMode
If “1” the daisy chain mode is used, if not the daisy chain relay is always open.
- IsSynchroous
Is there a sync pulse or the
- IsTimeMultiplexed
If “1” the message data 0 and message data 1 are sent in alternating order. If “0” only message data 0 is sent. Message data 0 and message data 1 are specified by calling `MIO_M406_SlaveSetTxData()`.
- UseCRC
If “1” the a 3-bit-CRC is used for data integrity if “0” only a parity bit is sent.

MIO_M406_SlaveEnableSensor()

```
void MIO_M406_SlaveEnableSensor (int Slot, int Channel, int EnableMask);
```

Description

This function enables the slave channels.

Parameters

- EnableMask
This parameter is a bitmask for enabling the slave channels.
bit0 enables sensor 0
bit1 enables sensor 1
bit2 enables sensor 2
bit3 enables sensor 3
bit4 enables sensor 4
bit5 enables sensor 5
bit6 enables sensor 6
bit7 enables sensor 7

MIO_M406_SlaveSetInitData()

```
void MIO_M406_SlaveSetInitData (int Slot, int Channel, int SensorNo,
                                tM406_SensorIDData *Data_PH2, double dPH1, unsigned char nPH2,
                                unsigned int Data_PH3[], unsigned char nPH3, unsigned char nValuesPH3);
```

Description

This configures the initialization phase of the sensor.

The initialization configuration of each sensor can differ in each detail from each other sensor.

Parameters

- **SensorNo**
The number of the sensor of the channel which is to be configured.
- **Data_PH2**
This is the configuration data in initialization phase 2.
See [struct tM406_SensorIDData](#) and the PSI5 specification for further details.
- **dPH1**
Duration of initialization phase “1” in seconds. In initialization phase “1” the sensor does nothing but self initialization. The initialization phase “1” is emulated by the M406.
According to the PSI5 specification this value should be in the range from 50.. 150ms, typical 100ms. The possible value range is 0.7ms..160ms.
- **nPH2**
Number of repetitions during initialization phase “2”. During initialization phase “2” the sensor sends its ID data. This parameters specifies how often each block ID and its corresponding block data is sent before the next block id is sent. The number of ID blocks itself is specified within the ID data.
See the PSI5 specification for further information.
According to the PSI5 specification this value should be at least 2messages, typical 10messages. The possible value range is 0..255.
- **Data_PH3**
Data_PH3 is the data that is sent in init phase 3.
During phase “3” the sensor transmits “Sensor Ready”, “Sensor Defect” or other status data. See the PSI5 specification for further information.
In most cases only one byte is sent which contains information if the sensor is ok or not, see [enum eM406_SensorState](#). But there are also slaves that send slave specific data in initialization phase 2.
At least the number of values specified in `nValuesPH3` must be configured here. A maximum of 128 Bytes is possible here.
- **nPH3**
This parameter specifies how often all the data in `Data_PH3` is repeated.
According to the PSI5 specification this value should be at least 2messages, typical 10messages. The possible value range is 0..255.
If `nPH3` is set to 0 the init phase 3 is not left which means the `Data_PH3` is repeated again and again. This for example is used if a sensor is not ready / defective.
- **nValuesPH3**
This is the number of values that are sent in init phase 3. The value must be in the range from 0..127.

MIO_M406_SlaveSetTxData()

```
void MIO_M406_SlaveSetTxData(int Slot, int Channel, unsigned int SensorNo, unsigned int Data0,  
                             unsigned int Data1);
```

Description

When the sensor is in run mode the data to be transmitted is specified here. Data 1 only takes effect if TimeMultiplexed mode is specified by [MIO_M406_SlaveConfigureSensor\(\)](#).

Parameters

- SensorNo
The number of the sensor of the channel which is to be configured.
- Data0, Data1
If TimeMultiplexed mode is specified by [MIO_M406_SlaveConfigureSensor\(\)](#) Data0 and Data1 are sent in an alternating order. If not, only data 0 is sent.

MIO_M406_MasterConfigure()

```
int MIO_M406_MasterConfigure (int Slot, char* CfgString, double *TimeSlotStart,  
                             double SyncDuration , int PWM2Slave );
```

Description

[MIO_M406_MasterConfigure\(\)](#) configures the PSI5 mode. This function analyzes the PSI5 string and configures the sensors based on this string.

Parameters

- CfgString
CfgString is the text for the PSI5 mode you use. The text can be found in the PSI5 specification.
- TimeSlotStart
This is an array of doubles. The number of elements of the array must be the same as the number of sensors on the channel. Each value stands for the time between the beginning of the sync pulse and the beginning of the time slot of the corresponding channel.
TimeSlotStart[0] corresponds to sensor 0.
TimeSlotStart[1] corresponds to sensor 1.
TimeSlotStart[2] corresponds to sensor 2.
TimeSlotStart[3] corresponds to sensor 3.
The unit is seconds. The valid value range is 100ns..1.3ms.
- SyncDuration
This is the length of the sync pulse. According to the PSI5 specification the sync pulse should sustain about 16us. However the sync pulse can be configured in the range 100ns .. 1.3ms. The unit is seconds.
- PWM2Slave
This is part of the preliminary PSI5 specification 2.0. Up to 1.3 the communication from the master to the slave is done via missing and existing sync pulses. Since specification 2.0 there are two modes for M2S communication: Missing teeth mode (classic mode) and the PWM mode where the bit value is displayed by the length of the syncpulse: normal length = "0", enhanced length "1". By setting this value to "0" the classic M2S communication is selected. When setting the value to "1" the Pulse length coded M2S communication is selected.

MIO_M406_MasterEnableTS()

```
void MIO_M406_MasterEnableTS (int Slot, int EnableMask);
```

Description

This function enables the reception of slave data in the corresponding time slot.

Parameters

- EnableMask
 - If bit 0 is set, data reception in time slot 0 is enabled.
 - If bit 1 is set, data reception in time slot 1 is enabled.
 - and so on ...
A maximum of 4 time slots is valid, what means the value must be in the range of 0x00 to 0x0f.

MIO_M406_MasterSendData()

```
void MIO_M406_MasterSendData(int Slot, unsigned int Data, unsigned int SensorAddr,  
                             unsigned int FunctionCode, unsigned int ReadAddress);
```

Description

Not yet implemented.

MIO_M406_MasterGetError()

```
unsigned int MIO_M406_MasterGetError(int Slot);
```

Description

Returns an error mask where the bits of the return value correspond to the timeslots. If a bit is set to "1" the master received erroneous data in the corresponding timeslot.

MIO_M406_MasterGetData()

```
int MIO_M406_MasterGetData(int Slot, int SensorNo, unsigned int *RxData);
```

Description

This function returns the data which the master received from the sensor. The master contains a FIFO for each connected slave. Up to 4 slaves are possible. Depending on the configured time slot for each slave and the time the data is received after the sync pulse the data is put into the corresponding FIFO.

Parameters

- SensorNo
SensorNo specifies the FIFO from which the data is read from. The FIFO number corresponds to the sensor number.
- RxData
Here the received data is returned.

Return Value

If there is any data in the FIFO the function returns “1” and the data is returned in RxData. If the FIFO is empty, the function returns “0”.

MIO_M406_MasterGetSensorState()

```
int MIO_M406_MasterGetSensor(int Slot);
```

Description

This function returns the sensor states of all sensors of a channel that the master received from its sensors during initialization phase 3. The sensor state is normalized to 8 bits according to the PSI5 specification. Also see [enum eM406_SensorState](#).

- Bit 7..0 correspond to Sensor 0
- Bit 15..8 correspond to Sensor 1
- Bit 23..16 correspond to Sensor 2
- Bit 31..24 correspond to Sensor 3

Examples

Loopback, PSI5-P10CRC-500/4H, Master channel, channels 0..3 as Slave

```
#define Slot_PSI5 1
#define nTIMESLOTS 4

struct {
    unsigned int SlaveTxData[nTIMESLOTS];
    unsigned int MasterRxData[nTIMESLOTS];
    unsigned int MasterIsError[nTIMESLOTS];
    unsigned int MasterDataAvailable[nTIMESLOTS];
    unsigned int MasterSensorState[nTIMESLOTS];
} tIO;
tIO IO;

int IO_Init (void) /* RTMaker: IO_Start */
{
    int s /* sensor, here also channel */;
    tM406_SensorIDData IDData; /* see struct tM406_SensorIDData and PSI5-Spec S. 44 for further
                                 details. */
    double ts[4] = {0.000044, 0.000270, 0.000268, 0.000380}; /* example timeslot start values */
    /* cfgstring=PSI5-Mode: Synchronous,parallel,10bits,CRC,189kbit/s,4 Time Slots (=4
     Slaves), Cycletime=500us */
    char cfgstring[] = "PSI5-P10CRC-500/4H";

    IDData.ProtVersion = 0x4;           // fixed value, protocoll version
    IDData.nInitNibbles = 0x20;        // 9 Nibbles = 0000 1001, 32 Nibbles = 0010 0000
    IDData.ManufacturerCode = 0x80;    // Continental = 1000 0000
    IDData.SensorType = 0x61;          // High g = XXXX 0001, Low g = XXXX 0010
    IDData.SensorParameter = 0x21;     // Example Value
    IDData.SensorManCode = 0x21;       // Example Value
    IDData.VehicleManCode = 0x21;      // Example Value
    IDData.SensorProdYear = 0x21;      // Example Value
    IDData.SensorProdMonth = 0x21;     // Example Value
    IDData.SensorProdDay = 0x21;       // Example Value

    if (MIO_M406_Config (Slot_PSI5) < 0) return RT_Failure;
    /* Configure the master */
}
```

```

MIO_M406_MasterConfigure (Slot_PSI5,
    cfgstring, /* see above */
    ts,        /*beginning of the time slots */
    0.000016, /* duration of sync pulse */
    0);
MIO_M406_MasterEnableTS (Slot_PSI5, 0x0f); /* Enable the 4 time slots */
/* Configure the slaves */
for (s=0; s<4; s++) {
    MIO_M406_SlaveConfigureChannel (Slot_PSI5, s, cfgstring, ts);
    MIO_M406_SlaveSetInitData (Slot_PSI5, s,
        4,           /* Number of sensors per channel */
        SensorReady, /* Sensors state, which is sent during initialization phase "3". */
        &IDData,    /* PSI5-Spec S. 4 */
        0.065,      /* dPH1: Duration of initialization phase "1" in seconds */
        4,          /* nPH2: Number of repetitions during initialization phase "2" */
        16         /* nPH3: Number of repetitions during initialization phase "3" */ );
    IO.SlaveTxData[s] = (0x1111 << s), /* channel0 sends 0x1111, channel1 sends 0x2222,
                                            a.s.o. */
    MIO_M406_SlaveSetTxData(Slot_PSI5,
        s,
        s /* Data for Timeslot s: Channel0 sends in timeslot 0, Channel1 in timeslot1,
            a.s.o. */,
        IO.SlaveTxData[s],
        0 /* Datal is only needed in multiplexed data mode*/);
    MIO_M406_SlaveEnableSensor (Slot_PSI5, s, (0x01 << s));
}
}

void IO_In (void)
{
    int s /* sensor, here also channel and timeslot */;
    int tmp1, tmp2;
    unsigned int data;

    tmp1 = MIO_M406_MasterGetError(Slot_PSI5);
    tmp2 = MIO_M406_MasterGetSensorState(Slot_PSI5);
    for (s=0; s<4;s++) {
        IO.MasterIsError[s] = (tmp1>>s) & 0x01;
        IO.MasterSensorState[s] = (tmp2 >>(s*4)) & 0x0f;
    }

    for (s = 0; s< 4; s++) {
        while (( tmp1 = MIO_M406_MasterGetData(Slot_PSI5, s, &data)) != 0) {
            IO.MasterDataAvailable[s] = tmp1;
            IO.MasterRxData[s] = data;
        }
    }
}

void IO_Out (void)
{
    int s /* sensor, here also channel and timeslot */;

    for (s=0; s<NCHANNELS; s++) {
        MIO_M406_SlaveSetTxData(Slot_PSI5, s /* channel */,
            s /* sensor = timeslot */,
            IO.SlaveTxData[s],
            0 /* Datal is only needed in multiplexed data mode*/);
    }
}

```

10.4.21 M407: SPI Interface, 2 Slaves or 1 Master

Connector assignment of M407 and the LVDS-Connector of the LVDS2SPI-Adapter

Pin	Signal Description		Pin	Signal Description	
	Master Mode	Slave Mode		Master Mode	Slave Mode
1	CLK Out(+) 14	CLK0 In(+) 13	14	CLK Out(-)	CLK0 In(-)
2	SS0 Out(+) 14	SS0 In(+) 13	15	SS0 Out(-)	SS0 In(-)
3	SS1 Out(+) 14	MOSI0 In(+) 13	16	SS1 Out(-)	MOSI0 In(-)
4	SS2 Out(+) 14	CLK1 In(+) 13	17	SS2 Out(-)	CLK1 In(-)
5	SS3 Out(+) 14	SS1 In(+) 13	18	SS3 Out(-)	SS1 In(-)
6	MOSI Out(+) 14	MISO0 Out(+) 13	19	MOSI Out(-)	MISO0 Out(-)
7	MISO In(+) 14	MOSI1 In(+) 13	20	MISO In(-)	MOSI1 In(-)
8	SS4 Out(+) 14	MISO1 Out(+) 13	21	SS4 Out(-)	MISO1 Out(-)
9	SS Cfg Out(+) 14		22	SS Cfg Out(-)	
10	not connected		23	+5V (Out)	
11	+5V (Out)		24	+3.3V (Out)	
12	+3.3V (Out)		25	GND	
13	GND				

Connector assignment of the Master-Connector of the LVDS2SPI-Adapter

Female SubD-15 Connector

Pin	Signal Description	Pin	Signal Description
1	CLK Out 9	9	GND
2	SS0 Out 15	10	GND
3	SS1 Out 15	11	GND
4	SS2 Out 15	12	GND
5	SS3 Out 15	13	GND
6	SS4 Out 15	14	GND
7	MISO In 15	15	GND
8	MOSI Out 15		

Connector assignment of the Slave-Connectors of the LVDS2SPI-Adapter

2x Female SubD-9 Connector

Pin	Signal Description	Pin	Signal Description
1	Clk In 6	6	GND
2	SS In 9	7	GND
3	MISO Out 9	8	GND
4	MOSI In 9	9	GND

Pin	Signal Description	Pin	Signal Description
5	not connected		

Master Signals

- SSn Out is the low active slave select signal from the master
- CLK Out is the clock signal of the master
- MOSI Out is the data output port of the master (master => slave)
- MISO In is the data input port of the master (slave => master)
- SS Cfg is the configuration select pin of the adapter

Slave Signals

- SSn In are the slave select signals from the slave n
- CLKn In is the clock signal input of the slave n
- MOSIn In is the data input port of the slave (master => slave)
- MISOn Out is the data output port of the slave (slave => master)
- SS Cfg is the configuration select pin of the adapter

Function Overview

The M407 needs the A24D32 address space to work.

Initialization and Configuration

- `MIO_M407_Config()`
- `MIO_M407_UnitConfig()`
- `MIO_M407_GetAdapterID()`
- `MIO_M407_MasterConfigure()`
- `MIO_M407_MasterSetBaudrate()`
- `MIO_M407_MasterSetCycleTime()`
- `MIO_M407_MasterSetDelaySS2Clk()`
- `MIO_M407_SlaveConfigure()`
- `MIO_M407_SlaveSetMbxMasks()`
- `MIO_M407_SlaveSetMbxAccess()`

Runtime Functions

- `MIO_M407_MasterSetTxData()`
- `MIO_M407_MasterGetDataAvailable()`
- `MIO_M407_MasterGetRxData()`
- `MIO_M407_MasterResetFIFO()`
- `MIO_M407_SlaveSetTxData()`
- `MIO_M407_SlaveGetDataAvailable()`
- `MIO_M407_SlaveGetRxData()`
- `MIO_M407_SlaveResetFIFO()`
- `MIO_M407_SlaveSetMbx()`
- `MIO_M407_SlaveGetMbx()`
- `MIO_M407_FaultSim_Configure()`

- [MIO_M407_FaultSim_Enable\(\)](#)
- [MIO_M407_FaultSim_GetCounter\(\)](#)
- [MIO_M407_FaultSim_ResetFaults\(\)](#)

enum eMIO_M407_TXMode

```
typedef enum {
    M407_TxDisabled = 0x0,
    M407_TxManually,
    M407_TxAuto,
    M407_TxAlways
} eMIO_M407_TXMode;
```

Description

This enumerator is used for [MIO_M407_MasterConfigure\(\)](#).

- M407_TxDisabled
The master does not initiate any communication.
- M407_TxManually
The master initiates the transmission if a write access to the Transmission Register occurs via [MIO_M407_MasterSetTxData\(\)](#). Exactly one transmission is done even if there are many data words in the TxFIFO.
- M407_TxAuto
The master initiates the transmission as long as the TxFIFO is not empty.
- M407_TxAlways
The master transmits data always. If the FIFO is empty the last data word is repeated until new data is written into the FIFO or the module is switched off.

MIO_M407_Config()

```
int MIO_M407_Config (int Slot);
```

Description

By calling the function [MIO_M407_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M407 is specified. This function must be called one time, before any attempted access to the module.

Initial state of the module:

- Message FIFOs are empty
- All Channels are disabled
- Nothing is sent
- Adapter (if available) is configured as slave with the IO-Voltage set to 0V.

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

MIO_M407_UnitConfig()

```
void MIO_M407_UnitConfig (int Slot, int Unit, int IsMaster);
```

Description

This function configures the direction of the LVDS lines of the M407. This means it specifies if the module is configured as master or as slave.

Configuring the unit(s) is only available if not IPG's LVDS2SPI adapter is used and thus the

configuration of the adapter is not recognised.

If the IPG adapter is recognized the module is configured by the adapter.

Parameters

Have changed in CarMaker 4.0.6.

- Unit
Must be 0. Further units are probably available in the future.
- IsMaster
If “1” the module as well as the adapter is configured as SPI master.
If “0” the module and the adapter is configured as an SPI slave.

MIO_M407_GetAdapterID()

```
int MIO_M407_GetAdapterID (int Slot, int Unit, int ReInit);
```

Description

This function returns the ID of the LVDS2SPI adapter and if it is configured as master / slave.

If the IPG LVDS2SPI-Adapter is connected the return value should be 0x89. Otherwise the value is undefined.

Parameters

- Relinit
If set to “1” the Adapter-ID the module re-reads the information from the adapter and reconfigures the module depending on the information of the adapter.
The function returns as soon as the reconfiguration of the module has finished.
Reconfiguration only works with hardware revisions >= 1.2.

MIO_M407_MasterConfigure()

```
void MIO_M407_MasterConfigure (int Slot, int Channel, eMIO_M407_TXMode TxMode,  
                               int PropagationDelay, int CPHA, int CPOL, int lsbfirst);
```

Description

This function configures the behavior of the master.

Parameters

- TxMode
see [enum eMIO_M407_TXMode](#)
- PropagationDelay
This configures the delay on the whole data path. The propagation delay is the duration from the time the master internally generates the SPI clock to the time the master receives the corresponding data from the slave.

Configuring the delay is important for very high baud rates because if delay on the data path is longer than half the period time, the master will receive erroneous data.

- cpha / cpol
 - CPHA (clock phase) specifies on which clock edge the data is read from and put to the bus.

- CPOL (clock polarity) defines the idle state of the clock signal.

The following table shows the active edge and the clock idle state depending on cpol and cpha:

CPOL	CPHA	Clock Idle State	Active Edge
0	0	low	rising
0	1	low	falling
1	0	high	falling
1	1	high	rising

- lsbfirrst

If “1” the LSB is sent first, if “0” the MSB is sent first.

MIO_M407_MasterSetBaudrate()

```
void MIO_M407_MasterSetBaudrate (int Slot, int Channel, double Baudrate);
```

Description

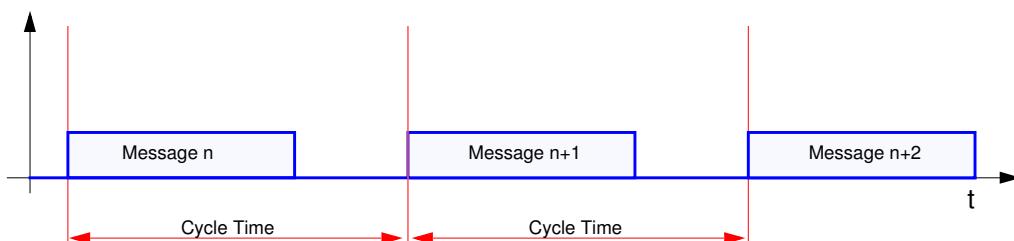
This function configures the frequency of the clock signal which is generated by the master. The unit is Baud and can be configured in the range from 0Baud .. 40MBaud. By default this value is set to 1MBaud.

MIO_M407_MasterSetCycleTime()

```
void MIO_M407_MasterSetRepetition (int Slot, int Channel, double CycleTime);
```

Description

The cycle time is the time between 2 subsequent messages as shown in the following figure.



Parameters

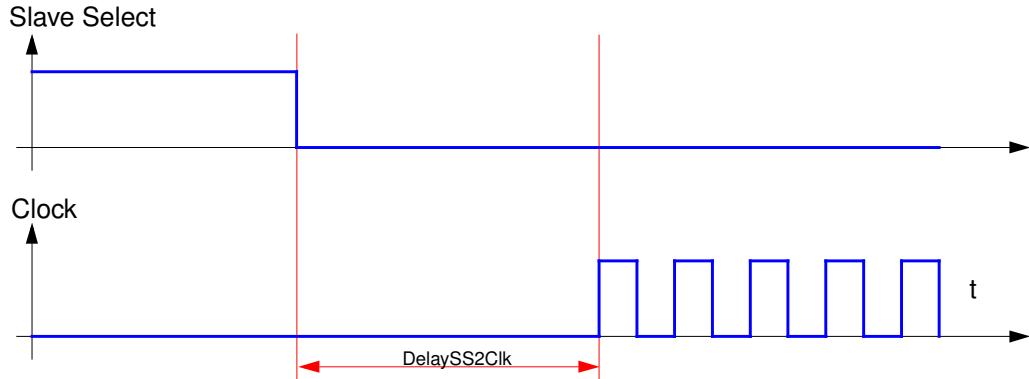
The unit of the cycle time is seconds. The valid value range is 0 .. 42seconds in steps of 10ns.

MIO_M407_MasterSetDelaySS2Clk()

```
void MIO_M407_MasterSetDelaySS2Clk (int Slot, int Channel, double Delay);
```

Description

Here you can configure the delay from the time the master pulls the select signal low to the time the first clock pulse starts.



Parameters

The unit of the delay is seconds. The valid value range is 0 .. 42seconds in steps of 10ns.

MIO_M407_MasterSetTxData()

```
void MIO_M407_MasterSetTxData (int Slot, int Channel, int SlaveNo, int nBits,
                               long long TxData);
```

Description

This function writes data that should be transmitted into the Tx-FIFO of the master. The selected TxMode specifies, when the data then is sent.

The Tx-FIFO of the master has a size of 128 messages. This means that the next 128 messages can be configured in advance.

MIO_M407_MasterGetDataAvailable()

```
int MIO_M407_MasterGetDataAvailable (int Slot, int Channel, int* TxBufLevel);
```

Description

[MIO_M407_MasterGetDataAvailable\(\)](#) returns the number of words available in the Rx-FIFO and if desired also the level of the Tx-FIFO.

Parameters

- **TxBufLevel**
If TxBufLevel is “NULL”, the parameter has no effect. Otherwise the parameter contains the level of the Tx-FIFO.

Return Value

The function returns the level of the Rx-FIFO.

MIO_M407_MasterGetRxData()

```
int MIO_M407_MasterGetRxData (int Slot, int Channel, long long *RxData, int *RxSlave,  
    int *RxnBits);
```

Description

[MIO_M407_MasterGetRxData\(\)](#) reads the data from the Rx-FIFO of the master.

The Rx-FIFO of the master has a size of 128 messages. This means up to 128 messages can be stored between two times of reading the FIFO.

Parameters

- RxData
After calling the function RxData contains the value read from the FIFO. The data is located right-aligned in the register.
- RxSlave
After the function is called this parameter contains the slave which the data is received from. This is identical with the activated /CS line during data reception.
If you don't need this information, set this value to NULL.
- RxnBits
After the function is called this parameter contains the number of data bits that are received. If you don't need this information, set this value to NULL.

Return Value

The function returns the number of data words remaining in the FIFO after calling this function. The return value can be used as an exit condition in loops.

If the Rx-FIFO is empty and thus the content of RxData is not valid, the function returns “-1”.

MIO_M407_MasterResetFIFO()

```
void MIO_M407_MasterResetFIFO (int Slot, int Channel);
```

Description

This function resets the Rx-FIFO and the Tx-FIFO of the specified slave channel. A reset must be done after a FIFO overrun to ensure a consistent behavior of the FIFOs.

MIO_M407_SlaveConfigure()

```
void MIO_M407_SlaveConfigure (int Slot, int Channel, int Enable, int cpha, int cpol,  
    int lsbfirst, int nBits, PropagationDelay, int Sensortype, int SensorVariant);
```

Description

This function configures the behavior of the slave.

Parameters

- Enable
If the slave is not enabled it does nothing. To enable it set the bit to '1'.
- cpha / cpol
 - CPHA (clock phase) specifies on which clock edge the data is read from and put to the bus.
 - CPOL (clock polarity) defines the idle state of the clock signal.

The following table shows the active edge and the clock idle state depending on cpol and cpha:

CPOL	CPHA	Clock Idle State	Active Edge
0	0	low	rising
0	1	low	falling
1	0	high	falling
1	1	high	rising

- lsbffirst
If "1" the LSB is sent first, if "0" the MSB is sent first.
- nBits
This is the number of bits the slave receives and transmits. Of course the real number of bits depends on the number of clock pulses that are recognized by the slave. If MSB first is selected the message length is used to decide, which is the first bit to be sent.
- PropagationDelay
With high baudrates, it may happen, that the propagation delay of the clock signal and the resulting delay in transmission of the data slave, causes problems in the master to receive the slave data. With this parameter it is possible to reduce or even eliminate the propagation delay. The value to specify here is the delay of the clock in the slave path (SPI2LVDS(Adapter) => LVDS2SPI (M407) => FPGA(M407)). With IPG's LVDS2SPI adaptor this value is 60ns. The unit is nanoseconds.
- Sensortype / SensorVariant
The sensortype and the sensor variant switch on / off the several sensor emulations, that are implemented in the M407.
The following table show which values configure which sensor emulation. Please note that several sensor emulations are only available for specific customers. Please ask IPG if an additional emulation is needed.

Sensor Type	Sensor Variant	Sensor Name	Alternative Sensor Name	Number of Axis	Mailbox / FIFO-Mode
0	0	no emulation, just SPI (default)			FIFO
	1				Mailbox
1	0	SMA550	ASG4 shrink	1	Mailbox
	1	SMA560		2	Mailbox
2	0	SMA660	ASG5.1	1	Mailbox
	1			2	Mailbox
3	0	CISS2	CISS2	1LF + 1HF	Mailbox
	1			2LF + 1HF	Mailbox
4	0		COMBO2.2	1	Mailbox
	1			2	Mailbox

Sensor Type	Sensor Variant	Sensor Name	Alternative Sensor Name	Number of Axis	Mailbox / FIFO-Mode
5	0	Sycamore	ASG4.2	1	Mailbox
	1			2	Mailbox

In the FIFO-Mode the Tx data of the slave is taken from the FIFO as soon as the /CS signal of the SPI bus goes low. If the FIFO is empty the last data is send again and again.

In the Mailbox Mode the Tx data of the slave is taken from the mailboxes. Therefore at least one bit of the MOSI message has to be configured as a mailbox address. When / CS goes low, the current data is read from the addressed mailbox.

MIO_M407_SlaveSetMbxMasks()

```
int MIO_M407_SlaveSetMbxMasks (int Slot, int Channel,
                               unsigned char WriteBit, unsigned char WriteLevel,
                               unsigned char AddrNBits, unsigned char AddrLSB);
```

Description

This function configures the address mask, the data mask and the write bit. Those are needed for the mailbox mode for correctly addressing the mailboxes.

Parameters

Have changed in CarMaker 4.0.6

- **WriteBit**

The write bit is the bit which is responsible for the write command. If the write bit is set to e.g. “0” the LSB is the write bit regardless of the value set for lsbfirst in [MIO_M407_SlaveConfigure\(\)](#).

- **WriteLevel**

This configures if the write-command is active high or active low.

- WriteLevel = “1” means that if the write bit is “1” the message contains a write command and if the write bit is “0” the message contains a read command
- WriteLevel = “0” means that if the write bit is “1” the message contains a read command and if the write bit is “0” the message contains a write command

- **AddrNBits and AddrLSB**

In the Mailbox mode these values specify, which bytes of a received message contain the address of the mailbox.

The module has 256 mailboxes for each slave. Therefore AddrNBits must be lower or equal 8.

If for example the message bits 23..17 contain the address of the mailbox, configure AddrLSB to 17 and AddrNBits to 7 (=23-17+1)

MIO_M407_SlaveSetMbxAccess()

```
void MIO_M407_SlaveSetMbxAccessMask (int Slot, int Channel, int MailboxAddress,
                                      int IsWriteable, int IsReadable);
```

Description

With this function it is specified if a mailbox is readable or writeable via the SPI bus.

Via the M-Module bus the mailboxes always are readable and writable.

Parameters

- IsWriteable
If set to "1" the corresponding mailbox can be written from the SPI bus.
- IsReadable
If set to "1" the corresponding mailbox can be read from the SPI bus.

MIO_M407_SlaveSetTxData()

```
void MIO_M407_SlaveSetTxData (int Slot, int Channel, unsigned int TxData);
```

Description

If the FIFO Mode is selected the function sets the data that the slave should transmit. Each time the slave's select pin changes from '1' to '0' the data is fetched from the FIFO and then transmitted.

The Tx-FIFO of the slaves have a size of 32 messages. This means that the next 32 messages can be configured in advance.

MIO_M407_SlaveGetDataAvailable()

```
int MIO_M407_SlaveGetDataAvailable (int Slot, int Channel, int* TxBufLevel);
```

[MIO_M407_SlaveGetDataAvailable\(\)](#) returns the number of words available in the Rx-FIFO and if desired also the level of the Tx-FIFO.

Parameters

- TxBufLevel
If TxBufLevel is "NULL", the parameter has no effect. Otherwise the parameter contains the level of the Tx-FIFO.

Return Value

The function returns the level of the Rx-FIFO.

MIO_M407_SlaveGetRxData()

```
int MIO_M407_SlaveGetRxData (int Slot, int Channel, unsigned int *RxData);
```

Description

[MIO_M407_SlaveGetRxData\(\)](#) reads the data from the Rx-FIFO of the slave. Each received message is written into the Rx-FIFO independent from the transmission mode, the slave is using. The Rx-FIFO of the slaves have a size of 128 messages. This means up to 128 messages can be stored between two times of reading the FIFO.

Parameters

- RxData

After calling the function RxData contains the value read from the FIFO. The data is located right-aligned in the register.

Return Value

The function returns the number of data words remaining in the FIFO after calling this function. The return value can be used as an exit condition in loops.

If the Rx-FIFO is empty and thus the content of RxData is not valid, the function returns “-1”.

MIO_M407_SlaveResetFIFO()

```
void MIO_M407_SlaveResetFIFO (int Slot, int Channel);
```

Description

This function resets the Rx-FIFO and the Tx-FIFO of the specified slave channel. A reset must be done after a FIFO overrun to ensure a consistent behavior of the FIFOs.

MIO_M407_SlaveSetMbx()

```
int MIO_M407_SlaveSetMbx (int Slot, int Channel, unsigned int MbxIndex, unsigned int Data);
```

Description

This function sets the content for the specified mailbox.

Parameters

- MbxIndex

Addresses the Mailbox where the data is to be written to.

- Data

This contains the data which should be written into the specified mailbox. The data must be written right-aligned into the register.

Return Value

The function returns “-1” if the MbxIndex is out of range.

MIO_M407_SlaveGetMbx()

```
int MIO_M407_SlaveGetMbx (int Slot, int Channel, unsigned int MbxAddress,  
                           unsigned int *Data);
```

Description

This function reads the content from the specified mailbox.

Parameters

- MbxIndex

Addresses the Mailbox where the data is to be read from.

- Data

After calling the function Data contains the value read from the mailbox. The data is located right-aligned in the register.

Return Value

The function returns “-1” if the MbxIndex is out of range.

MIO_M407_FaultSim_Configure()

```
void MIO_M407_FaultSim_Configure(int Slot, int Channel, int MailboxAddress,  
                                int FaultData, int nFaults, eM407_FaultType FaultType);
```

Description

Since Rev 1.2 a fault simulation is implemented for the mailbox mode. This can be used to send erroneous data for a specified number of SPI cycles. Therefore there is a second set of mailboxes, called the “error mailboxes”, a counter and a flag to activate the error simulation of all mailboxes which are configured to have errors.

[MIO_M407_FaultSim_Configure\(\)](#) configures a Mailbox to send the data from the error mailbox instead of the default mailbox.

This does not activate the sending of the error data but only configures the error data itself. Sending the error data is only active as long as fault simulation is enabled and the Counter (nFaults) is greater 0. Otherwise the data from the default mailbox is sent.

Parameters

- MailboxAddress
This is the address of the mailbox to be configured as erroneous. Up to 256 mailboxes (0..255) can be configured to be erroneous at the same time.
- FaultData
The erroneous data.
- nFaults
This specifies how often the erroneous data is sent.
- FaultType
Must be 0. Reserved for future use.

Return Value

MIO_M407_FaultSim_Enable()

```
void MIO_M407_FaultSim_Enable(int Slot, int Channel, int Enable);
```

Description

[MIO_M407_FaultSim_Enable\(\)](#) enables and disables the transmission of erroneous data. There are two possible ways to configure errors.

- The fault simulation is always enabled
This means that any erroneous mailbox data is send immediately when the [MIO_M407_FaultSim_Configure\(\)](#) for a this mailbox is called with the parameter nFaults > 1.
- The fault simulation is enabled as soon as all desired mailboxes are configured to be erroneous. In this case the error simulation starts at the same time for all mailboxes.

Parameters

- Enable

Setting this value to 1 starts the transmission of erroneous data of all configured registers. The fault data thereafter is send as long as the fault counter is >0. If the counter is 0 the data from the default mailbox is sent. If the counter of all erroneous mailboxes is 0 the user must set Enable to “0” before starting a new fault simulation session.

MIO_M407_FaultSim_GetCounter()

```
int MIO_M407_FaultSim_GetCounter(int Slot, int Channel, int MailboxAddress);
```

Description

This function returns the error counter of a mailbox. If the counter is 0 for all erroneous mailboxes

Parameters

- MailboxAddress

This is the address of the mailbox to be configured as erroneous. Up to 256 mailboxes (0..255) can be configured to be erroneous at the same time.

Return Value

The error counter of a mailbox

MIO_M407_FaultSim_ResetFaults()

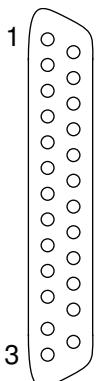
```
void MIO_M407_FaultSim_ResetFaults(int Slot, int Channel);
```

Description

This function sets the error counter of all mailboxes of a channel to “0”.

10.4.22 M408: PWM Meter and SENT Receiver, 20 Channels

Connector assignment



Pin	Signal Description	Pin	Signal Description
1	Input Channel 0	14	Input Channel 1
2	Input Channel 2	15	Input Channel 3
3	Input Channel 4	16	GND1
4	Input Channel 5	17	Input Channel 6
5	Input Channel 7	18	Input Channel 8
6	Input Channel 9	19	GND1
7	Input Channel 10	20	Input Channel 11
8	Input Channel 12	21	Input Channel 13
9	Input Channel 14	22	GND2
10	Input Channel 15	23	Input Channel 16
11	Input Channel 17	24	Input Channel 18
12	Input Channel 19	25	GND2
13	n.c.		

Function Overview

If the A24D32 address space is available on the used carrier board it is used. Otherwise the A8D16 address space is used.

Initialization and Configuration

- [MIO_M408_Config\(\)](#)
- [MIO_M408_SetThreshold\(\)](#)
- [MIO_M408_SetTimeout\(\)](#)
- [MIO_M408_ResetPeriodCounter\(\)](#)

Runtime Functions

- [MIO_M408_GetBinaryIn\(\)](#)
- [MIO_M408_GetBetweenThresholds\(\)](#)
- [MIO_M408_GetEventRising\(\)](#)
- [MIO_M408_GetEventFalling\(\)](#)
- [MIO_M408_GetTimeBase\(\)](#)
- [MIO_M408_GetTimeout\(\)](#)
- [MIO_M408_GetPeriod\(\)](#)
- [MIO_M408_GetFrequency\(\)](#)
- [MIO_M408_GetDutyCycle\(\)](#)
- [MIO_M408_GetHighTime\(\)](#)
- [MIO_M408_GetPeriodCounter\(\)](#)
- [MIO_M408_GetHighTimePeriod16\(\)](#)
- [MIO_M408_SENTConfig\(\)](#)
- [MIO_M408_SENTRstFIFO\(\)](#)
- [MIO_M408_SENTGetData\(\)](#)

- [MIO_M408_SENTGetSerialData\(\)](#)
- [MIO_M408_SENTGetStatus\(\)](#)

Examples

- [PWM Input from one rising edge to the next one](#)
- [SENT Receiver Example](#)

MIO_M408_Config()

```
int MIO_M408_Config (int Slot);
```

Description

By calling the function `MIO_M408_Config()` the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M408 is specified. This function must be called one time, before any attempted access to the module.

Initial state of the module:

- Lower threshold voltage: 2.0V
- Upper threshold voltage: 3.0V
- Timeout is set to 20s

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

MIO_M408_SetThreshold()

```
void MIO_M408_SetThreshold (int Slot, int Channel, float UHigh, float ULow,  
int EdgeMode, int EnDecreasingFreq);
```

Description

This function configures the threshold voltages, the edge mode and if decreasing frequencies should be detected before a delayed edge is recognized. The threshold voltages can be in a range from 0.2V to 25.27V.

Parameters

- UHigh / ULow

These are the threshold voltages of each channel. If the input signal voltage is above UHigh the input signal is detected as “1”. If the input signal voltage is below ULow the input signal is detected as “0”. In any other case the input keeps its state.
Condition: UHigh - ULow <= 7.5V

- EdgeMode

If EdgeMode is set to “0” the cycle duration is measured from the rising edge to the subsequent rising edge. If EdgeMode is set to “1” the cycle duration is measured from the falling edge to the subsequent falling edge.

For measuring duty cycles the edge mode has to be configured to the leading edge of the signal period. Otherwise changing the duty cycle will also change the measured frequency, compromising the results.

- EnDecreasingFreq

If EnDecreasingFreq is set to “1” decreasing frequencies will be detected immediately. In this case if the time since the last period is longer than the cycle duration time of the last period the cycle duration is calculated from the time since the last period instead from the cycle duration of the last period.

If EnDecreasingFreq=“0” the value which is presented in the Period Register is the time of the last period as long as no new active edge and no timeout happens.



MIO_M408_SetTimeout()

```
void MIO_M408_SetTimeout (int Slot, int Channel, double Timeout);
```

Description

If no signal edge is detected on a channel for more than the time configured by this function the channel is set into timeout mode. In the timeout mode the cycle duration is set to about 43s or respectively a frequency of about 0.023Hz. Additionally the timeout flag is set for the mentioned channel in the *Global Timeout Register* which can be read by [MIO_M408_GetTimeout\(\)](#).

After initialization the timeout value is set to 0x7fffffff. This corresponds to 21 seconds.

The value range for timeout is 0.0001s to 32.0s.

MIO_M408_ResetPeriodCounter()

```
void MIO_M408_ResetPeriodCounter (int Slot, int Channel);
```

Description

By calling this function the period counter of the specified channel is set to 0x00000000.

MIO_M408_GetBinaryIn()

```
unsigned int MIO_M408_GetBinaryIn (int Slot);
```

Description

This function returns the actual state of the signal inputs. A logical “1” thereby means that the input voltage of the channel is above the upper threshold voltage as against a logical “0” means the input voltage of the channel is below the lower threshold.

Bit 0 correspond to channel 0, bit 1 corresponds to channel 1 and so on.

MIO_M408_GetBetweenThresholds()

```
unsigned int MIO_M408_GetBetweenThresholds (int Slot);
```

Description

This function returns the actual state of the signal inputs. A logical “1” thereby means that the input voltage of the channel is between the two threshold voltages as against a logical “0” means the input voltage of the channel is either above the upper threshold or below the lower threshold.

Bit 0 correspond to channel 0, bit 1 corresponds to channel 1 and so on.

MIO_M408_GetEventRising()

```
unsigned int MIO_M408_GetEventRising (int Slot);
```

Description

This function returns a channel mask which displays if either a rising edge happened on the channels since the last call of this function or not. A logical “1” thereby means that a rising edge happened, a logical “0” means that no rising edge happened on the corresponding channel. This function can be used for detecting short trigger signals.

Bit 0 correspond to channel 0, bit 1 corresponds to channel 1 and so on.

MIO_M408_GetEventFalling()

```
unsigned int MIO_M408_GetEventFalling (int Slot);
```

Description

This function returns a channel mask which displays if either a falling edge happened on the channels since the last call of this function or not. A logical “1” thereby means that a falling edge happened, a logical “0” means that no falling edge happened on the corresponding channel. This function can be used for detecting short trigger signals.

Bit 0 correspond to channel 0, bit 1 corresponds to channel 1 and so on.

MIO_M408_GetTimeBase()

```
unsigned int MIO_M408_GetTimeBase (int Slot);
```

Description

This function returns the 32bit value of the internal time base counter of the module.

MIO_M408_GetTimeout()

```
unsigned int MIO_M408_GetTimeout (int Slot);
```

Description

If no signal edge is detected on a channel for more than the time configured by [MIO_M408_SetTimeout\(\)](#) this channel is set into timeout mode and the timeout flag is then set for this channel. [MIO_M408_GetTimeout\(\)](#) returns a channel mask in which the bits of all channels which are in timeout mode are set to “1”.

Bit 0 correspond to channel 0, bit 1 corresponds to channel 1 and so on.

MIO_M408_GetPeriod()

```
double MIO_M408_GetPeriod (int Slot, int Channel);
```

Description

This function returns the cycle duration of the specified channel in seconds.

MIO_M408_GetFrequency()

```
double MIO_M408_GetFrequency (int Slot, int Channel);
```

Description

This function returns the frequency of the specified channel in Hertz.

MIO_M408_GetDutyCycle()

```
double MIO_M408_GetDutyCycle (int Slot, int Channel);
```

Description

this function returns the pulse pause relationship of the specified channel as a value between 0.0 and 1.0. The returned value corresponds to (HighTime / cycle duration). This means, the longer the high time is against the cycle duration, the higher is the value.

There is a synchronization between frequency (or cycle duration) and duty cycle so that if first the frequency (or cycle duration) is read by [MIO_M408_GetFrequency\(\)](#) or [MIO_M408_GetPeriod\(\)](#) and then the duty cycle the returned duty cycle is the one which belongs to the frequency (or cycle duration) read before.

MIO_M408_GetHighTime()

```
double MIO_M408_GetHighTime (int Slot, int Channel);
```

Description

This function returns the duration of the high state of a channel within the corresponding cycle duration.

There is a synchronization between frequency (or cycle duration) and high time so that if first the frequency (or cycle duration) is read by [MIO_M408_GetFrequency\(\)](#) or [MIO_M408_GetPeriod\(\)](#) and then the high time the returned high time is the one which belongs to the frequency (or cycle duration) read before.

MIO_M408_GetPeriodCounter()

```
unsigned int MIO_M408_GetPeriodCounter (int Slot, int Channel);
```

Description

This function returns the period counter of the specified channel. The period counter is incremented on the beginning of each period.

MIO_M408_GetHighTimePeriod16()

```
void MIO_M408_GetHighTimePeriod16 (int Slot, int Channel, double* Period, double* HighTime);
```

Description

`MIO_M408_GetHighTimePeriod16()` returns the high time and the cycle duration by only one 32bit access onto the module.

This halves the needed access time to the module compared to the use of `MIO_M408_GetFrequency()` or `MIO_M408_GetPeriod()` and `MIO_M408_GetHighTime()`. Due to the range of the registers this function can only be used if the to be measured frequency is higher than about 800Hz respectively the cycle duration is lower than 1.25ms.

However using this function means that the resolution is only 20ns instead of 10ns when using `MIO_M408_GetFrequency()`, `MIO_M408_GetPeriod()` and `MIO_M408_GetHighTime()`. But for typical PWM signals with a frequency of about 1kHz this is usually sufficient.

MIO_M408_GetDutyCyclePeriod16()

```
void MIO_M408_GetDutyCyclePeriod16 (int Slot, int Channel, double* Period, double* DutyCycle);
```

Description

`MIO_M408_GetDutyCyclePeriod16()` returns the duty cycle and the cycle duration by only one 32bit access onto the module.

This halves the needed access time to the module compared to the use of `MIO_M408_GetFrequency()` or `MIO_M408_GetPeriod()` and `MIO_M408_GetDutyCycle()`. Due to the range of the registers this function can only be used if the to be measured frequency is higher than about 800Hz respectively the cycle duration is lower than 1.25ms.

However using this function means that the resolution of the frequency is only 20ns instead of 10ns when using `MIO_M408_GetFrequency()` and `MIO_M408_GetPeriod()`. But for typical PWM signals with a frequency of about 1kHz this is usually sufficient.

The duty cycle has the same accuracy as when using `MIO_M408_GetDutyCycle()` because it is calculated from the 32 bit values of high time and cycle duration.

MIO_M408_SENTConfig()

```
void MIO_M408_SENTConfig (int Slot, int Channel, int Enable, int nNibblesExp,
                           int WithPausePulse, float TickTime, int EnhancedMsgFormat);
```

Description

`MIO_M408_SENTConfig()` configures a channel as SENT receiver.

Parameters

- **Channel**
A maximum of 8 receiver units are available. Each unit can be assigned to any channel of the M408. The assignment is done automatically when `MIO_M408_SENTConfig()` is called with a channel that was not previously configured as SENT channel.
If more than 8 SENT units are to be configured, the function generates an error message in the CarMakers Log.
The SENT functions of the M408 manage the assignment between channel and SENT unit internally. Therefore the SENT-API of the M408 offers the channel instead of the SENT unit.
- **Enable**
Enables / Disables the channel as a SENT channel.
- **nNibblesExp**
Number of expected Nibbles

- WithPausePulse
'1' if message frame contains a pause pulse
- TickTime
This is the time one "tick" within a SENT frame takes. The time must be given in seconds.
- EnhancedMsgFormat
EnhancedMsgFormat switches between the standard message format and the enhanced message format. The message format affects only the serial information within the communication nibble of a frame.

MIO_M408_SENTRstFIFO()

```
void MIO_M408_SENTRstFIFO (int Slot, int Channel);
```

Description

[MIO_M408_SENTGetSerialData\(\)](#) clears the RxFIFO of the specified SENT-channel. This should be done immediately after the configuration is done, or if [MIO_M408_SENTGetStatus\(\)](#) returns with -1 (buffer full).

MIO_M408_SENTGetData()

```
int MIO_M408_SENTGetData (int Slot, int Channel, unsigned char Data[6], unsigned char *CRC,  
unsigned char *COM);
```

Description

[MIO_M408_SENTGetSerialData\(\)](#) returns the received SENT messages from the Rx-FIFO.

Parameters

- Data[6]
data is an array with 6 elements. After the function was called, Data contains the data nibbles of the received message if the FIFO was not empty.
 - Data[0] contains data nibble 0
 - Data[1] contains data nibble 1, and so on up to
 - Data[5] contains data nibble 5
- After the function was called, CRC contains the value of the received CRC nibble of the message
- After the function was called, COM contains the value of the received "Communication and Status Nibble" value of the message

Return Value

- -1: Data is not valid, because no data is available
- Other values: Number of received nibbles

MIO_M408_SENTGetSerialData()

```
unsigned int MIO_M408_SENTGetSerialData (int Slot, int Channel);
```

Description

The SENT protocol contains a serial information bit in each message which is sent within the communication nibble and contains information about the sensor. The information data thus is splitted in several subsequent messages.

[MIO_M408_SENTGetSerialData\(\)](#) returns the merged information data as follows:

31	24	23	16	15	0
Message ID		Serial CRC			Serial Data

MIO_M408_SENTGetStatus()

```
int MIO_M408_SENTGetStatus (int Slot, int Channel);
```

Description

This function returns the state of the SENT Rx-FIFO of the M408.

Return Value

- 1 - Buffer full, either read all data from the FIFO or call [MIO_M408_SENTRstFIFO\(\)](#).
- 0 - No data available
- 1 - Data available

MIO_M408_SENTGetTicktime()

```
float MIO_M408_SENTGetTicktime (int Slot, int Channel);
```

Description

[MIO_M408_SENTGetTicktime\(\)](#) returns the synchronized ticktime which is used for the specified channel. The SENT receiver calculates the ticktime of the SENT frames after the synchronization pulse is received and uses the synchronized ticktime for the remaining frame. The receiver is able to “catch” the real ticktime within a range of +/-10% of the ticktime configured by [MIO_M408_SENTConfig\(\)](#).

Examples

PWM Input from one rising edge to the next one

```
#define NCHANNELS    20
#define Slot_PWMIn 1

struct {
    double Frequency;
    double DutyCycle;
    int TO;
```

```

        int PeriodCounter;
    } tIO;
tIO IO;

int IO_Init (void) /* RTMaker: IO_Start */
{
    int c /* channel */;
    if (MIO_M408_Config (Slot_PWMIn) < 0) return RT_Failure;

    for (c=0; c<NCHANNELS; c++) {
        MIO_M408_SetTimeout (Slot_PWMIn, c, 2.000 /* Timeout = 2 s */);
        MIO_M408_SetThreshold(Slot_PWMIn, c,
            3.5, /* upper threshold */
            0.5, /* lower threshold */
            0, /* measure from rising edge to rising edge */
            0 /* Do not detect increasing frequencies */ );
    }
}

void IO_In (void)
{
    int TO, c /* channel */;

    int TO = MIO_M408_GetTimeout(Slot_PWMIn);
    for (c=0; c<NCHANNELS; c++) {
        IO.Frequency[c] = MIO_M408_GetFrequency(Slot_PWMIn, c);
        IO.DutyCycle[c] = MIO_M408_GetDutyCycle(Slot_PWMIn, c);
        IO.TO[c] = (TO >> c) & 0x01;
        IO.PeriodCounter[c] = MIO_M408_GetPeriodCounter(Slot_PWMIn, c);
    }
}
}

```

SENT Receiver Example

```

#define Slot_M408 1
#define SENT_Channel 7

struct {
    unsigned char RxData[6];
} IO;

int IO_Init (void) /* RTMaker: IO_Start */
{
    // [...]
    MIO_M408_Config (Slot_M408);
    MIO_M408_SetThreshold (Slot_M408, SENT_Channel, 1.5, 1.0, 0, 0 /*EnDecreasingFreq*/);
    MIO_M408_SENTConfig (Slot_M408, SENT_Channel, 1 /*Enable*/, 6 /* expected nibbles*/,
        1 /* Message frame contains pause pulse */
        0.00000003 /* 3us */, 0 /* Not EnhancedMsgFormat */);
    MIO_M408_SENTRstFIFO (Slot_M408, SENT_Channel);
    // [...]
}

void IO_In (void)
{
    unsigned char rxdata[6], crc, comstat;
    int i;

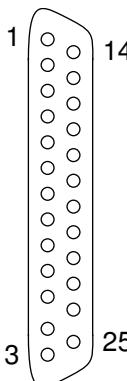
    // [...]
}

```

```
if ((n = MIO_M408_SENTGetData (Slot_M408, SENT_Channel, &rxdata, &crc, &comstat)) >= 0) {  
    for (i=0;i<n; i++) {  
        IO.RxData[i] = rxdata[i];  
        /* Do something with IO.RxData */  
    }  
    // [...]  
}
```

10.4.23 M409: Power Supply Control, 2 Units

Connector Assignment



Pin	Unit	Signal Description	Pin	Unit	Signal Description
1	0	D _{in} 0	14	0	D _{out} 0
2	0	D _{in} 1	15	0	D _{out} 1
3	0	D _{in} 2	16	0	DGND
4	0	D _{in} 3	17	0	AGND
5	0	A _{in} 0	18	0	A _{out} 0
6	0	A _{in} 1	19	0	A _{out} 1
7	1	D _{in} 0	20	1	D _{out} 0
8	1	D _{in} 1	21	1	D _{out} 1
9	1	D _{in} 2	22	1	DGND
10	1	D _{in} 3	23	1	AGND
11	1	A _{in} 0	24	1	A _{out} 0
12	1	A _{in} 1	25	1	A _{out} 1
13	1	A _{in} 2			

Annotation:

- D/A_{in/out} = Digital/Analog Input/Output
- DGND = Digital Ground
- AGND = Analog Ground

Function Overview

Initialization and Configuration

- `MIO_M409_Config()`
- `MIO_M409_DinSetThreshold()`
- `MIO_M409_DinSetThreshold()`
- `MIO_M409_DinGetThreshold()`

Power Supply Functions

- `struct tMIO_M409_PowerSupply`
- `MIO_M409_PowerSupplyInit()`
- `MIO_M409_PowerSupplySet()`
- `MIO_M409_PowerSupplyGet()`

Example

- Usage of the Power Supply functionality

Multi Purpose I/O Functions

- `MIO_M409_AinGet()`
- `MIO_M409_AoutSet()`
- `MIO_M409_AoutGet()`
- `MIO_M409_DinGet()`

- [MIO_M409_DoutSet \(\)](#)
- [MIO_M409_DoutGet \(\)](#)

MIO_M409_Config()

```
int MIO_M409_Config (int Slot);
```

Description

By calling the function [MIO_M409_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M409 is specified. This function must be called one time, before any attempted access to the module.

Initial state of the module:

- All Analog Outputs are set to 0.0V
- All Digital Outputs are set to '0' (switch open)
- The Threshold Voltage of all Digital Inputs is set to 2.0V
- Both units are enabled (see [MIO_M409_DinSetThreshold\(\)](#))
- If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

MIO_M409_DinSetThreshold()

```
void MIO_M409_DinSetThreshold (int Slot, int Unit, int Channel, float fValue);
```

Description

Sets the threshold voltage for the digital Input <channel> to the float value <fValue>.

Parameters

- fValue

This is the voltage which will be compared to the digital input voltage. If the input voltage is bigger than the configured threshold voltage, the corresponding input goes to logic high '1'

Range: 0.0V ... 5.0V (Default 2.0V)

MIO_M409_DinGetThreshold()

```
float MIO_M409_DinGetThreshold (int Slot, int Unit, int Channel);
```

Description

Reads back the configured Din threshold voltage value (see [MIO_M409_DinSetThreshold\(\)](#)).

Power Supply Functions

The M409 module is primarily intended to be used as a power supply control hardware, which can handle up to 2 power supplies. The default power supply functions have a similar pin assignment like the PowerUTA (USB to Analog) to assure some backwards compatibility.



Hint:

If you want to use the M409 module as a Multi Purpose I/O (MPIO) device you can skip the following struct and power supply functions. They only refer to the usage of the M409 as a power supply control unit and aren't needed for the MPIO mode. See section '[Multi Purpose I/O Functions](#)' for details.

In Software one Power Supply is represented in a C-Language struct:

struct tMIO_M409_PowerSupply

```
typedef struct {
    /* Analog Outputs */
    float Vsel;           // 0.0 .. 80.0V -> Aout(0)
    float Csel;           // 0.0 .. 100.0A -> Aout(1)
    /* Analog Inputs */
    float Vmon;           // 0.0 .. 80.0V -> Ain(0)
    float Cmon;           // 0.0 .. 100.0A -> Ain(1)
    /* Digital Outputs */
    char Remote;          // 1->Remote Control enabled => Output pulled to GND
                          // 0->Remote Control disabled => Output=High-Z
    char Standby;         // 1->PowerSupply disabled => Output=High-Z
                          // 0->PowerSupply enabled => Output pulled to GND
    /* Digital Inputs */
    char Error;           // 0->Low=OK 1->Open=Error -> Din(0)
    char CC_CV;           // 0->Low=CV 1->Open=CC     -> Din(1)
    char OVP;             // 0->Low=OK 1->Open=Error -> Din(2)
    char OT;              // 0->Low=OK 1->Open=Error -> Din(3)
} tMIO_M409_PowerSupply;
```

All relevant data is hold in this struct. The user has to define a variable of the type tMIO_M409_PowerSupply and initialize it properly.

Default Power Supply Connection

PSC Signal	Description	Note	PS-9080-100 Pin
D _{in} 0	Error	L=0V, H=open to DGND	10
D _{in} 1	CC_CV	L=0V, H=open to DGND	7
D _{in} 2	OVP	L=0V, H=open to DGND	8
D _{in} 3	OT	L=0V, H=open to DGND	9
A _{in} 0	Vmon	0..10V	5
A _{in} 1	Cmon	0..10V	6
D _{out} 0	Remote	open drain output	22
D _{out} 1	Standby	open drain output	23
A _{out} 0	Vsel	0..10V	3
A _{out} 1	Csel	0..10V	2
DGND	digital ground		20, 21
AGND	analog ground		14, 15, 16

Warning

The EA PS-9080-100 is the example device. Other devices have different pinning!

MIO_M409_PowerSupplyInit()

```
void MIO_M409_PowerSupplyInit (int Slot, int Unit,
                               float VmaxControl,
                               float VmaxPS,
                               float CmaxPS);
```

Description

This function initializes the internal PowerSupply configuration. It has to be called if the user wants the Power Supply Control functionality.

Parameters

- VmaxControl
This parameter defines the maximum control voltage. It depends on the Power Supply which control voltage is needed. Common values are 5.0V or 10.0V
- VmaxPS
The maximum output voltage of the Power Supply (e.g. 80.0V)
- CmaxPS
The maximum output current of the Power Supply (e.g. 100A)

MIO_M409_PowerSupplySet()

```
void MIO_M409_PowerSupplySet (int Slot, int Unit, tMIO_M409_PowerSupply *ps);
```

Description

This function updates the output values of the M409 module by reading the appropriate values from the tMIO_M409_PowerSupply struct given by the user.

Parameter

- tMIO_M409_PowerSupply *ps
Pointer to a Power Supply struct

MIO_M409_PowerSupplyGet()

```
void MIO_M409_PowerSupplyGet (int Slot, int Unit, tMIO_M409_PowerSupply *ps);
```

Description

This function reads the input values of the M409 module and saves them to the struct specified by the *ps pointer.

Parameter

- tMIO_M409_PowerSupply *ps
Pointer to a Power Supply struct

Example

Usage of the Power Supply functionality

```
/* Define Power Supply variable */
tMIO_M409_PowerSupply ps;

int
IO_Init (void) {
    [...]
    /* Default Power Supply values */
    ps.Vsel = 0.0; // output 0V
    ps.Csel = 0.0; // output 0V
    ps.Remote = 0; // output active High -> 'Z'
    ps.Standby = 1; // output active High -> 'Z'
    [...]
    /* Module Initialisation */
    MIO_M409_Config(Slot_M409);
    MIO_M409_Enable(Slot_M409, Unit_M409, 1);
    /* Configure Power Supply */
    MIO_M409_PowerSupplyInit (Slot_M409, Unit_M409,
        10.0,           // Maximum Control Voltage
        80.0,           // Maximum Output Voltage of PowerSupply
        100.0          // Maximum Output Current of PowerSupply
    );
    [...]
    return 0;
}

void
IO_In (unsigned CycleNo) {
    [...]
    /* Read the Inputs */
    MIO_M409_PowerSupplyGet(Slot_M409, Unit_M409, &ps);
    /* Do something with the Inputs */
    Log("Vmon: %f\n", ps.Vmon);
```

```
Log("Cmon: %f\n", ps.Cmon);
Log("Error: %c\n", ps.Error);
Log("CC_CV: %c\n", ps.CC_CV);
Log("OVP: %c\n", ps.OVP);
Log("OT: %c\n", ps.OT);
[...]
}

void
IO_Out (unsigned CycleNo) {
[...]
/* Set Output Values */
ps.Vsel = 16.0; // 16.0V
ps.Csel = 10.0; // 10.0A
ps.Remote = 1; // enable external control
ps.Standby = 0; // enable Power Supply output
/* Write the Outputs */
MIO_M409_PowerSupplySet(Slot_M409, Unit_M409, &ps);
[...]
}
```

Multi Purpose I/O Functions

MIO_M409_AinGet()

```
float MIO_M409_AinGet (int Slot, int Unit, int Channel);
```

Description

This function reads the voltage from an analog input pin. The user has to take care that the analog input voltage does not exceed the range of 0.0V ... 10.0V.

Parameters

- Channel
Analog Input Channel 0 ... 1 [2 - only Unit 1]
Range: 0.0V ... 10.0V

MIO_M409_AoutSet()

```
void MIO_M409_AoutSet (int Slot, int Unit, int Channel, float fValue);
```

Description

This function writes a given voltage to an analog output pin. The user has to take care that the connected device is able to handle configured voltage. Please refer the Hardware Manual for maximum ratings concerning the available current per channel.

Parameters

- Channel
Analog Output Channel 0 ... 1
- fValue
Range: 0.0V ... 10.0V

MIO_M409_AoutGet()

```
float MIO_M409_AoutGet (int Slot, int Unit, int Channel);
```

Description

Reads back the configured Aout voltage value (see [MIO_M409_AoutSet\(\)](#)).

MIO_M409_DinGet()

```
unsigned char MIO_M409_DinGet (int Slot, int Unit);
```

Description

This function reads the digital inputs to a unsigned char variable. The four least significant bits represent the corresponding digital inputs:

- Bit 0: Din(0) / Error
- Bit 1: Din(1) / CC_CV
- Bit 2: Din(2) / OVP
- Bit 3: Din(3) / OT

Example

The following example demonstrates how to use [MIO_M409_DinGet\(\)](#). The received value uDin is extracted into the char array Din[4]

```
/* Din */
char Din[4];
unsigned char uDin = MIO_M409_DinGet (Slot_M409, Unit_M409);
Din[0] = (char) ((uDin >> 0) & 0x1);
Din[1] = (char) ((uDin >> 1) & 0x1);
Din[2] = (char) ((uDin >> 2) & 0x1);
Din[3] = (char) ((uDin >> 3) & 0x1);
```

MIO_M409_DoutSet()

```
void MIO_M409_DoutSet (int Slot, int Unit, unsigned char Dout);
```

Description

This function writes the digital outputs of the M409. It requires an unsigned char variable as parameter input. The two least significant bits represent the corresponding digital outputs:

- Bit 0: Dout(0) / Remote
- Bit 1: Dout(1) / Standby

Example

The following example demonstrates how to use [MIO_M409_DoutSet\(\)](#). The char array Dout[2] is converted into an unsigned char Dout and send to the module.

```
/* Dout */
char Dout[2];
Dout[0] = 0; // open switch
Dout[1] = 1; // close switch
```

```
unsigned char uDout = ((Dout[1] & 0x1) << 1) + // Dout(1)
                     ((Dout[0] & 0x1) << 0); // Dout(0)
MIO_M409_DoutSet (Slot_M409, Unit_M409, uDout);
```

MIO_M409_DoutGet()

```
unsigned char MIO_M409_DoutGet (int Slot, int Unit);
```

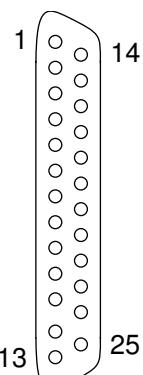
Description

This function reads back the configured Dout to a unsigned char.

See [MIO_M409_DoutSet \(\)](#).

10.4.24 M410: Quadruple CAN/CANFD Interface

Connector assignment



Pin	Signal	Pin	Signal
1	CAN0_low	14	CAN0_high
2	CAN0_GND	15	
3		16	
4	CAN1_low	17	CAN1_high
5	CAN1_GND	18	
6		19	
7	CAN2_low	20	CAN2_high
8	CAN2_GND	21	
9		22	
10	CAN3_low	23	CAN3_high
11	CAN3_GND	24	
12		25	
13			

Function Overview

Since CarMaker 5.0 the M410 is supported. The older M51 can be replaced one by one by the M410. Therefore a compatibility mode is implemented in the M51-functions since CarMaker 5.0. Thus no need for changes in the source-code is needed. Simply a recompilation with linking against the 5.0 libraries must be done.

The M410 does not support the Mailbox-Mode of the M51.

Of course this compatibility mode does support new functions of the M410 like CANFD and the switchable termination resistor.

At the end of this chapter an example for the usage of the M410 is available.

Structs and Typedefs

- `enum tMatchCode`
- `struct CANFD_Msg`
- `MIO_M410_Config()`

Initialization and Configuration

- `MIO_M410_Config()`
- `MIO_M410_ConfigureCAN()`
- `MIO_M410_ConfigureCANFD()`
- `MIO_M410_SetCommParam()`
- `MIO_M410_EnableIds()`

Sequential Send/Receive Mode

- `MIO_M410_Send()`
- `MIO_M410_Recv()`
- `MIO_M410_ResetFIFO()`

Hook Functions

- [MIO_M410_HookFct \(\)](#)
- [MIO_M410_RxHookSet \(\)](#)
- [MIO_M410_TxHookSet \(\)](#)

Diagnostics

- [MIO_M410_GetTxErrCnt \(\)](#)
- [MIO_M410_GetRxErrCnt \(\)](#)

enum tMatchCode

```
typedef enum {
    M410_MatchStandardID = 0x1,
    M410_MatchExtendedID = 0x2,
    M410_MatchStandardCAN = 0x4,
    M410_MatchCANFD = 0x8
} tMatchCode;
```

Description

This enum is needed for configuring the filter by [MIO_M410_EnableIds\(\)](#).

Values

- M410_MatchStandardID - Receive frames with 11-bit-IDs
- M410_MatchExtendedID - Receive frames with 29-bit-IDs
- M410_MatchStandardCAN - Receive standard CAN frames
- M410_MatchCANFD - Receive CANFD frames

It is possible to or the values, e.g. to receive standard-IDs and extended IDs, only with standard CAN frames.

struct CANFD_Msg

```
typedef struct {
    unsigned int MsgId : 29;      // Message-ID
    unsigned int IDE : 1;         // 0: Standard Frame ID, 1: Extended Frame ID
    unsigned int RTR : 1;         // Request to Transmit (Standard-CAN only)
    unsigned int DLC : 4;         // Data Length Code
    unsigned int FDF : 1;         // = EDL-Bit: 0: Standard CAN, 1: CANFD
    unsigned int BRS : 1;         // 0: No Baudrate Switch, 1: Baudrate Switch with CANFD
    unsigned int ESI : 1;         // Error State Indicator with CANFD (Rx only)
    unsigned int reserved : 26;
    unsigned char Data[64];       // Data: Standard-CAN uses only Data[7..0]
} CANFD_Msg;
```

Description

This struct defines the structure of a CAN/CANFD frame with the M410. The members all are described by the CANFD specification.

MIO_M410_Config()

```
int MIO_M410_Config (int Slot);
```

Description

Description

By calling the function [MIO_M410_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M410 is specified. This function must be called one time, before any attempted access to the module.

Initial state of the module:

- The CAN controller does not do anything.
- The Rx-FIFO as well as the Tx-FIFO is cleared.
- All Rx-filters are cleared, thus no messages are accepted.
- Hook-Functions ([MIO_M410_RxHookSet\(\)](#) and [MIO_M410_TxHookSet\(\)](#)) are deleted.

MIO_M410_GetReceiverStatus()

```
unsigned int MIO_M410_GetReceiverStatus(int Slot, int Channel);
```

Description

This function returns the number of messages in Rx-FIFO. The Rx-FIFO is arranged in bytes but not in messages and has a size of 4kbytes. Therefore the number of possible messages in the FIFO depends on the number of data bytes of each message.

Example

A standard CAN message with 8 data bytes needs 24 bytes in the FIFO which means about 170 messages can be stored before the buffer is full. With a data rate of 500kBaud the Rx-FIFO is full after about 34milliseconds.

A CANFD message with 64 data bytes needs 80 bytes in the FIFO which means about 51 messages can be stored before the buffer is full. With a data rate of 1MBaud for the arbitration and 8MBaud for the data the Rx-FIFO is full after about 5 milliseconds.

MIO_M410_GetTransmitterStatus()

```
unsigned int MIO_M410_GetTransmitterStatus(int Slot, int Channel);
```

Description

This function returns the number of messages in Tx-FIFO. The transmitter FIFO is arranged in bytes but not in messages and has a size of 4kbytes. Therefore the number of possible messages in the fifo depends on the number of data bytes of each message.

Example

For example a standard CAN message with 8 data bytes needs 24 bytes in the FIFO which means about 170 messages can be stored before the buffer is full. With a data rate of 500kBaud the Tx-data for the next 34 milliseconds can be stored in the FIFO.

A CANFD message with 64 data bytes needs 80 bytes in the FIFO which means about 51 messages can be stored before the buffer is full. With a data rate of 1MBaud for the arbitration and 8MBaud for the data the Tx-data for the next 5 milliseconds can be stored in the FIFO.

MIO_M410_ConfigureCAN()

```
int MIO_M410_ConfigureCAN(int Slot, int Channel, int Baudrate, int TerminatorEnable);
```

Description

`MIO_M410_ConfigureCAN()` is the function that usually should be used to configure standard CAN messages. It internally sets most of the needed communication parameters, so that the user only needs to configure the baudrate and if the termination should be used. This function internally sets the following values:

- Sample point = 75%
- SyncJumpWidth = 2
- Listen Only Mode = 0

For further details also see [MIO_M410_SetCommParam\(\)](#).

Parameters

- The controller is enabled
- Baudrate: This is the CAN baudrate
- TerminatorEnable: This parameter must be set to "1" if the onboard termination should be used, otherwise it must be set to "0"

Return Value

`MIO_OK` if everythin was fine, `MIO_ERROR` if an error happened.

MIO_M410_ConfigureCANFD()

```
int MIO_M410_ConfigureCANFD(int Slot, int Channel,
                             int BaudrateSlow, int SamplePointSlow /* [%] */,
                             int BaudrateFast, int SamplePointFast /* [%] */,
                             int TerminatorEnable, int IsISO);
```

Description

`MIO_M410_ConfigureCANFD()` is the function that usually should be used to configure CANFD messages. It internally sets most of the needed communication parameters, so that the user only needs to configure network specific parameters.

This function internally sets the following values:

- The controller is enabled
- SyncJumpWidth = Time segment 2, but a maximum of 2 (changed in CM 5.0.3)
- Listen Only Mode = 0

For an example of using this function see [Listing 10.9](#) at the end of this chapter.

For further details also see [MIO_M410_SetCommParam\(\)](#).

Parameters

- BaudrateSlow: This is the baudrate of the arbitration field of the CANFD message.
- SamplePointSlow: Because of the baudrate switching point the sample point within the arbitration field must be the same for all knodes of the network. The sample point must bei given in "%".
- BaudrateFast: This is the baudrate of the data field of the CANFD message.
- SamplePointFast: Because of the baudrate switching point the sample point within the data field must be the same for all knodes of the network. The sample point must bei given in "%".

- TerminatorEnable: This parameter must be set to "1" if the onboard termination should be used, otherwise it must be set to "0"
- IsISO specifies if the (older) Non-ISO specification is used, or if the (newer) ISO CAN-FD standard is used. Because of the different behaviour of those both protocols both CANFD versions are not compatible. Thus this value must be the same for all knodes in the network.
"0" uses the Non-ISO standard, "1" uses the ISO standard.

Return Value

MIO_OK if everything was fine, MIO_ERROR if an error happened.

MIO_M410_SetCommParam()

```
int MIO_M410_SetCommParam (int Slot, int Channel, int Enable, int IsISO, int IsCANFD,
                           int IsQuiet, int TerminatorEnable,
                           int BaudrateSlow, int BaudrateFast,
                           int SyncJumpWidth, int SamplePointSlow, int SamplePointFast);
```

Description

This function is the expert configuration function which allows you to specify any communication parameter yourself.

Parameters

- Enable
"0" disables the controller, "1" enables it. When disabled also the transceiver is disabled and the knode is not recognizable on the bus.
- IsISO specifies if the (older) Bosch specification is used, or if the (newer) ISO CANFD standard is used. Because of the different behaviour of those both protocols this value must be the same for all knodes in the network.
"0" uses the Non-ISO standard, "1" uses the ISO standard.
- IsCANFD
"0" if the controller works as standard CAN controller, "1" if the controller works as a CANFD controller.
- IsQuiet
If set to "1" the M410 works as a bus monitor and does not send acknowledges on the bus.
- TerminatorEnable: This parameter must be set to "1" if the onboard termination should be used, otherwise it must be set to "0"
- BaudrateSlow: This is the baudrate of the arbitration field of the CANFD message.
- BaudrateFast: This is the baudrate of the data field of the CANFD message.
- SyncJumpWidth: With CAN and CANFD the time segments which define the CAN bitrate can automatically be lengthened or shortened to synchronize the CAN knodes if the bitrate of the transmitter and the receiver differ. The SyncJumpWidth determines by how many time quantas the time segments may be maximally extended or shortened. The number time segments as well as the prescaler depend on the bitrate and the

sample point SamplePoint. Therefore the parameter SJW with the M410 does not specify an absolute value but scales the time segment 2. If [MIO_M410_ConfigureCAN\(\)](#) or [MIO_M410_ConfigureCANFD\(\)](#) are used, SJW is set to 2.

Table 10.3:

SJW Parameter	Corresponding internal SJW
<=1	$SJW = 0,25 * TimeSegment2$
2	$SJW = 0,5 * TimeSegment2$
3	$SJW = 0,75 * TimeSegment2$
>=4	$SJW = 1 * TimeSegment2$

- SamplePointSlow: Because of the baudrate switching point the sample point within the arbitration field must be the same for all nodes of the network. The sample point must be given in "%".
- SamplePointFast: Because of the baudrate switching point the sample point within the data field must be the same for all nodes of the network. The sample point must be given in "%".

Return Value

`MIO_OK` if everything was fine, `MIO_ERROR` if an error happened.

MIO_M410_EnableIds()

```
int MIO_M410_EnableIds (int Slot, int Channel, tMatchCode MatchCode, int StartId, int nIDs);
```

Description

By default no messages are received because the acceptance filters are cleared. In order to receive any messages their ID have to be enabled by calling the [MIO_M410_EnableIds\(\)](#). This function enables the receipt of the specified CAN-IDs, beginning with StartId and ending with StartId + (nIDs-1) on all the CAN/CANFD channels.

Up to 256 filters can be configured for each channel. Each call of the function configures at least one new filter, starting with index 0. Depending on the ID range probably more than one filter is configured with one function call.

The module can also be configured to be transparent which means all messages are received.

Additionally it is possible to specify if only 11-bit-identifier messages or 29-bit-identifier-messages and CAN or CANFD messages should be received. Also any combination of those are possible.

Disabling already enabled IDs is not possible.

Also see the example at the end of this chapter. (see [Listing 10.9](#))

Return Value

`MIO_OK` if everything was fine, `MIO_ERROR` if an error happened.

MIO_M410_Send()

```
int MIO_M410_Send (int Slot, int Channel, CANFD_Msg *Msg);
```

Description

The specified message is written to the Tx-FIFO of the M410. If the transmit buffer is full, the message is ignored and the error code -1 is returned.

Also see the example at the end of this chapter. (see [Listing 10.9](#))

Parameters

Msg is the address of a data structure `struct CANFD_Msg`.

Return Value

`MIO_OK` if everythin was fine, `MIO_ERROR` if the buffer was full or if a hook function - if configured - returns an error; also see [MIO_M410_TxHookSet\(\)](#).

MIO_M410_Recv()

```
int MIO_M410_Recv (int Slot, int Channel, CANFD_Msg *Msg);
```

Description

Reads the next message from the receive buffer of the M410. By default no messages are received because the acceptance filters are cleared. In order to reveive any message their ID have to be enabled by calling the function [MIO_M410_EnableIds\(\)](#).

Parameters

Msg is the address of a data structure of type `struct CANFD_Msg`.

Return Value

[MIO_M410_Recv\(\)](#) returns the number of messages remaining in the Rx-FIFO. The return value is -1 (`MIO_ERROR`) if the data returned in Msg is not valid, e.g. if the buffer was empty when calling [MIO_M410_Recv\(\)](#) or if a hook function - if configured - returns an error; also see [MIO_M410_RxHookSet\(\)](#).

MIO_M410_ResetFIFO()

```
void MIO_M410_ResetFIFO (int Slot, int Channel);
```

Description

Calling this function cleares the Rx-FIFO as well as the Tx-FIFO.

MIO_M410_GetRxErrCnt()

```
int MIO_M410_GetRxErrCnt (int Slot, int Channel);
```

Description

This function returns the receive error counter. The receive error counter uses the CAN rules.

MIO_M410_GetTxErrCnt()

```
int MIO_M410_GetTxErrCnt (int Slot, int Channel);
```

Description

This function returns the transmit error counter. The transmit error counter uses the CAN rules.

MIO_M410_HookFct()

```
typedef int (*MIO_M410_HookFct) (CANFD_Msg *);
```

Description

It is possible to hook a function into the [MIO_M410_Send\(\)](#) and [MIO_M410_Recv\(\)](#) function. Those hook functions can be used to trace the traffic on the CAN/CANFD bus. Additionally the data which is received or is to be transmitted could be modified, e.g. for failure simulations. If defined, the RxHook function is called immediately after reading the data from the Rx-FIFO of the M410 within [MIO_M410_Recv\(\)](#). The TxHook function is called directly before the data to be transmitted is written to the Tx-FIFO within [MIO_M410_Send\(\)](#).

Parameters

Msg is the address of a data structure of type [struct CANFD_Msg](#). In case of the TxHook function Msg contains the data which is to be send by the M410. Each value of the message struct can be modified which causes the M410 to send the modified message instead of the original data. In case of the RxHook function Msg contains the data which is read from the module. This also could be modified.

Return Values

All hook functions should return 0. Otherwise in case of the TxHook function read / modified data is not send and [MIO_M410_Send\(\)](#) returns with an error code. In case of the RxHook function the data read from the module is marked as invalid because [MIO_M410_Recv\(\)](#) returns with an error code.

MIO_M410_RxHookSet()

```
MIO_M410_HookFct MIO_M410_RxHookSet (int Slot, int Ch, MIO_M410_HookFct Func);
```

Description

Hooks a function into the [MIO_M410_Recv\(\)](#). The function is hooked immediately after the reading of the Rx-FIFO. Therefore it is possible to read or modify received data.

MIO_M410_TxHookSet()

```
typedef int (*MIO_M410_HookFct) (CANFD_Msg *);  
MIO_M410_HookFct MIO_M410_TxHookSet (int Slot, int Ch, MIO_M410_HookFct Func);
```

Description

Hooks a function into the `MIO_M410_Send()`. The function is hooked immediately before the writing the Tx-FIFO. Therefore it is possible to read or modify transmitted data.

Example

Listing 10.9: M410 Usage

```
1: int TxHook(CANFD_Msg *TxMsg)
2: {
3:     /* Example: Modify DLC for Message with ID 0x43 */
4:     if (TxMsg->MsgId==0x43) TxMsg->DLC--;
5:     return 0;
6: }
1:
1:
1: int IO_Init(void)
2: {
3:     MIO_Init (NULL);
4:     MIO_M410_Config (CAN_SLOT);
5:     // Ch=0: StandardCAN, 500kBaud, Terminator closed
6:     MIO_M410_ConfigureCAN(CAN_SLOT, 0 /*ch*/, 500000 /*Baud*/, 1 /*Terminator*/);
7:     // Ch=1: CANFD, 500kBaud/4MBaud, SP_Slow=SPFast=75%, ISO, Terminator closed,
8:     // TransmitterDelay: auto
9:     MIO_M410_ConfigureCANFD(CAN_SLOT, 1, 500000 /*Arb-Baud*/, 75 /*SP-Arb*/,
10:                           4000000 /*Data-Baud*/, 75 /*SP-Data*/,
11:                           1 /*Terminator*/, 1 /*ISO-Standard*/)
12:     // Channel 0: Receive all messages with StandardID, but not with extended ID
13:     MIO_M410_EnableIds (CAN_SLOT, 0, M410_MatchStandardID | M410_MatchStandardCAN,
14:                          -1 /*StartID*/, -1 /*nIDs*/);
15:     // Channel 1: Receive IDs 64-100, StandardID and ExtendedID, CAMFD
16:     MIO_M410_EnableIds (CAN_SLOT, 1, M410_MatchStandardID | M410_MatchExtendedID |
17:                          M410_MatchCANFD,
18:                          64 /*StartID*/, 37 /*nIDs*/);
19:     // Only if needed: Configure a Hook-function for Tx of channel 0
20:     MIO_M410_TxHookSet (CAN_SLOT, 0, TxHook);
21: }
22:
23:
24: int IO_In(unsigned CycleNo)
25: {
26:     CANFD_Msg RxMsg;
27:     int ch;
28:     /* Receive data from all channels as long as there is some available */
29:     for (ch=0; ch<2; ch++) {
30:         while (MIO_M410_Recv(CAN_SLOT, ch, &RxMsg)>=0) {
31:             /* Do something with the received data */
32:         }
33:     }
34: }
35:
36:
37: int IO_Out(unsigned CycleNo)
38: {
39:     CANFD_Msg TxMsg;
40:     /* Set TxData from IO.XXX */
41:     TxMsg.MsgId = 0x43;
42:     TxMsg.DLC = 2;
43:     TxMsg.FDF = 0; /* Standard CAN */
44:     TxMsg.IDE = 1; /* Extended Frame */
45:     TxMsg.RTR = 0; /* No RTR */
46:     TxMsg.Data[0] = IO.XXX1;
47:     TxMsg.Data[1] = IO.XXX2;
48:     MIO_M410_Send(CAN_SLOT, 0, &TxMsg);
49: }
50:
```

10.4.25 M412: Parksensor Simulation Board

Connector assignment

Pin	Signal Description	Pin	Signal Description
1	Transducer0 (+)	14	Transducer0 (-)
2	Transducer1 (+)	15	Transducer1 (-)
3	Transducer2 (+)	16	Transducer2 (-)
4	Transducer3 (+)	17	Transducer3 (-)
5	Transducer4 (+)	18	Transducer4 (-)
6	Transducer5 (+)	19	Transducer5 (-)
7	BinaryI/O Ch0	20	Binary GND
8	BinaryI/O Ch1	21	Binary GND
9	BinaryI/O Ch2	22	Binary GND
10	BinaryI/O Ch3	23	Binary GND
11	BinaryI/O Ch4	24	Binary GND
12	BinaryI/O Ch5	25	Binary GND
13	n.c.		

Function Overview

Structs and Typedefs

- enum tMIO_M412_TriggerAction

Initialization and Configuration

- MIO_M412_Config()
- MIO_M412_AnalogChannelConfigure()
- MIO_M412_BinaryChannelConfigure()
- MIO_M412_BinarySetTriggerTiming()

Runtime Functions

- MIO_M412_USSetDistance(), MIO_M412_BinarySetDistance()
- MIO_M412_BinarySetTriggerTiming()
- MIO_M412_BinaryConfAction()

enum tMIO_M412_TriggerAction

```
enum {
    MIO_M412_TrigDisabled = 0x0,
    MIO_M412_USReceiveSend = 0x1,
    MIO_M412_USReceive = 0x2,
    MIO_M412_TxSerial = 0x3
} tMIO_M412_TriggerAction;
```

Description

This enumerator is used to specify the kind of action if a valid message is received from the ECU in binary mode. The values are used with the function [MIO_M412_BinaryConfAction\(\)](#) and have the following meaning:

- MIO_M412_TrigDisabled
Ignore the configured message and do nothing. This is the default after power-on.
- MIO_M412_USReceiveSend
If the configured message is detected, the module emulates first the transmission of an ultrasonic pulse on the control line between ECU and Sensor and thereafter eventually emulates the reception of this pulse on the control line.
- MIO_M412_USReceive
If the configured message is detected, the module emulates the reception of this pulse on the control line.
- MIO_M412_TxSerial
If the configured message is detected, the module sends a serial message on the control line between ECU and Sensor, e.g. to emulate the status of the sensor.

MIO_M412_Config()

```
int MIO_M412_Config (int Slot)
```

Description

By calling the function [MIO_M412_Config\(\)](#), the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M412, is specified. This function must be called one time, before any attempted access to the module.

Initial state of the module:

- All channels are disable
- the waveform is initialized to an ideal sinus with an amplitude of +/-4.096V

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

MIO_M412_AnalogChannelConfigure()

```
void MIO_M412_AnalogChannelConfigure (int Slot, int Channel, int Enable,
                                         float OutputGain /* 0 .. 1.0; refers to data in RAM */,
                                         int ConfigOrigin /* 0-frequency/nPeriods from CM, 1-measured frequency, nPeriods*/,
                                         float Frequency, /* 25kHz .. 100kHz */,
                                         int nPeriods);
```

Description

This function configures a channel in the ultrasonic mode of the M412. It configures how the echoes are generated and how they look like.

Parameters

- **Enable**
"1" enables this channel
- **OutputGain**
This is the amplification factor for the output waveform values. During generation of the output waveform the voltage values are taken from the waveform buffer and are multiplied by the gain value. The valid range is from 0.0 to 1.0.
- **ConfigOrigin**
"1" means the number of periods and frequency of the output signal are taken from the measured input signal. Due to the amplitude of the captured signal these values can differ slightly from the real ultrasonic signal.
"0" means these values are taken from [MIO_M412_AnalogChannelConfigure\(\)](#).
- **Frequency**
This is the frequency of the echoes on this channel. The valid range is 25kHz to 100kHz. Frequency specifies how fast the waveform buffer is replayed. Please note that this value is only considered, if ConfigOrigin is set to "0".
- **nPeriods**
This is the number of periods of the echoes on this channel. The valid range is 0 to 255. nPeriods specifies how often the waveform buffer is replayed. Please note that this value is only considered, if ConfigOrigin is set to "0".

MIO_M412_BinaryChannelConfigure()

```
void MIO_M412_BinaryChannelConfigure (int Slot, int Channel, int Enable,
                                      int USTxDelay /* Delay between Tx-Command from ECU and Answer from Sensor [usec] */,
                                      int USTxDuration /* Duration of Tx-Signal on Control Line [usec] */,
                                      int USRxDuration /* Duration of Rx-Signal on Control Line [usec] */,
                                      int SerialTxDelay      /* [usec] */,
                                      int SerialBitTime       /* [usec] */)
```

Description

This function configures a channel in the binary mode of the M412. The binary mode takes influence on the control line between the ECU and the ultrasonic sensor. In this mode the M412 captures commands from the ECU on the control line and answers on the control lines as the real sensor would do.

Parameters

- **Enable**
"1" enables this channel
- **USTxDelay**
This specifies the delay between the time a "Do Transmit" command is received via the control line to the time the sensor answers with "Transmission active" on the control line in microseconds.
- **USTxDuration**
USTxDuration is the duration of the "Transmission active" signal on the control line in microseconds.

- **USRxDuration**
USRxDuration is the duration, the sensor emulation pulls the control line low, when it detects an ultrasonic pulse in microseconds.
- **SerialTxDelay**
This specifies the delay between the time a "Send Serial Message" command is received via the control line to the time the sensor emulation answers with the serial message on the control line in microseconds.
- **SerialBitTime**
The SerialBitTime specifies how many microseconds one bit needs when sending a serial message from the M412 to the ECU. The number of bits thereby is limited to 24.

MIO_M412_SetWaveform()

```
void MIO_M412_SetWaveform (int Slot, int Channel, float Buf[M412_WF_BUFSIZE])
```

Description

There is a waveform buffer of 2048x12bit for each ultrasonic output channel. This buffer contains one period of the ultrasonic signal. After power on, this buffer is initialized with an ideal sinus period with the maximum possible amplitude -4.096V to +4.095V.

By this function it is possible to overwrite the buffer with your own waveform.

Parameters

- **Buf[]**
Buf contains the waveform of one period of the output signal. The buffer consists of 2048 value where each value must be in the range from -4.096V to +4.095V.

MIO_M412_USSetDistance(), MIO_M412_BinarySetDistance()

```
void MIO_M412_USSetDistance (int Slot, int Channel,
                           int SensorMask[M412_NECHOS],
                           int tDistance[M412_NECHOS] /* usec */)
void MIO_M412_BinarySetDistance (int Slot, int Channel,
                                 int SensorMask[M412_NECHOS],
                                 int tDistance[M412_NECHOS] /* usec */);
```

Description

[MIO_M412_USSetDistance\(\)](#), [MIO_M412_BinarySetDistance\(\)](#) configures whether, when and on which sensors echoes occur after the pulse from a vehicle sensor was detected. Thereby the configuration is done from the viewpoint of the vehicle sensors.

Up to 16 echoes can be configured for each sensor. The distances specify the time after that the echoes occur, the sensor masks specify which vehicle sensors should here the echo.

MIO_M412_USSetDistance() only work in ultrasonic (analog) mode,
MIO_M412_BinarySetDistance() only works in binary mode.

Parameters

- Distance
Distance specifies the delay between the detection of a signal on this sensor to the generation of the echo in microseconds. If a distance value is 0 no further echoes are generated. The distance values and the corresponding mask values are sorted by [MIO_M412_USSetDistance\(\)](#), [MIO_M412_BinarySetDistance\(\)](#) internally in the way that the lowest value (!=0) comes first and the highest value comes last. Additionally echoes with the same distance value but different mask values are combined.
- Mask
Mask is a bitmask which defines which vehicle sensors should here receive the echoes at the corresponding time.
Bit 0 corresponds to vehicle sensor 0
Bit 1 corresponds to vehicle sensor 1 and so on.

Example

The following example does the following:

If vehicle sensor 0 (=channel 0) sends an ultrasonic pulse then

- after 1000us an echo is generated for vehicle sensor 0
- after 1500us an echo is generated for vehicle sensor 1
- after 2500us echos are generated for the vehicle sensors 0 and sensor 2

```
unsigned int Distance[16] = {1000,1500,2500};
unsigned int Mask[16] = {0x01, 0x02, 0x5};
int channel = 0;
MIO_M412_USSetDistance (Slot_M412, channel, Mask, Distance);
```

MIO_M412_BinarySetTriggerTiming()

```
void MIO_M412_BinarySetTriggerTiming (int Slot, int Channel,
                                      int TimingID,      // 0..7
                                      int MinTime        /*usec*/,
                                      int MaxTime        /*usec*/);
```

Description

With [MIO_M412_BinarySetTriggerTiming\(\)](#) the timing for the command from the ECU to the sensor on the control line is specified. The timing is specified by the duration of a high state or a low state of the control line.

This means that, starting from the idle state (long time High-Z) of the control line after each change in level, the duration of how long the level remains the same is measured. If these durations are in ranges specified by [MIO_M412_BinarySetTriggerTiming\(\)](#) their timing-IDs are set for the corresponding nibbles in an internal Rx command register. For each newly received valid TimingID the content of the command register is shifted left by 4 and the new value is inserted as nibble 0 (bit 3..0). The action which corresponds to this command must be defined by [MIO_M412_BinaryConfAction\(\)](#).

There must not be any ambiguity within the timing at any time!

For additional information also see the example at the end of this chapter.

Parameters

- TimingID
Each configured Timing must have its own unique TimingID.
Up to 8 (0..7) timings can be specified. The TimingIDs combined over the whole cycle define the TriggerCommand for [MIO_M412_BinaryConfAction\(\)](#)
- MinTime, MaxTime
This is the minimum / maximum duration that the level keeps the same to be recognized as valid for the corresponding TimingID and to cause "TimingID to be added to the internal command word.

MIO_M412_BinaryConfAction()

```
void MIO_M412_BinaryConfAction (int Slot, int Channel,
                                int ActionID, // 0..7
                                unsigned int TriggerCommand,
                                tMIO_M412_TriggerAction Action,
                                int Tx_NBits,unsigned int Tx_Data);
```

Description

With [MIO_M412_BinaryConfAction\(\)](#) the actions are specified which follow to a received command. The received command consists of a list of consecutive TimingIDs. If this list matches the configured TriggerCommand the corresponding action is executed. There must not be any ambiguity within the TriggerCommand at any time during reception!

Up to 8 actions can be specified at all.

For additional information also see the example at the end of this chapter.

Parameters

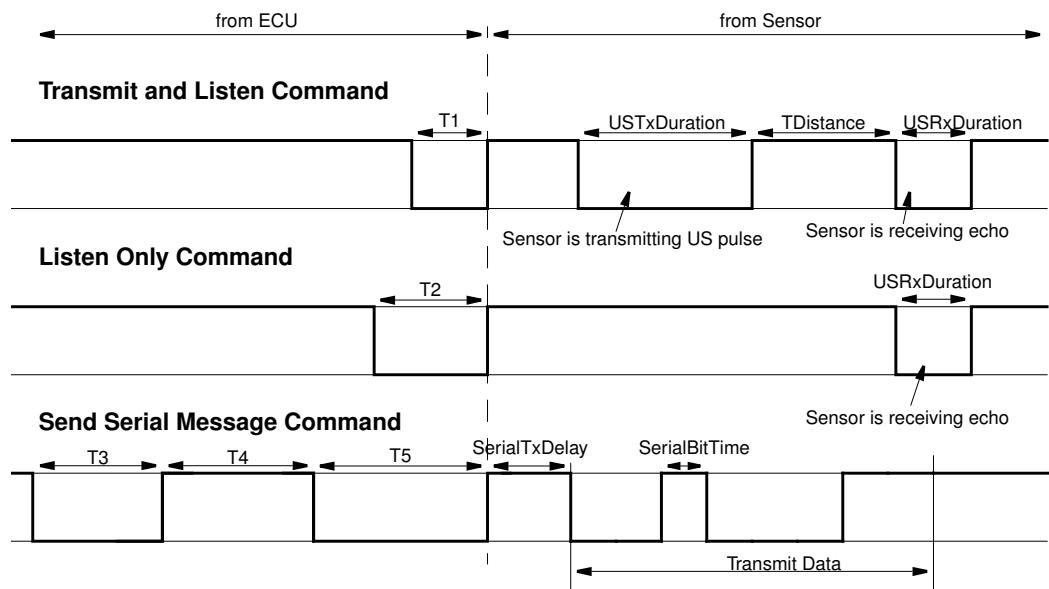
- ActionID
Each configured action must have its own ActionID.
- TriggerCommand
If TriggerCommand matches the command received from the ECU, the corresponding Action is triggered.
- Action
Action is from type [enum tMIO_M412_TriggerAction](#). The action specified, what is to be done if the corresponding TriggerCommand is received.
- Tx_NBits
If the corresponding action is *MIO_M412_TxSerial* this specifies the number of bits to be send to the ECU.
- Tx_Data
If the corresponding action is *MIO_M412_TxSerial* this specifies the message which is send to the ECU.

Binary Mode Example

- If the control line for channel 0 is pulled low by the ECU for T1 (50us < T1 <100us) the sensor emulation should emulate an ultrasonic transmission.
1000usecs after transmission the Sensors 0 should emulate the reception of an echo.

- If the control line for channel 0 is pulled low by the ECU for T2 ($100\text{us} < T2 < 150\text{us}$) the sensor emulation should emulate the reception of echoes only but not transmit an ultra-sonic pulse.
- If the control line for channel 0 is pulled low by the ECU for T3 ($150\text{us} < T3 < 200\text{us}$), thereafter be high for T4 ($200\text{us} < T4 < 250\text{us}$), followed by a low level T5 ($250\text{us} < T5 < 300\text{us}$) the sensor should send 2 bytes with a value 0x23 and a bittime of 10us to the ECU.

Timing Diagram



Listing:

```

int IO_Init(void)
{
    [...]
    MIO_M412_Config(SLOT_M412);
    MIO_M412_BinaryChannelConfigure (SLOT_M412, 0 /* Channel */, 1 /* Enable */,
        5 /* USTxDelay in usec */,
        100 /* USTxDuration in usec */,
        50 /* USRxDuration in usec */,
        5 /* SerialTxDelay in usec */,
        10 /* SerialBitTime in usec */);

    /* Configure Timings T1, T2 and T3 */
    MIO_M412_BinarySetTriggerTiming (SLOT_M412, 0 /* Channel */,
        1 /* TimingID */, 50 /* MinTime in usec */, 100 /* MaxTime in usec */);
    MIO_M412_BinarySetTriggerTiming (SLOT_M412, 0 /* Channel */,
        2 /* TimingID */, 101 /* MinTime in usec */, 150 /* MaxTime in usec */);
    MIO_M412_BinarySetTriggerTiming (SLOT_M412, 0 /* Channel */,
        3 /* TimingID */, 151 /* MinTime in usec */, 200 /* MaxTime in usec */);
    MIO_M412_BinarySetTriggerTiming (SLOT_M412, 0 /* Channel */,
        4 /* TimingID */, 201 /* MinTime in usec */, 250 /* MaxTime in usec */);
    MIO_M412_BinarySetTriggerTiming (SLOT_M412, 0 /* Channel */,
        5 /* TimingID */, 251 /* MinTime in usec */, 300 /* MaxTime in usec */);

    /* Timing-ID 1 (command = 0x01): Transmission US-Pulse and wait for echoes */
    MIO_M412_BinaryConfAction (SLOT_M412, /* Channel */,
        0 /* ActionID */, 0x01 /* TriggerCommand */, MIO_M412_USReceiveSend /* Action */,
        0 /* Tx_NBits: No serial message */, 0 /* Tx_Data: No serial message */);
    /* Timing-ID 2 (=> command=0x02): Enable the sensor for receiving echoes. */
}

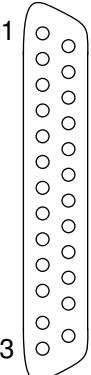
```

```
MIO_M412_BinaryConfAction (SLOT_M412, /* Channel */,
    1 /* ActionID */, 0x02 /* TriggerCommand */, MIO_M412_USReceive /* Action */,
    0 /* Tx_NBits: No serial message */, 0 /* Tx_Data: No serial message */);
/* Timing-ID 3 followed by T4 and T5 (=> command=0x345): Send Serial Message 0x23. */
MIO_M412_BinaryConfAction (SLOT_M412, /* Channel */,
    2 /* ActionID */, 0x0345 /* TriggerCommand */, MIO_M412_TxSerial /* Action */,
    8 /* Tx_NBits: 8 bits*/, 0x23 /* Tx_Data: message = 0x23 */);
[...]
}

void IO_Out (void)
{
[...]
    IO.tDistance[0] = 1000; /* usec, defined in IO.h */
    IO.SensorMask[0] = 0x01; /* Echo 0 occurs on Sensor 0, defined in IO.h */
    MIO_M412_BinarySetDistance (SLOT_M412, 0 /* Channel */, IO.SensorMask, IO.tDistance );
[...]
}
```

10.4.26 M413: 5x 4:1 Multiplexer / De-Multiplexer

Connector assignment



Pin	Signal Description	Pin	Signal Description
1	MUX0:IO0	14	MUX1:IO0
2	MUX0:IO1	15	MUX1:IO1
3	MUX0:IO2	16	MUX1:IO2
4	MUX0:IO3	17	MUX1:IO3
5	MUX0:COM	18	MUX1:COM
6	MUX2:COM	19	MUX3:COM
7	MUX2:IO0	20	MUX3:IO0
8	MUX2:IO1	21	MUX3:IO1
9	MUX2:IO2	22	MUX3:IO2
10	MUX2:IO3	23	MUX3:IO3
11	MUX4:IO0	24	MUX4:IO1
12	MUX4:IO2	25	MUX4:IO3
13	MUX4:COM		

Function Overview

Initialization and Configuration

- `MIO_M413_Config()`

Runtime Functions

- `MIO_M413_Set ()`
- `MIO_M413_Get ()`

MIO_M413_Config()

```
int MIO_M413_Config (int Slot);
```

Description

By calling the function `MIO_M413_Config()`, the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M413, is specified. This function must be called one time, before any attempted access to the module.

Initial state of the module:

- All channels are switched off

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

MIO_M413_Set ()

```
void MIO_M413_Set (int Slot, int Mux[5]);
```

Description

This function configures all the multiplexers on the board.

Parameters

- int Mux[5]

Mux is an array of 5 integer values which configure the multiplexers.

Index 0 configures the MUX0, index 1 configures MUX1, and so on.

The following integer values are valid:

- 1: Disconnect COM from all IOs
 - 0: Connect COM to IO0
 - 1: Connect COM to IO1
 - 2: Connect COM to IO2
 - 3: Connect COM to IO3
- Other values must not be used!

Example

```
void Disconnect_All_Muxes(int Slot)
{
    int Mux[5] = {-1, -1, -1, -1, -1};
    MIO_M413_Set (Slot, Mux);
}

void Connect_Mux0_IO3(int Slot)
{
    int Mux[5] = {3, -1, -1, -1, -1};
    MIO_M413_Set (Slot, Mux);
}
```

MIO_M413_Get ()

```
int MIO_M413_Get (int Slot);
```

Description

`MIO_M413_Get ()` returns the current state of the Multiplexers as integer value.

Return Value

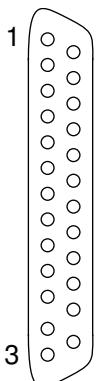
- Bit 2..0 corresponds to MUX0
- Bit 5..3 corresponds to MUX1
- Bit 8..6 corresponds to MUX2
- Bit 11..9 corresponds to MUX3
- Bit 14..12 corresponds to MUX4

The bits have the following meaning:

- “111” - COM disconnected from all IOs
- “000” - COM connected to IO0
- “001” - COM connected to IO1
- “010” - COM connected to IO2
- “011” - COM connected to IO3

10.4.27 M440: Resistor Cascade, 6 Channels

Connector assignment



Pin	Signal Description	Pin	Signal Description
1	Channel 0 - Connector A	14	Channel 0 - Connector B
2	not connected	15	not connected
3	Channel 1 - Connector A	16	Channel 1 - Connector B
4	not connected	17	not connected
5	Channel 2 - Connector A	18	Channel 2 - Connector B
6	not connected	19	not connected
7	Channel 3 - Connector A	20	Channel 3 - Connector B
8	not connected	21	not connected
9	Channel 4 - Connector A	22	Channel 4 - Connector B
10	not connected	23	not connected
11	Channel 5 - Connector A	24	Channel 5 - Connector B
12	not connected	25	not connected
13	not connected		

Function Overview

Initialization and Configuration

- [MIO_M440_Config\(\)](#)

Runtime Functions

- [MIO_M440_SetResistor\(\)](#)
- [MIO_M440_GetCurrent\(\)](#)
- [MIO_M440_GetOverloadState\(\)](#)
- [MIO_M440_ResetOverload\(\)](#)

MIO_M440_Config()

```
int MIO_M440_Config (int Slot);
```

Description

By calling the function `MIO_M440_Config()`, the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M440, is specified. This function must be called one time, before any attempted access to the module.

Initial state of the module:

- All channels are shut off
- The resistor value of each channel is set to infinite

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.

MIO_M440_SetResistor()

```
void MIO_M440_SetResistor (int Slot, int Channel, double Resistor);
```

Description

Set the resistor of the specified channel to the desired value.

Parameters

“Resistor” is the resistor value of the channel. The following table shows, which value results in which effect.

Parameter “Resistor”	Channel Resistor
<0	infinite
0 .. 16	16Ohms
16 .. 8000000	as specified
>8000000	8MOhm

MIO_M440_GetCurrent()

```
double MIO_M440_GetCurrent (int Slot, int Channel);
```

Description

Returns the current of the specified channel average over the last 400ms. If the overload protection of the channel is active, the return value is the current which caused the overload protection to get active, also averaged over 400ms.

The unit of the return value is Ampere.

MIO_M440_GetOverloadState()

```
unsigned short MIO_M440_GetOverloadState (int Slot)
```

Description

This function returns the overload protection state of all channels.

Return Value

- Bit 0 corresponds to channel 0
- Bit 1 corresponds to channel 1
- Bit 2 corresponds to channel 2
- and so on

The return value can be used to reset the overload state by [MIO_M440_ResetOverload\(\)](#).

MIO_M440_ResetOverload()

```
void MIO_M440_ResetOverload (int Slot, unsigned short ResetMask)
```

Description

The overload protection is not reset automatically but must be reset by this function. Calling this function re-establishes the connection and the resistor value of each channel.

To be safe from destroying resistor channels due to fastly repeated overloads, the connection is not re-established earlier than 3 seconds after the activation even if [MIO_M440_ResetOverload\(\)](#) is called immediately after the activation.

Parameters

- Bit 0 corresponds to channel 0
- Bit 1 corresponds to channel 1
- Bit 2 corresponds to channel 2
- and so on

The return value of [MIO_M440_GetOverloadState\(\)](#) can be used as parameter for this function.

10.4.28 M441: PWM Out and SENT Transmitter, 12 / 16 Channels

Connector assignment

Since Revision 2.x

Female SubD-25 ConnectorFemale SubD-25 ConnectorFemale SubD-25 Connector

Pin	Signal Description	Pin	Signal Description
1	Output Channel 0	14	Output Channel 1
2	Output Channel 2	15	Output Channel 3
3	Output Channel 4	16	Output Channel 5
4	Output Channel 6	17	Output Channel 7
5	Output Channel 8 ^a	18	Output Channel 9 ^a
6	Output Channel 10 ^a	19	Output Channel 11 ^a
7	Output Channel 12 ^a	20	Output Channel 13 ^a
8	Output Channel 14 ^a	21	Output Channel 15 ^a
9	n.c.	22	n.c.
10	V_{REF0}	23	V_{REF0}
11	V_{REF1}	24	V_{REF1}
12	GND	25	GND
13	GND		

a. Channel does not support SENT.

Revision 1.x

Female SubD-25 ConnectorFemale SubD-25 Connector

Pin	Signal Description	Pin	Signal Description
1	Output Channel 0	14	Output Channel 1
2	Output Channel 2	15	Output Channel 3
3	n.c.	16	n.c.
4	Output Channel 4	17	Output Channel 5
5	Output Channel 6	18	Output Channel 7
6	n.c.	19	n.c.
7	Output Channel 8 ^a	20	Output Channel 9 ^a
8	Output Channel 10 ^a	21	Output Channel 11 ^a
9	n.c.	22	n.c.
10	V_{REF0}	23	V_{REF0}
11	V_{REF1}	24	V_{REF1}
12	GND	25	GND
13	GND		

a. Channel does not support SENT.

Output Stage

For some better understanding the following diagram shows the simplified output stage of an M441 channel.

M441 Output Channel

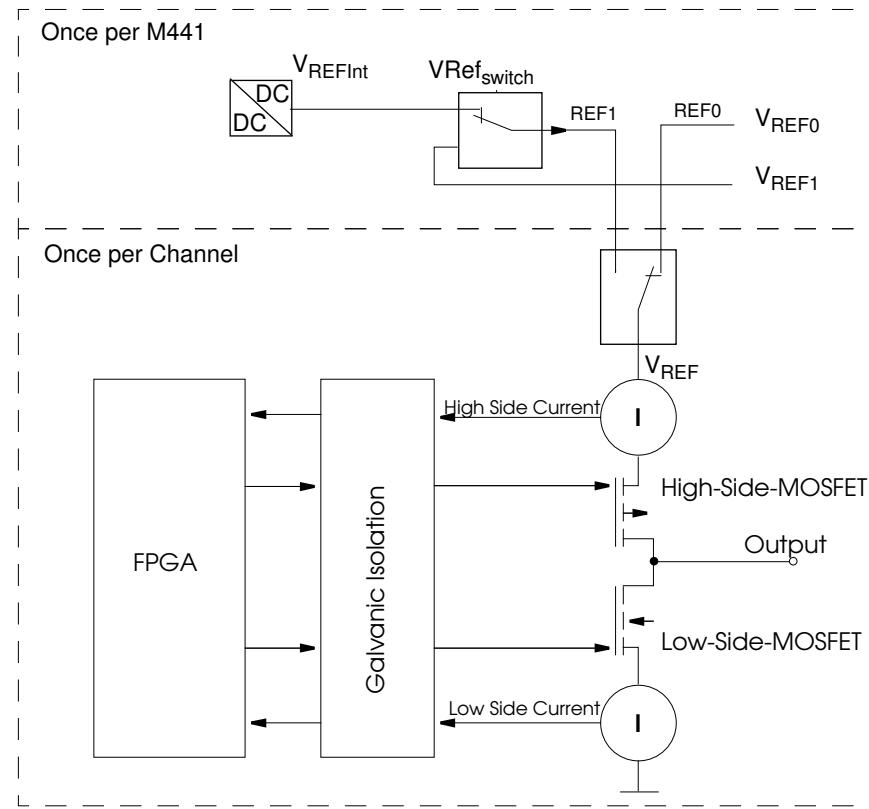


Figure 10.6: Output Driver Stage

Function Overview

Initialization and Configuration

- [MIO_M441_Config\(\)](#)
- [MIO_M441_EnableChannels\(\)](#)
- [MIO_M441_SetMode\(\)](#)
- [MIO_M441_EnableInternalReference\(\)](#)
- [MIO_M441_SENTConfig\(\)](#)

Runtime Functions

- [MIO_M441_SetFrequency\(\)](#)
- [MIO_M441_SetPeriod\(\)](#)
- [MIO_M441_SetDutyCycle\(\)](#)
- [MIO_M441_SetPulseWidth\(\)](#)
- [MIO_M441_SetFrequency_MP\(\)](#)
- [MIO_M441_SetPeriod_MP\(\)](#)
- [MIO_M441_SetDutyCycle_MP\(\)](#)
- [MIO_M441_SetPulseWidth_MP\(\)](#)
- [MIO_M441_QuadEncSetPhase\(\)](#)

- [MIO_M441_SetBinary\(\)](#)
- [MIO_M441_GetCurrent\(\)](#)
- [MIO_M441_GetOverCurrentMask\(\)](#)
- [MIO_M441_ResetOverCurrentState\(\)](#)
- [MIO_M441_SENTSetData\(\)](#)
- [MIO_M441_SENTTriggerMsgNo\(\)](#)
- [MIO_M441_SENTSetIDData\(\)](#)

Examples

- All Channels as PWM out with Open Collector
- All channels as PWM out with internal reference
- Quadrature Encoder with REF0 for 2 channels

enum eMIO_M441_Mode

```
typedef enum {
    M441_BinaryOut = 0x000000,
    M441_FrequencyOut = 0x004000,
    M441_SENTOut = 0x008000,
    M441_EnablePullGND = 0x010000,
    M441_UseRef0 = 0x020000,
    M441_UseRef1 = 0x040000,
    M441_EnableTrigMM = 0x100000,
    M441_EnableTrigA = 0x200000,
    M441_EnableTrigB = 0x400000
} eMIO_M441_Mode;
```

Description

This enumerator is used for [MIO_M441_SetMode\(\)](#). It consists of 3 parts:

- Signal Output Mode
 - M441_BinaryOut
The channel is used as a binary output channel.
 - M441_FrequencyOut
The channel is used as a frequency (PWM) generator, a multi-pulse generator or a quadrature encoder.
 - M441_SENTOut
The channel generates SENT signals (SAE J2716).
- If none of the three values is specified, the channel works as a binary output channel.
- Signal Output Level
 - M441_EnablePullGND
If this value is given, the channel is pulled against GND if the signal output level should be low. If it is void, the low-side MOSFET is always opened which results in an open-emitter output. In this case the channel should be pulled low externally.
 - M441_UseRef0
This determines that the channel is pulled against REF0 if the signal output level should be high.
 - M441_UseRef1
This determines that the channel is pulled against REF1 if the signal output level should be high.
If neither M441_UseRef0 nor M441_UseRef1 are specified the channel works as an open-collector output. In this case the channel should be pulled high externally.
- If none of the three values is specified, both MOSFETs of the channel are open. Thus the channel is in high-Z mode.
- Signal Triggering Mode
 - These values define whether a pulse or a sequence of pulses is generated continuously or only when a certain event happened.
 - M441_EnableTrigMM
A pulse or sequence of pulses is generated if a write access to the pulse number 0 is done via the function [MIO_M441_SetFrequency\(\)](#) or [MIO_M441_SetFrequency_MP\(\)](#).
 - M441_EnableTrigA
A pulse or sequence of pulses is generated if a rising edge happens on the TrigA input of the M-Module connector of the module.

- M441_EnableTrigB

A pulse or sequence of pulses is generated if a rising edge happens on the TrigB input of the M-Module connector of the module.

If none of the three values are specified, the signal is generated continuously without waiting for any trigger signal. This only works in frequency mode!

MIO_M441_Config()

```
int MIO_M441_Config (int Slot);
```

Description

By calling the function [MIO_M441_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M441 is specified. This function must be called one time, before any attempted access to the module.

Initial state

- All channels are disabled
- All outputs are high-impedance
- The output mode is set to “binary”
- The Frequency is set to 0Hz
- The duty cycle is set to 0

MIO_M441_EnableChannels()

```
void MIO_M441_EnableChannels(int Slot, int EnableMask);
```

Description

This function enables the output channels.

Parameters

- EnableMask

This parameter is a bitmask for enabling the output channels.

bit0 enables channel 0
bit1 enables channel 1
bit2 enables channel 2
bit3 enables channel 3
bit4 enables channel 4
bit5 enables channel 5
bit6 enables channel 6
bit7 enables channel 7
bit8 enables channel 8
bit9 enables channel 9
bit10 enables channel 10
bit11 enables channel 11
bit12 enables channel 12
bit13 enables channel 13
bit14 enables channel 14
bit15 enables channel 15

MIO_M441_SetMode()

```
void MIO_M441_SetMode (int Slot, int Channel, int nPulses, int DataChangeMode,  
                      int PulseWidthMode, eMIO_M441_Mode OutputMode, int IsQuadEnc);
```

Description

`MIO_M441_SetMode()` configures the output mode of each channel of the M441.

Parameters

- **nPulses**
This is the number of pulses that a multi-pulse signal consists of. Set this value to 1 if a common PWM-Signal is used.
- **DataChangeMode**
0 - Frequency is immediately (within the current period) adapted when it changes multi-pulse sequence continues with the current pulse number
1 - Frequency changes are adapted after the current period, the multi-pulse sequence continues with the current pulse number
3 - Changes are done immediately, the multi-pulse sequence is restarted with pulse 0
4 - Changes are done after the last pulse of a multi-pulse signal
5 - The multi-pulse sequence is sent once after a trigger event occurs
- **Pulse Width Mode**
0 - Pulse width is depending on period (duty cycle mode)
1 - reserved
2 - fix low duration within period
3 - fix high duration within period
- **OutputMode**
This value is composed of three parts which are all part of the [enum eMIO_M441_Mode](#).
See [enum eMIO_M441_Mode](#) for further details. See the [Output Stage](#) to better understand how the output stage works.
- **IsQuadEnc (since M441 firmware 1.3)**
If this value is set to 1 the specified channel and the next one are combined to a quadrature encoded signal with a to be configured phase shift.
IsQuadEnc can only be set to "1" for even numbered channels.

MIO_M441_EnableInternalReference()

```
void MIO_M441_EnableInternalReference(int Slot, int Enable);
```

Description

This function replaces the obsolete `MIO_M441_SetRef2IntExt()`.

With this function you specify if the external REF1 input or the internal reference voltage is used if a channel is specified to use REF1. See the [Output Stage](#) to better understand how the output stage works.

Parameters

- 0 - Use the external reference input REF1
- 1 - Use the internal 5V-reference-voltage

MIO_M441_SetFrequency()

```
void MIO_M441_SetFrequency (int Slot, int Channel, int PulseNo, double Frequency);
```

Description

[MIO_M441_SetFrequency \(\)](#) configures the frequency of the specified pulse and the specified channel. If a common PWM-signal is to be generated, `PulseNo` must always be set to “0”.

`PulseNo` must be in the range of 0 .. 63.

`Frequency` must be in the range 0Hz .. 1MHz.

If you have to configure multi-pulse signals the function [MIO_M441_SetFrequency_MP \(\)](#) should be preferred.

In the quadrature encoder mode the frequency must be set for the first channel (Track A). The maximum frequency in this mode is 520kHz. Negative frequency values means “turning left” positive frequency values means “turning right”. The phase shift can be freely configured in a range from - 2PI .. +2PI via [MIO_M441_QuadEncSetPhase \(\)](#).

MIO_M441_SetPeriod()

```
void MIO_M441_SetPeriod (int Slot, int Channel, int PulseNo, double Period);
```

Description

This function is almost the same as [MIO_M441_SetFrequency \(\)](#). The difference is, that at this point you can specify the period time of the output signal in seconds.

`PulseNo` must be in the range of 0 .. 63.

`Period` must be in the range 1us .. 40s.

If you have to configure multi-pulse signals, the function [MIO_M441_SetPeriod_MP \(\)](#) should be preferred.

MIO_M441_SetDutyCycle()

```
void MIO_M441_SetDutyCycle (int Slot, int Channel, int PulseNo, double DutyCycle);
```

Description

Configures the relationship between high duration and low duration within one period for the specified `PulseNo`. If `DutyCycle` is 0.0, the output signal is all low, if `DutyCycle` is 1.0, the output signal is all high.

`PulseNo` must be within the range of 0 .. 63.

The value of `DutyCycle` must be within the range of 0.0 ... 1.0.

This function only works correctly if the “Pulse Width-Mode” is set to “Duty Cycle Mode” in [MIO_M441_SetMode \(\)](#).

If you have to configure multi-pulse signals, the function [MIO_M441_SetDutyCycle_MP \(\)](#) should be preferred.

MIO_M441_SetPulseWidth()

```
void MIO_M441_SetPulseWidth (int Slot, int Channel, int PulseNo, double PulseWidth);
```

Description

With this function the value for the fixed low duration or the fixed high duration (depending on the configured pulse width mode within [MIO_M441_SetMode \(\)](#)) is set.

This duration stays the same, even if the frequency changes. This means that if the period is lower than the specified pulse width, the signal keeps high or low all the time (depending on the configured pulse width mode).

PulseNo must be in the range of 0 .. 63.

The unit of PulseWidth is [sec] and must be within the range 0sec to 40sec.

This function only works correctly if the “Pulse Width Mode” is set to “Fix low duration” or “Fix high duration” in [MIO_M441_SetMode \(\)](#).

If you have to configure multi-pulse signals, the function [MIO_M441_SetPulseWidth_MP \(\)](#) should be preferred.

MIO_M441_SetFrequency_MP()

```
void MIO_M441_SetFrequency_MP (int Slot, int Channel, int nPulses, double* Frequency);
```

Description

[MIO_M441_SetFrequency_MP \(\)](#) configures the frequency for the specified number of pulses at once.

nPulses must be in the range of 0 .. 63.

Frequency is an array of nPulses double values which must be in the range 0Hz .. 1MHz.

If you have to configure a common PWM signal, the function [MIO_M441_SetFrequency \(\)](#) should be used.

Also see [MIO_M441_SetMode \(\)](#) for more details about the point in time, where new data is adjusted.

MIO_M441_SetPeriod_MP()

```
void MIO_M441_SetPeriod_MP (int Slot, int Channel, int nPulses, double* Period);
```

Description

[MIO_M441_SetPeriod_MP \(\)](#) configures the period for the specified number of pulses at once.

nPulses must be in the range of 0 .. 63.

Period is an array of <nPulses> double values which must be in the range 1us .. 40s.

If you have to configure a common PWM signal, the function [MIO_M441_SetPeriod \(\)](#) should be used.

Also see [MIO_M441_SetMode \(\)](#) for more details about the point in time where new data is adjusted.

MIO_M441_SetDutyCycle_MP()

```
void MIO_M441_SetDutyCycle_MP (int Slot, int Channel, int nPulses, double* DutyCycle);
```

Description

[MIO_M441_SetDutyCycle_MP\(\)](#) configures the duty cycle for the specified number of pulses at once.

nPulses must be in the range of 0 .. 63.

DutyCycle is a array of <nPulses> double values which must be in the range of 0.0 ... 1.0.

If you have to configure a common PWM signal the function [MIO_M441_SetPeriod\(\)](#) should be used.

Also see [MIO_M441_SetMode\(\)](#) for more details about the point in time where new data is adjusted.

MIO_M441_SetPulseWidth_MP()

```
void MIO_M441_SetPulseWidth_MP (int Slot, int Channel, int nPulses, double* PulseWidthBuffer);
```

Description

[MIO_M441_SetPulseWidth_MP\(\)](#) configures the pulse width for the specified number of pulses at once.

nPulses must be in the range of 0 .. 63.

PulseWidth is a array of <nPulses> double values which must be in the range of 1 us .. 40s.

If you have to configure a common PWM signal, the function [MIO_M441_SetPeriod\(\)](#) should be used.

Also see [MIO_M441_SetMode\(\)](#) for more details about the point in time where new data is adjusted.

MIO_M441_QuadEncSetPhase()

```
void MIO_M441_QuadEncSetPhase (int Slot, int Channel, int PulseNo, double PhaseAngle);
```

Description

This function configures the phase shift of the second channel of a quadrature encoded against the first channel. The unit is rad. By default the phase shift is set to 0rad.

The value range is between -2Pi and +2Pi.

- If the value is >0.0rad and the frequency is positive track B legs after track A.
- If the value is <0.0rad and the frequency is positive track B leads before track A.
- With negative frequency values it is vice versa.

The phase shift can be freely configured in a range from - 2PI .. +2PI.

In the example below the phase shift is set to +1/2PI (+90deg).

MIO_M441_SetBinary()

```
void MIO_M441_SetBinary (int Slot, unsigned short BinaryMask);
```

Description

Set the outputs of all channels which are configured as binary outputs. Bit 0 of `Value` corresponds to channel 0, bit 1 corresponds to output port 1 and so on.

MIO_M441_GetCurrent()

```
double MIO_M441_GetCurrent (int Slot, int Channel);
```

Description

This function returns the current through the high-side MOSFET in Amps.

MIO_M441_GetOverCurrentMask()

```
unsigned int MIO_M441_GetOverCurrentMask (int Slot);
```

Description

This function returns the overload protection state of all channels in the lower 16bits of the return value. A “1” means, that the channel is shut-off due to over-current. Additionally the position of an over current is returned in the upper 16bits of the return value. There a “1” means that the high side MOSFET was overload while a “0” means the low side MOSFET was overload.

Return Value

Bit 15..0: Overload Mask.

“1” means overload protection state is active.

- Bit 0 corresponds to channel 0
- Bit 1 corresponds to channel 1
- Bit 2 corresponds to channel 2
- ...
- Bit 11 corresponds to channel 11
- Bits 12..15 are reserved for future use.

Bit 31..16: Overload Position

“0” means the low side MOSFET was overload, “1” means the high side MOSFET was overload

- Bit 16 corresponds to channel 0
- Bit 17 corresponds to channel 1
- Bit 18 corresponds to channel 2
- ...
- Bit 27 corresponds to channel 11
- Bits 28..31 are reserved for future use.

The return value anded by `0xffff` can be used to reset the overload state by `MIO_M441_ResetOverCurrentState()`.

MIO_M441_ResetOverCurrentState()

```
void MIO_M441_ResetOverCurrentState (int Slot, unsigned short ChannelMask);
```

Description

The overload protection is not reset automatically but must be reset by this function. Calling this function re-enables all the channels whose corresponding bit is set to “1”.

To be safe from destroying channels due to fastly repeated overloads the channel is not re-enabled earlier than 3 seconds after the activation even if `MIO_M441_ResetOverCurrentState()` is called immediately after the activation.

Parameters

- Bit 0 corresponds to channel 0
- Bit 1 corresponds to channel 1
- Bit 2 corresponds to channel 2
- and so on

The return value of `MIO_M441_GetOverCurrentMask()` can be used as parameter for this function.

MIO_M441_SENTConfig()

```
void MIO_M441_SENTConfig(int Slot, int Channel, float TickTime, int nTicksPauseFLength,
                         int nMessages, int SerialIDDataMode, int SendContinous, int SendUserCRC,
                         int FixFrameLengthMode, int SignalGenerationMode);
```

Description

This function configures the SENT transmitter. SENT is an asynchronous PWM based uni-directional sensor interface. The SENT message frame has the following format:

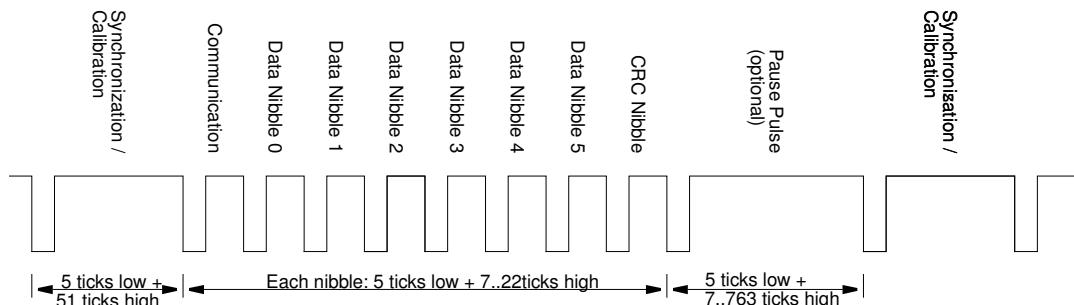


Figure 10.7: SENT Signal Structure

For further information see the HardwareManual and the SAE J2716.

Parameters

- Channel
SENT is only available for channels 0..7!



- TickTime
As you can see above, with SENT everything is defined in time units, so called “ticks”. Defined by the SENT specification the tick time is 3us .. 90us. However, with the M441 values must be in a range from 500ns .. 40.0s. The unit is seconds.
- nTicksPauseFLength
This is the length of the optional pause pulse. If it is 0, no pause pulse is created. otherwise the pause pulse has the specified length in “ticks”. The length of the pause pulse can be set in a range from 1 to 65635 ticks.
Since revision 2.2 of the M441 it is possible to configure a fixed frame length instead of the length of the pause pulse. In this case the frame length in ticks must be configured here. The length of the pause pulse then is adopted dynamically to the length of the data nibbles and the communication and status nibble. To use the fix frame length mode set the bit "FixFrameLengthMode" to "1".
-
- nMessages
The module has a message queue with a size of 16 messages. Therefore, up to 16 different messages can be sent in a sequence, continuously repeated.
If the number of message is set to N (0 < N < 17), the messages #0 to #(N-1) are sent subsequently.
- SerialIDDataMode
As described in the SAE J2716 the communication nibble is used to transmit serial data, e.g. ID data of the sensor. There are 3 different valid data modes. Additionally it is configurable that no serial data is sent.

Table 10.4:

Serial Data Mode	Number of Bits in the Serial Data		
	Message ID	Serial CRC	Serial Data
0x0	No serial data is sent, status data is taken from “Status” from MIO_M441_SENTSetData()		
0x1	4	4	8
0x2	8	6	12
0x3	4	6	16

- SendContinous
As defined in the SAE J2716 messages should be send continuously without a break between two messages (except the optional pause pulse). However, the module is configurable to send messages software triggered. If this value is set to “1”, messages are sent continuously. If set to “0”, messages are sent with every write access to the module via [MIO_M441_SENTTriggerMsgNo\(\)](#). This mode is not part of the SENT specification but is only for testing purposes.
- SendUserCRC
If this parameter is set to “0”, the M441 generates the CRC nibble itself and always correctly. If the value is set to “1”, the user must specify the content of the CRC nibble. This is recommended for fail-safe testing only.
This configuration only affects the CRC nibble of the SENT data. The CRC of the serial data within the communication nibble must always be calculated by the user.
- FixFrameLengthMode
To use the fix frame length mode set this bit to "1". If the length of the pause pulse should be fix, set this bit to "0".
- SignalGenerationMode
If this bit is set to "0" a nibble starts with 5 ticks low followed by 7+n ticks high.
If it is set to "1" a frame starts with 7+n ticks low followed by 5 ticks high.

MIO_M441_SENTSetData()

```
void MIO_M441_SENTSetData(int Slot, int Channel, int MsgNo, unsigned char CRC,  
                           unsigned int nNibbles, unsigned char *SENTData, char Status);
```

Description

The M441 has a message queue for 16 messages. This means that up to 16 different messages can be send one after another and then starting with the first one again if the “Continous Mode” is selected.

Parameters

- **MsgNo**
MsgNo must be in the range from 0 .. 15 and specifies the message number whose data is set.
- **CRC**
This is the user calculated CRC value of the message. This value is ignored if `SendUserCRC` within [MIO_M441_SENTConfig\(\)](#) is set to “0”
- **nNibbles**
is the number of nibbles that are sent with the specified message.
- **SENTData**
Here you configure each data nibble. SENTData is an array of at least <nNibbles> character values. SENTData[0] corresponds to data nibble 0, SENTData[1] corresponds to data nibble 1, and so on.
- **Status**
Since M441 revision 2.2 it is possible to set the value of the communication and status nibble of the sent frame manually. If this value is set to “-1” it is ignored. otherwise the communication and status nibble is set to the configured value.
Depending on `SerialIDDataMode` from the [MIO_M441_SENTConfig\(\)](#) register the status and communication nibble is set completely or partially by the value from Status.
If `SerialIDDataMode` is set to “0” all the 4 bits of the status and communication nibble are set, otherwise only the bits 0 and 1 are set.

MIO_M441_SENTSetIDData()

```
void MIO_M441_SENTSetIDData(int Slot, int Channel, int SerialID, int SerialCRC,  
                            int SerialData);
```

Description

As described above, there are 3 different modes for the serial data. This serial data is send bit by bit in one message after the other. With [MIO_M441_SENTSetIDData\(\)](#) the ID, CRC and Data of the serial message is set. For further details see the SAE J2716.

MIO_M441_SENTTriggerMsgNo()

```
void MIO_M441_SENTTriggerMsgNo(int Slot, int Channel, int MsgNo);
```

Description

This function causes the message with number <MsgNo> to be sent on the SENT bus. This only works if *SendContinous* is set to "0" by calling [MIO_M441_SENConfig\(\)](#). Otherwise the messages are sent in subsequent order anyway.
MsgNo must be in the range from 0 .. 15 and specifies the buffer number.

Examples

All Channels as PWM out with Open Collector

```
#define NCHANNELS    16
#define Slot_PWMOut 1

struct {
    double Current[16];
    int OverloadMask;
    int Overload[16];
    double Freq[16];
    double DutyCycle[16];
} tIO;
tIO IO;

int IO_Init (void) /* RTMaker: IO_Start */
{
    int c /* channel */;
    if (MIO_M441_Config (Slot_PWMOut) < 0) return RT_Failure;
    MIO_M441_EnableChannels(Slot_PWMOut, 0xffff);
    MIO_M441_ResetOverCurrentState (Slot_PWMOut, 0xffff);
    for (c=0; c<NCHANNELS; c++) {
        MIO_M441_SetMode (Slot_PWMOut, c,
                           1 /* Number of Pulses: Single Pulse Series*/,
                           0 /* DataChangeMode: Immediately*/,
                           0 /* PulseWidthMode: Duty Cycle*/,
                           M441_FrequencyOut | M441_EnablePullGND,
                           0 /* No Quadratur-Encoder */ );
        MIO_M441_SetFrequency (Slot_PWMOut, c,
                               0 /* Pulse 0 */,
                               1000.0 /* Init Frequency with 1000Hz */ );
        MIO_M441_SetDutyCycle (Slot_PWMOut, c,
                               0 /* Pulse 0 */,
                               0.5 /* Init Duty Cycle with = 50% */ );
    }
}

void IO_In (void)
{
    int c /* channel */;

    IO.OverloadMask = MIO_M441_GetOverCurrentMask (Slot_PWMOut);
    for (c=0; c<NCHANNELS; c++) {
        /* IO.Current[c] = MIO_M441_GetCurrent (Slot_PWMOut, c); */ /* Makes no sence with OC */
        IO.Overload[c] = (IO.OverloadMask >> c) & 0x01;
    }
}

void IO_Out (void)
{
    int c /* channel */;

    for (c=0; c<NCHANNELS; c++) {
        if (IO.OverloadMask != 0x0000) {
            MIO_M441_ResetOverCurrentState(Slot_PWMOut, IO.OverloadMask)
```

```

        }
        MIO_M441_SetFrequency (Slot_PWMOut, c,
            0 /* Pulse 0 */,
            IO.Freq[c]);
        MIO_M441_SetDutyCycle (Slot_PWMOut, c,
            0 /* Pulse 0 */,
            IO.DutyCycle[c]);
    }
}

```

All channels as PWM out with internal reference

```

#define NCHANNELS 16
#define Slot_PWMOut 1

struct {
    double Current[16];
    int OverloadMask;
    int Overload[16];
    double Freq[16];
    double DutyCycle[16];
} tIO;
tIO IO;

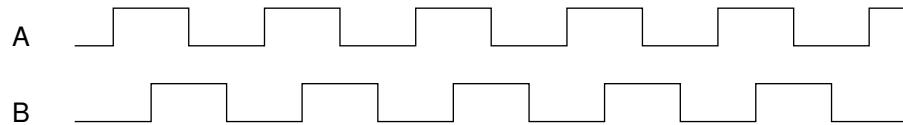
int IO_Init (void) /* RTMaker: IO_Start */
{
    int c /* channel */;
    if (MIO_M441_Config (Slot_PWMOut) < 0) return RT_Failure;
    MIO_M441_EnableInternalReference(Slot_PWMOut, 1 /* Enable */);
    MIO_M441_EnableChannels(Slot_PWMOut, 0xffff);
    MIO_M441_ResetOverCurrentState (Slot_PWMOut, 0xffff);
    for (c=0; c<NCHANNELS; c++) {
        MIO_M441_SetMode (Slot_PWMOut, c,
            1 /* Number of Pulses: Single Pulse Series*/,
            0 /* DataChangeMode: Immediatly*/,
            0 /* PulseWidthMode: Duty Cycle*/,
            M441_FrequencyOut | M441_EnablePullGND | M441_UseRef1,
            0 /* No Quadratur-Encoder */ );
        MIO_M441_SetFrequency (Slot_PWMOut, c,
            0 /* Pulse 0 */,
            1000.0 /* Init Frequency with 1000Hz */);
        MIO_M441_SetDutyCycle (Slot_PWMOut, c,
            0 /* Pulse 0 */,
            0.5 /* Init Duty Cycle with = 50% */);
    }
}

void IO_In (void) {} /* see above */
void IO_Out (void) {} /* see above */

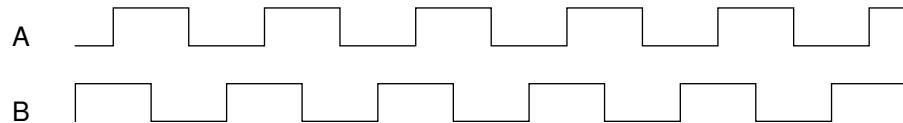
```

Quadrature Encoder with REFO for 2 channels

0 - Turning Right, B legs 0.5rad after A



1 - Turning Left, B leads 0.5rad before A



```
#define NCHANNELS    2
#define Slot_PWMOut 1

struct {
    double Freq[16];
    double DutyCycle[16];
} tIO;
tIO IO;

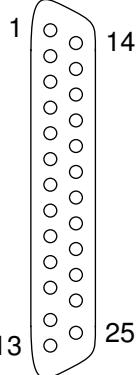
int IO_Init (void) /* RTMaker: IO_Start */
{
    int c /* channel */;
    if (MIO_M441_Config (Slot_PWMOut) < 0) return RT_Failure;

    MIO_M441_EnableChannels(Slot_PWMOut, 0x000f);
    MIO_M441_ResetOverCurrentState (Slot_PWMOut, 0x000f);
    for (c=0; c<NCHANNELS; c+=2) { /* each second channel must be configured only */
        MIO_M441_SetMode (Slot_PWMOut, c,
                           1 /* Number of Pulses: Single Pulse Series*/,
                           0 /* DataChangeMode: Immediately*/,
                           0 /* PulseWidthMode: Duty Cycle*/,
                           M441_FrequencyOut | M441_EnablePullGND | M441_UseRef0,
                           1 /* Quadrature-Encoder */);
        MIO_M441_SetFrequency (Slot_PWMOut, c,
                               0 /* Pulse 0 */,
                               1000.0 /* Init Frequency with 1000Hz */);
        /* MIO_M441_SetDutyCycle () does not work with Quadrature encoder channels!!! */
    }
    MIO_M441_QuadEncSetPhase(Slot_PWMOut, 0, 0, 0.5 * M_PI);
    MIO_M441_QuadEncSetPhase(Slot_PWMOut, 2, 0, (-0.5 * M_PI));
}
```

10.5 Discontinued M-Modules

10.5.1 M34: Analog Inputs (16/8 channels)

Connector assignment



Pin	Signal	Pin	Signal
1	In_0	14	In_8 (In_0-)
2	In_1	15	In_9 (In_1-)
3	In_2	16	In_10 (In_2-)
4	In_3	17	In_11 (In_3-)
5	In_4	18	In_12 (In_4-)
6	In_5	19	In_13 (In_5-)
7	In_6	20	In_14 (In_6-)
8	In_7	21	In_15 (In_7-)
9	BI	22	Trigger
10	GND	23	+15V
11	GND	24	-15V
12	GND	25	+5V
13	GND		

Function Overview

Initialization and Configuration

- [MIO_M34_Config\(\)](#)

Get Input Signals

- [MIO_M34_Read\(\)](#)
- [MIO_M34_Read_bp\(\)](#)
- [MIO_M34_ReadV\(\)](#)

MIO_M34_Config()

```
int MIO_M34_Config(int Slot)
```

Description

(see function description for [MIO_M35_Config\(\)](#))

MIO_M34_Read()

```
int MIO_M34_Read(int Slot, int Ch, int Gain)
```

Description

(see function description for [MIO_M35_Read\(\)](#))

MIO_M34_Read_bp()

```
int MIO_M34_Read_bp(int Slot, int Ch, int Gain)
```

Description

(see function description for [MIO_M35_Read_bp\(\)](#))

MIO_M34_ReadV()

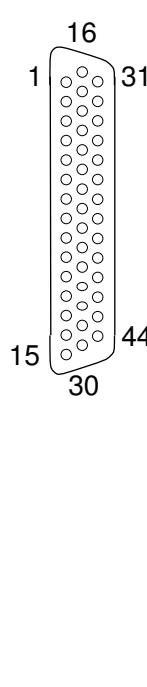
```
void MIO_M34_ReadV(int Slot, unsigned ChMask, int GainMode, int *Values)
```

Description

(see function description for [MIO_M35_ReadV\(\)](#))

10.5.2 M72: Motion Counter / Multipurpose Module

Connector assignment M72



	Pin	Signal	Pin	Signal	Pin	Signal
1	1		16		31	
	2	AIN0+	17	AIN1+	32	AIN2+
	3	AIN0-	18	AIN1-	33	AIN2-
	4		19		34	
	5	BIN0+	20	BIN1+	35	BIN2+
	6	BIN0-	21	BIN1-	36	BIN2-
	7		22		37	
	8	CIN0+	23	CIN1+	38	CIN2+
	9	CIN0-	24	CIN1-	39	CIN2-
	10		25		40	
	11	DIN0+	26	DIN1+	41	DIN2+
	12	DIN0-	27	DIN1-	42	DIN2-
	13		28		43	
	14	Out0	29	Out1	44	Out2
	15	IGND	30	Out3		

Function Overview

Initialization and Configuration

- [MIO_M72_Config\(\)](#)
- [MIO_M72_ConfigFromFile\(\)](#)
- [MIO_M72_Download\(\)](#)

M72 as Motion Counter

- [MIO_M72_MC_SetMode\(\)](#)

Counter Get / Set

- [MIO_M72_MC_GetCounter\(\)](#)
- [MIO_M72_MC_SetCounter\(\)](#)

Set Output Signals

- [MIO_M72_MC_SetOutPort\(\)](#)

MIO_M72_Config()

```
int MIO_M72_Config (int Slot, tMIO_M72_Cfg *CfgData)
```

Description

By calling the function [MIO_M72_Config\(\)](#) the configuration of the I/O hardware, i.e. the allocation of a M-Module card location with a module of type M72, is specified. This function must be called one time, before any attempted access to the module.

The argument CfgData is a pointer to a data struct of type `tMIO_M72_Cfg` with user specific data, like firmware (PLD data for the ALTERA FLEX device on the M72) and user specific Config, Reset and Delete functions. The data type `tMIO_M72_Cfg` is defined as follows:

Listing 10.10: The `tMIO_M72_Cfg` struct

```
51:  typedef struct tMIO_M72_Cfg {
52:      char          *Ident;
53:      unsigned       Version;
54:      long          VersionDate;
55:      unsigned char *PlpData;
56:      int           PlpDataSize;
57:      int           (*UsrConfigFunc) (int Slot);
58:      void          (*UsrResetFunc) (int Slot);
59:      void          (*UsrDeleteFunc) (int Slot);
60:  } tMIO_M72_Cfg;
```

The data fields `Ident`, `Version` and `VersionDate` are used to identify the PLD data uniquely. `PlpData` and `PlpDataSize` refer to the PLD data itself, which is to be downloaded to the M72 module and which defines the functionality of the Module. `PlpData` is a pointer to an array of type `unsigned char` containing the PLD data, whereas the first 4 bytes are used for length information of the following data. `PlpDataSize` is the size of the `PlpData` array, including the 4 bytes length information at the beginning.

With the parameters `UsrConfigFunc`, `UsrResetFunc` and `UsrDeleteFunc`, the user can specify his own Config, Reset and Delete functions for individual initialization or configuration of the module. The user Config function `UsrConfigFunc` is called right after the firmware download, if the function pointer is not `NULL`. The user reset function `UsrResetFunc` (if not `NULL`) is called by the function [MIO_ResetModules\(\)](#) to (re-)initialize the module hardware to its default state (according to the loaded firmware). On deletion of the M72 module, when calling the functions [MIO_M72_Delete\(\)](#) or [MIO_DeleteAll\(\)](#), the user delete function `UsrDeleteFunc` is called (if not `NULL`). This function can be used to e.g. deactivate the hardware or to free unused memory.

If the module cannot be configured-initialized the function returns an error code, otherwise 0 is returned.



How to operate the M72 as Motion Counter

In order to operate the M72 module in its original operating mode as Motion Counter, there is a predefined configuration data `M72_Plidata`, which has to be passed to [MIO_M72_Config\(\)](#) for initialization:

```
if (MIO_M72_Config(SlotM72, M72_Plidata))
    printf("Error MIO_M72_Config()\n");
```

To use this configuration data the following file must be included:

```
# include <mio_m72_pld.h>
```

This call to [MIO_M72_Config\(\)](#) downloads the original Motion Counter firmware into the module. After the download, the module is in its initial state, with no timer active.

MIO_M72_ConfigFromFile()

```
int MIO_M72_ConfigFromFile(int Slot, const char *FName,
                           int (*UsrConfigFunc)(int Slot),
                           void (*UsrResetFunc) (int Slot),
                           void (*UsrDeleteFunc)(int Slot))
```

Description

Alternatively to the function [MIO_M72_Config\(\)](#), [MIO_M72_ConfigFromFile\(\)](#) can be used to configure a M72 module by loading the firmware out of a file. According to the configuration data, which is needed with [MIO_M72_Config\(\)](#), this function also accepts user defined Config, Reset and Delete functions.

This function registers a M-Module of type M72, located at slot `Slot`, loads the firmware from the file `FName` and calls the user defined `Config` function `UsrConfigFunc` (if not `NULL`). The user defined reset and delete functions `UsrResetFunc` and `UsrDeleteFunc` are registered as well.

If the module cannot be configured-initialized, the function returns an error code, otherwise 0 is returned.

MIO_M72_Download()

```
int MIO_M72_Download(int Slot, const unsigned char *Data, int DataLen)
```

Description

The function [MIO_M72_Download\(\)](#) takes a new firmware `Data` and downloads `DataLen` bytes to the M72 module. The firmware will be automatically downloaded when calling [MIO_M72_Config\(\)](#). So, there is no need to call this function separately.

MIO_M72_Delete()

```
void MIO_M72_Delete(int Slot)
```

Description

This function calls the user defined `Delete` function to deactivate the M-Module. If there was no user defined `Delete` function specified during initialization with [MIO_M72_Config\(\)](#) or [MIO_M72_ConfigFromFile\(\)](#), the function does nothing.

MIO_M72_MC_SetMode()

```
void MIO_M72_MC_SetMode(int Slot, int Ch, MIO_M72_Mode Mode)
```

Description

Sets the operating mode of counter Ch to Mode. The following operating modes are supported:

Mode	Operating mode
0	inactive
1	single count
2	1x quadrature
3	2x quadrature
4	4x quadrature
5	frequency measurement
6	pulse width “high”
7	pulse width “low”
8	period measurement
9	timer

Description of operating modes

- Single Count mode:

This is the operation mode for classic up- and down-counting.

A rising edge at `xIN0` increments the counter, a rising edge at `xIN1` decrements the counter. A rising edge at `xIN2` can clear or load the counter, depending on the programming of the Counter Control Register.

Alternatively, a connection to the comparator unit for clearing or loading is possible. The Counter Control Register also determines the event on which the counter value is passed to the Counter Output Latch Register, as is the case for all counter modes. The latched counter value can be read out from the Counter Output Latch Register.

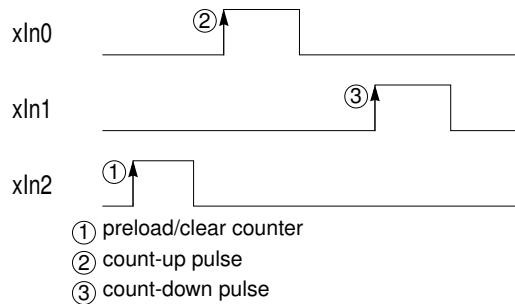


Figure 10.8: Single Count mode

- 1x Quadrature mode:

Similar to Single Count mode, 1x Quadrature mode is a classic up- and down-counting mode.

However, a rising edge at `xIN0` generates a count pulse in any case. `xIN1` defines the direction: “high” means counting up, “low” means counting down. (When using RS422 signals, it is possible to invert polarities by reversing the input lines.)

In 1x Quadrature mode you can already evaluate the signals of a rotary encoder. The “A” signal is connected to `xIN0`, the “B” signal is connected to `xIN1`. Depending on the direction of rotation, the counter value is incremented or decremented by 1 with each pulse at `xIN0`. The signal often called “Z” can be connected to `xIN2` and can be used to set, clear or latch the counter value as in Single Count mode.

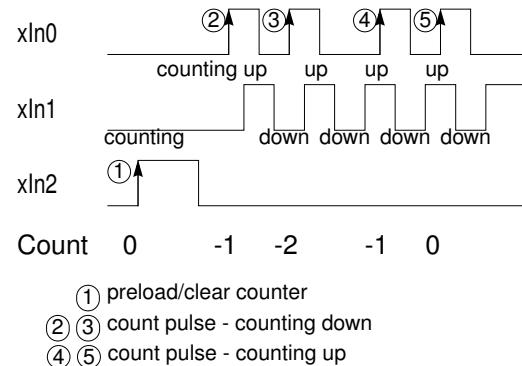


Figure 10.9: 1x Quadrature mode

- 2x Quadrature mode:

In 2x Quadrature mode the `xIN0` and `xIN1` signals are evaluated differently than in Single Count and 1x Quadrature mode.

Each a rising and falling edge at `xIN0` generate a count pulse. `xIN1` defines the direction, depending on `xIN0`: “high” with a falling edge and “low” with a rising edge at `xIN0` count down, “high” with a rising edge and “low” with a falling edge at `xIN0` count up.

This behavior make this mode suitable only for rotary or similar sensors. The “A” signal is now evaluated on both edges.

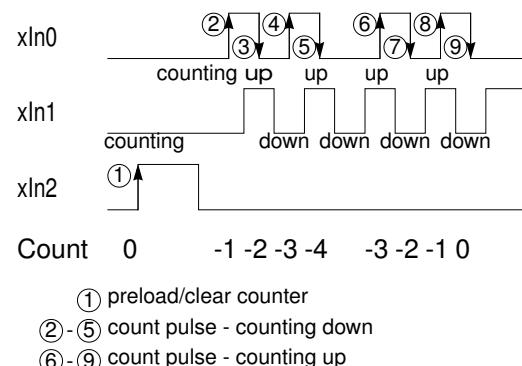


Figure 10.10: 2x Quadrature mode

- 4x Quadrature mode:

The counter will be cleared or preloaded by signal `xIN2` or by software. A rising and falling edge at `xIN0` and `xIN1` create a count pulse. The counter increments when `xIN0` is before `xIN1`, the counter decrements when `xIN0` is after `xIN1`.

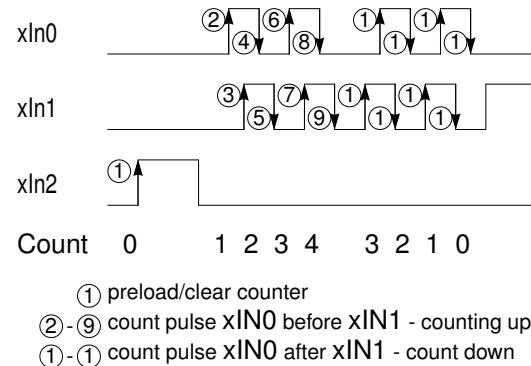


Figure 10.11: 4x Quadrature mode

- Frequency Measurement mode:

The frequency will be measured at input `xIN0` after activating the internal time base (10ms) (bit `TimeBase` in the Counter Control Register). The time base clears the counter and starts the measurement, the counter increments at each rising edge at `xIN0`. The measurement stops after the time-out of the time base. The time base generates a Ready interrupt and the counter value is latched.

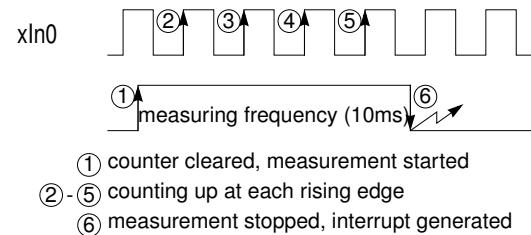


Figure 10.12: Frequency Measurement mode

- Pulse Width “high” mode:

The signal, connected to input `xIN0`, will be measured. The counter can be cleared or preloaded, either using `xIN2`, or by software. The measurement starts after the rising edge at `xIN0`. It stops with the falling edge at `xIN0`, activates the Ready interrupt and latches the counter value. The counter clock (2.5 MHz) controls the counter.

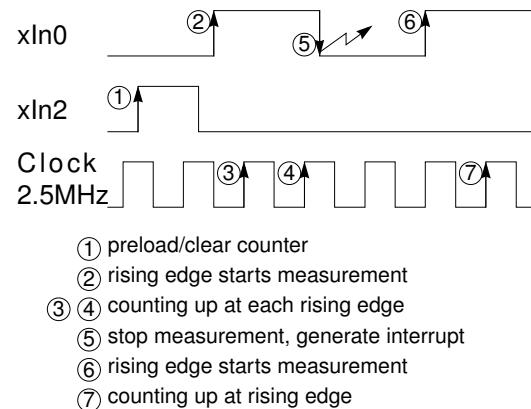


Figure 10.13: Pulse Width “high” mode

- Pulse Width “low” mode:

The signal, connected to input `xIN0`, will be measured. The counter can be cleared or preloaded, either using `xIN2`, or by software. The measurement starts after the falling edge at `xIN0`. It stops with the rising edge at `xIN0`, activates the Ready interrupt and latches the counter value. The counter clock (2.5 MHz) controls the counter.

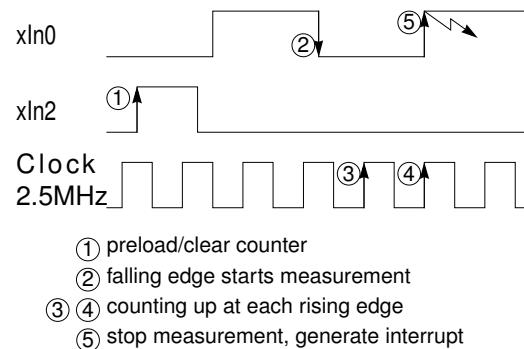


Figure 10.14: Pulse Width “low” mode

- Period Measurement mode:

The signal, connected to input `xIN0`, will be measured. The counter can be cleared or preloaded, either using `xIN2`, or by software. The measurement starts after the rising edge at `xIN0`. It stops with the next rising edge at `xIN0`, activates the Ready interrupt and latches the counter value. The counter clock (2.5 MHz) controls the counter.

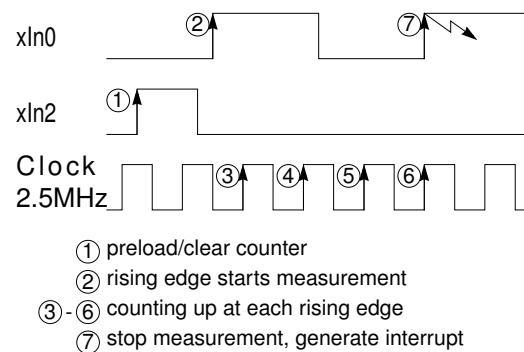


Figure 10.15: Period Measurement mode

- Timer mode:

To initialize the timer, the counter must be cleared or preloaded using `xIN2` or by software, then the `Comparator_X` register must be loaded with the timer value. The timer starts either on `xIN2` or by software (see bit `TimerStart` in the Counter Control Register). `Comparator_X` generates the Comparator interrupt after the time-out and stops the counter. The M72 can be programmed to start further timer sequences through the Counter Control Register after the Comparator interrupt. Depending on the Preload and Clear bits, the Comparator interrupt clears or preloads and then starts the counter.

again. Signal `xIN1` controls the count direction. The counter increments when `xIN1` is “high” and decrements when `xIN1` is “low”. The counter runs with the internal counter clock (2.5MHz).

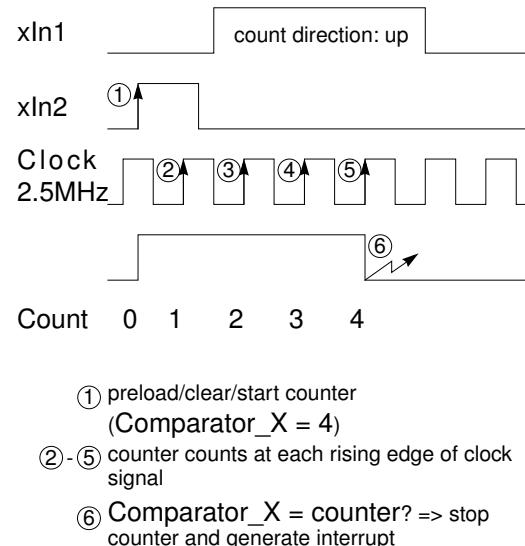


Figure 10.16: Timer mode

MIO_M72_MC_GetCounter()

```
long MIO_M72_MC_GetCounter(int Slot, int Ch)
```

Description

Returns the current value of the 32 bit counter on slot `Slot` and channel `Ch`.

MIO_M72_MC_SetCounter()

```
void MIO_M72_MC_SetCounter(int Slot, int Ch, long Value)
```

Description

Preloads the counter at slot `Slot` and channel `Ch` with `Value`.

MIO_M72_MC_SetOutPort()

```
void MIO_M72_MC_SetOutPort(int Slot, unsigned Value)
```

Description

The function `MIO_M72_MC_SetOutPort()` sets the values of the four output signals `Out1`, `Out2`, `Out3` and `Out4`.

`Slot` gives the M-Module card location.

`Value` defines the states for the output signals, whereas only the 4 least significant bits are used. If bit n is 1, then output signal $n+1$ will be set "high", otherwise output signal $n+1$ will be set "low".

Chapter 11

USB IO – USB Input/Output Interfaces

11.1 PowerUTA

PowerUTA is an interface to control power supplies with an analog programming interface via an USB port of the real time system. PowerUTA is supported by Xpack4/XENO

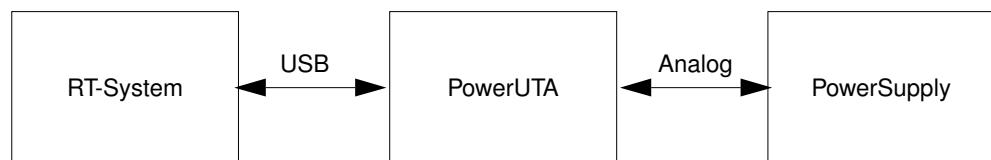
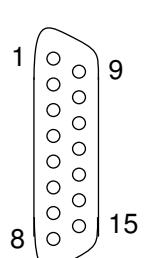


Figure 11.1: PowerUTA

11.1.1 Connector assignments



Pin	Description	Notes
1	analog output channel 1	0..10V
2	analog output channel 2	0..10V
3	n.c.	
4	analog ground	
5	switchable output 1 (Remote)	open collector, max. 20V
6	switchable input 1 (OT)	L=0V, H=5V
7	switchable input 1 (CP)	L=0V, H=5V
8	switchable input 3 (Err)	L=0V, H=5V
9	analog input channel 1 (Umon)	0..10V
10	analog input channel 1 (Imon)	0..10V
11	digital ground	
12	n.c.	
13	switchable output 2 (Standby)	open collector, max. 20V
14	switchable input 4 (OVP)	L=0V, H=5V
15	switchable input 5 (CC/CV)	L=0V, H=5V

11.1.2 Functional description

struct PowerUTA

```
typedef struct {
    struct DevUTA *uta;
    float Uout;
    float Iout;
    float Umon;
    float Imon;
    char Remote; /* (out) remote control 1 = 0V, active */
    char Standby; /* (out) standby 1 = 0V, power off */
    char Error; /* (in) power failure */
    char CC_CV; /* (in) const. current/volt 1 = CC 1 */
    char OVP; /* (in) over voltage protection */
    char OT; /* (in) over temperature */
    char CP; /* (in) */
} PowerUTA;
```

Description

One variable of the type *PowerUTA* for each power supply must be defined inside the real time code. This variable contains all the control signals for and the monitoring signals from the power supply. The data is updated automatically every 50ms by a background thread after [PowerUTA_Init \(\)](#) is called.

Example:

```
typedef struct {
...
    PowerUTA PowerSupply1;
    PowerUTA PowerSupply2;
} tIOVec;
```

Description of the Struct Members

As shown below, some of the members of the struct PowerUTA have different meanings or different logic levels depending on the connected power supply.

Table 11.1: PowerUTA Signal Description

Variable	Value	EA-PS9018-100	Delta SM15-100	GEN2038 / GEN2076
Uout		Desired Output Voltage		
IOut		Desired Output Current		
Umon		Current Output Voltage		
Imon		Current Output Current		
Remote	0	Remote Programming disabled	---	---
	1	Remote Programming enabled	---	---
Standby	0	Output enabled	Output enabled	Output enabled
	1	Output disabled	Output disabled	Shut-Off
Error	0	---	OK	Fault Condition
	1	---	Power Sink Overload	OK
CC_CV	0	The power supply works in voltage limitation mode		
	1	The power supply works in current limitation mode		
OVP	0	OVP is inactive	OVP is inactive	---
	1	OVP is active	OVP is active	---
OT	0	---	Temperature is OK	---
	1	---	Over Temperature	---
CP	0	---	---	---
	1	---	---	---

The function of the variables depend on the connected power supply. This is because not every power supply offers the same functionality and the same logic levels for a function via the analog interface. Thus if your power supply is not shown above, see the manual of your power supply and the CarMaker Hardware Manual for further information.

PowerUTA_Init ()

Function prototype

```
int PowerUTA_Init (PowerUTA *pwr, double Umax, double Imax, const char *DevName);
```

Description

This function assigns a PowerUTA variable and a name to one power supply. Thus it must be called one time for each power supply during IO_Init(). Of course, there must be one PowerUTA variable for each power supply. Additionally, the maximum voltage and the maximum current for the power supply are specified. Therefore no scaling of voltage and current is needed by the user.

[PowerUTA_Init \(\)](#) starts a background thread that updates the power supply's configuration and its state every 50ms.

PowerUTA_Finish ()

Function prototype

```
void PowerUTA_Finish (PowerUTA *pwr);
```

Description

This function stops the background thread. It should be called just before the CarMaker executable is finished.

PowerUTA_DeclQuants ()

Function prototype

```
void PowerUTA_DeclQuants (PowerUTA *pwr, const char *NamePrefix);
```

Description

`PowerUTA_DeclQuants ()` declares the data dictionary quantities for the PowerUTA variables. The `NamePrefix` is a text string which is used as a prefix in the data dictionary.

Example

If the prefix is "PowerSupply1", the quantity name are

- PowerSupply1.Uout
- PowerSupply1.Iout
- PowerSupply1.Umon
- PowerSupply1.Imon
- PowerSupply1.Standby

and so on ...

Chapter 12

FlexRay

In the last years, more and more ECUs with different functions has been integrated in modern vehicles. They improve driving safety, optimize driving performance or just add some more comfort for the driver and his passengers. However, the more intelligent components are stowed in a car, the more communication traffic has to be managed in order to guarantee a perfect cooperation between all of them.

Traditionally, the standard of connecting different ECUs within a vehicle is the CAN bus, allowing a data rate of at most 1 MBit/s. For complex board nets, especially those with x-by-wire applications, developers have been struggling with the limitations of the CAN bus – not only because of its limited bandwidth, but also due to its limitations when used with applications which require extreme data integrity.

12.1 Overview

12.1.1 Requirements

FlexRay communication with CarMaker/HIL on a signal and frame basis requires additional software for the export of communication parameters, signal and frame descriptions from FIBEX data bases:

Software Module	Description
FlexConfig Developer	Design and configuration software for automotive networks from Eberspächer Electronics : <ul style="list-style-type: none">• Management of FlexRay clusters• Configuration of cluster and node parameters• ECU, frame, and signal description• Import of FIBEX / AUTOSAR / CANdb files
IPG CarMaker/HIL Plugin	Export of frame and signal description for CarMaker/HIL: <ul style="list-style-type: none">• Selection of real and virtual ECUs• Generation of frame and signal list• Generation of CHI configuration file for FlexCard

12.1.2 Basic information

In order to connect a Hardware-in-the-Loop system to a FlexRay bus, a powerful FlexRay interface is required. For CarMaker HIL TestSystems, IPG has found the FlexCard PMC and FlexCard PMC II from Eberspächer Electronics (<http://www.eberspaecher-electronics.com>), that meet the requirements.

However, having a FlexRay controller and connecting it with other devices does not make a running configuration yet. In contrast to the CAN bus, you need to know more than just the baud rate. Instead, there are a couple of configuration and timing parameters, as well as message and signal descriptions, which describe the topology of a FlexRay network. Without knowing this information, you cannot connect and talk to other bus members.

For FlexRay networks, the [ASAM Group](#) has introduced the FIBEX standard to describe the network topology, together with all network nodes, data frames and signals. Having a network description in the FIBEX format, the FlexConfig Utility from Eberspächer Electronics can be used to connect a CarMaker HIL TestSystem with FlexRay devices for rest bus simulation.

In combination with a CarMaker/HIL Plug-in, FlexConfig is used to export special CarMaker Info files, describing FlexRay frames and signals which are transferred between real and virtual ECUs. Additionally, a *.CHI file is generated as well, which allows to initialize the communication controllers of the FlexCard easily. Based on this and together with a mapping table which describes the relation between FlexRay signals and CarMaker model variables, a rest bus simulation can be implemented.

The following Figure 12.1 gives an overview about how FIBEX data bases are imported for FlexRay rest bus simulation with CarMaker/HIL:

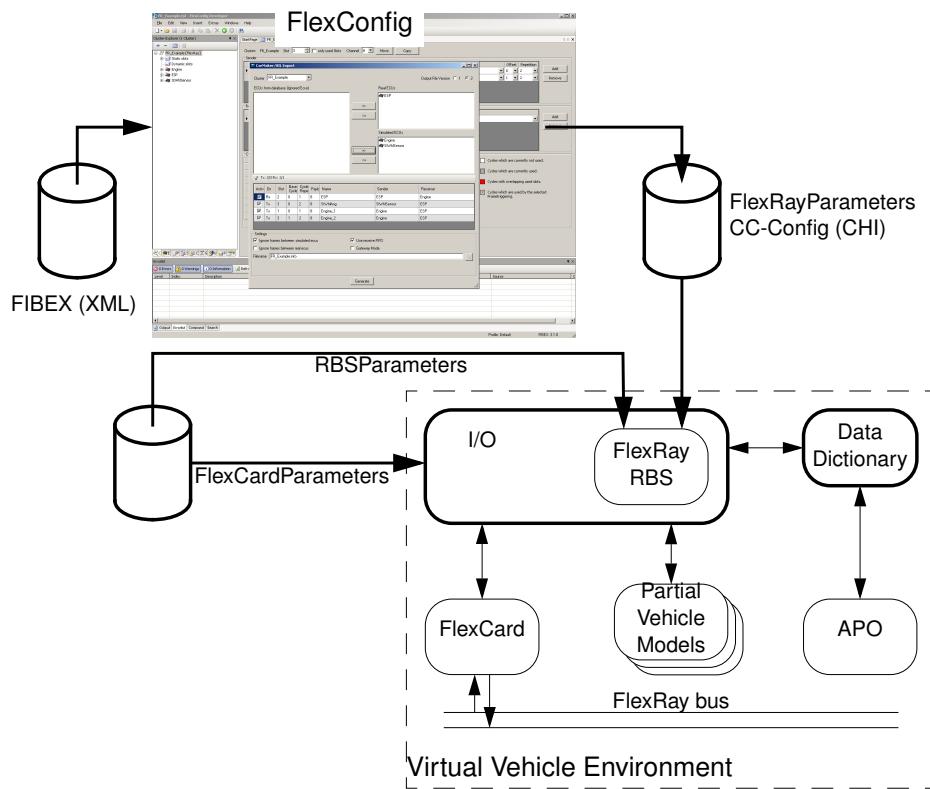


Figure 12.1: FIBEX Import

12.1.3 Features

The description of a FlexRay network is quite complex. Even more, when designing the board network of a car. Several different networks have to be described, including gateways which connect different sub networks like CAN and FlexRay. So, car developers like to use universal tools that allow to design the overall board network. For the exchange with others, the FIBEX standard is often used. FIBEX defines a method to describe controller networks and bus systems of any kind in XML data format.

With FlexConfig, Eberspächer Electronics offers a tool that addresses the network designer as well as the supplier for electronic control units. FlexConfig offers the possibility to import different data formats including FIBEX up to version 3.1.0 and even AUTOSAR, which is beneficial especially for use with CarMaker/HIL.

Since the FlexCard is also a product of Eberspächer, CarMaker benefits from FlexConfigs capabilities to generate *.CHI files for the Bosch ERay FlexRay controller on the FlexCard.

However, having a tool to describe a network topology with ECUs, data frames and signals etc., is not enough to set up a rest bus simulation. An important step is to split up an existing network and to extract and describe the information that has to be provided from the virtual (simulated) part of the network to the remaining real part, the ECUs under test.

For use with FlexConfig, a plug-in has been developed to export a frame and signal description for CarMaker/HIL, together with a FlexRay controller configuration (*.CHI file) to configure the FlexCard as an interface between the ECUs to simulate and the remaining real part of the FlexRay network. Based on this, a FlexRay rest bus simulation has been implemented in CarMaker/HIL, which is configured dynamically with Info files.

Using FlexConfig in combination with the CarMaker/HIL plug-in above, the FlexRay rest bus simulation solution of CarMaker/HIL has the following features:

- Design of FlexRay clusters from scratch.
- Import of FIBEX data bases for FlexRay clusters, designed by other tools.
- Supported FIBEX versions:
 - 1.2.0a
 - 2.0.0d
 - 2.0.1
 - 3.0.0
 - 3.1.0
- Conversion filters for:
 - AUTOSAR 3 to FIBEX 3
 - CANdb to FIBEX 2.0 / 3.0
 - FIBEX 2.0.0d to FIBEX 3.0
 - FIBEX+ to FIBEX 3.0
- Export of lists of frames and signals for configuration of the FlexRay rest bus simulation in CarMaker/HIL:
 - Definition of real ECUs (units under test), Simulated ECUs (required for rest bus simulation) and ignored ECUs (not necessary for rest bus simulation).
 - Possibility to deselect single FlexRay frames which are not of interest.
 - Possibility to filter frames by type (e.g. SERVICE, TPL, DIAG, ...).
- Gateway Mode. This allows to connect real ECUs to several single FlexRay communication controllers in order to control and manipulate the frames and signals between the real ECUs.
- No need to recompile the CarMaker/HIL realtime application. The rest bus simulation is configured with CarMaker Info files and can be changed even without restarting the realtime application.

12.2 FlexConfig

This section describes the usage of the CarMaker/HIL plug-in for FlexConfig and how to export the frame and signal descriptions for use with CarMaker/HIL. For instructions how to use FlexConfig and how to manage a FlexRay cluster, please refer to the FlexConfig user documentation.

Having a FIBEX configuration file, you have to import it in FlexConfig in order to use it as a base for the FlexRay rest bus simulation in CarMaker/HIL. The result of the import is a FlexConfig project (*.fpf file). It is a good idea to save it into the *Data/Misc* folder, located below the CarMaker project directory.

In the following, we will assume a small FlexRay network as shown in [Figure 12.1](#) consisting of three ECUs, an ESP controller named *ESP*, an engine controller named *Engine* and a simple steering wheel sensor device (named *StWhlSensor*). Supposed to be, we want to connect the ESP controller to the test rig, but we do not have the engine controller nor the steering wheel sensor. So, we need to simulate parts of the FlexRay network, which means to simulate the FlexRay traffic for the ECUs Engine and StWhlSensor. The following [Figure 12.1](#) gives an overview of the configuration:

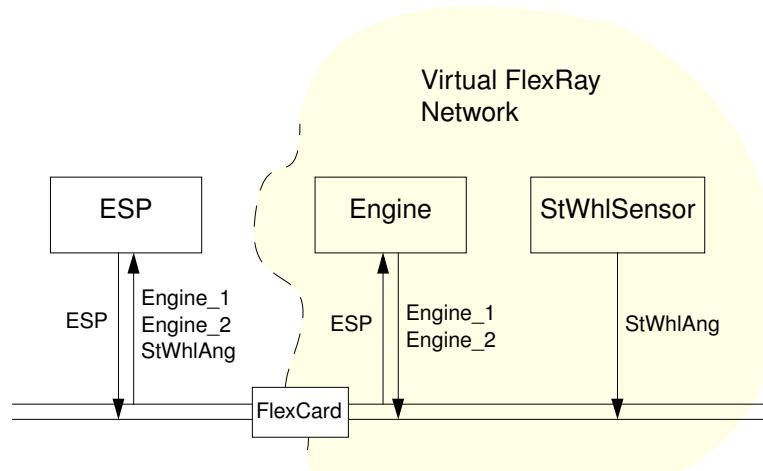


Figure 12.2: FlexRay network example

12.2.1 CarMaker/HIL Plug-in

To open the CarMaker/HIL plug-in, open the *Extras* menu and select *IPG CarMaker/HIL*:

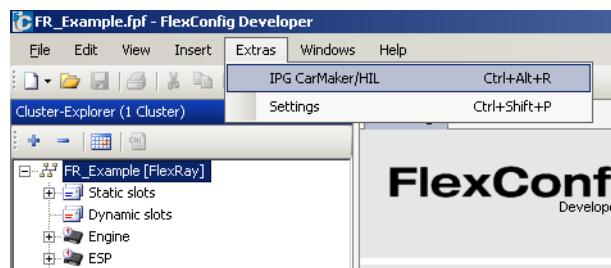


Figure 12.3: FlexConfig Extras menu

If you open the CarMaker/HIL plug-in for the first time since starting FlexConfig or loading a new data base, you will be asked if you want to start with a new configuration or if you want to load an existing one:



Figure 12.4: Loading an existing configuration

Whenever the CarMaker/HIL plugin exports a frame and signal description for rest bus simulation in CarMaker/HIL, the current configuration is saved in a file with the extension *.fccm*. To load an existing configuration, click the *Load* button and select the desired **.fccm* file in the following file browser dialog.

In the CarMaker/HIL Export dialog, you see an overview of the available network nodes (ECUs), a (possibly empty) list of FlexRay data frames and a section for settings and options in the lower part of the dialog.

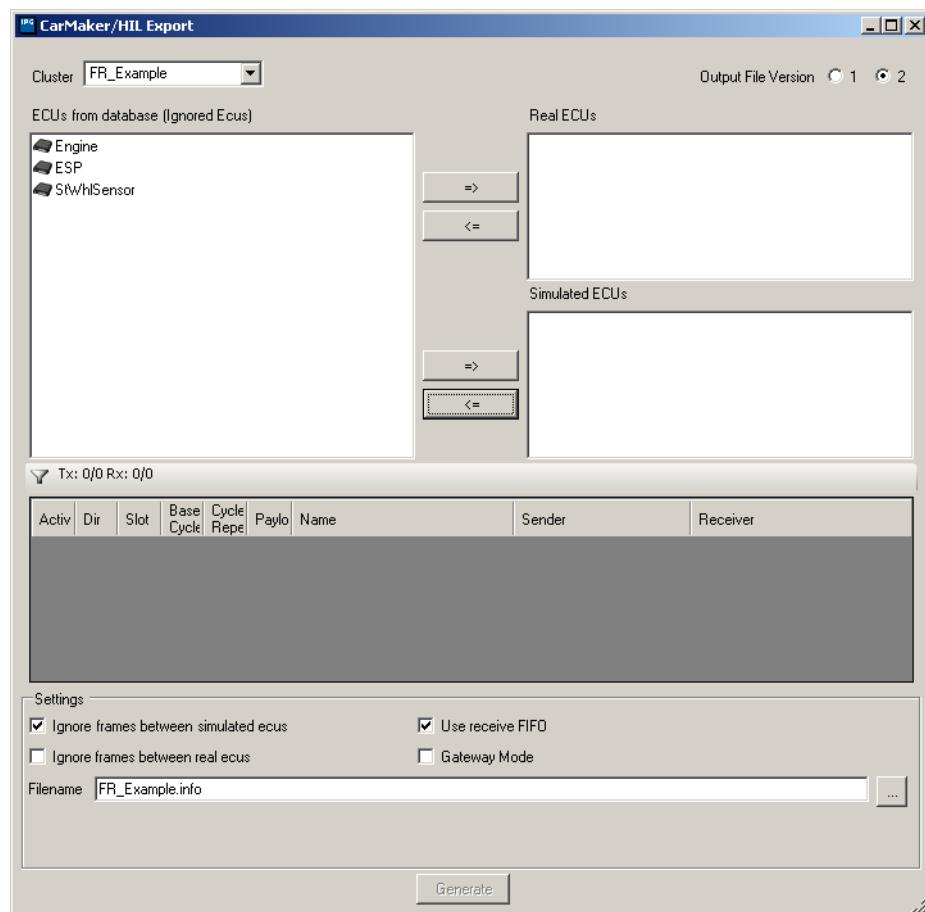


Figure 12.5: IPG CarMaker/HIL popup

Depending on the project, there may be more than one FlexRay network (cluster) defined. Different FlexRay clusters may have different configurations, which are most likely incompatible to each other. Therefore, the CarMaker/HIL plug-in only allows to export information

from one single FlexRay cluster at a time. Exports from multiple clusters require multiple rest bus simulations. The FlexRay cluster for the export can be selected with the *Cluster* selection at the top of the dialog:

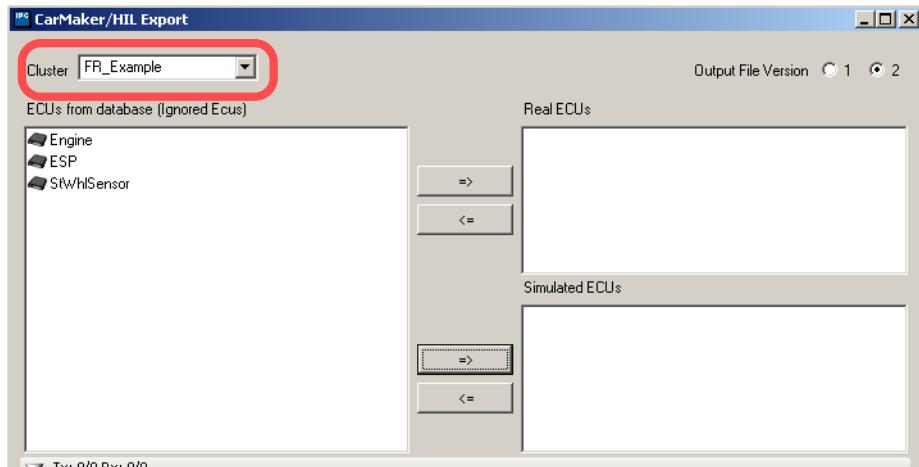


Figure 12.6: IPG CarMaker/HIL popup - Cluster selection



With CarMaker 4.0, the format of the generated Info file with the list of frames and signals has been changed. The new *Output File Version 2* supports exports for rest bus simulation in CarMaker/HIL in *Gateway Mode* and is much more efficient regarding the size of the generated files than with version 1.

All available ECUs within the selected cluster are listed in the upper left part of the dialog. When generating the Info file output for CarMaker/HIL, all ECUs that remain in this list will be ignored, i.e. no frames that are sent or received only by these ECUs will be exported:

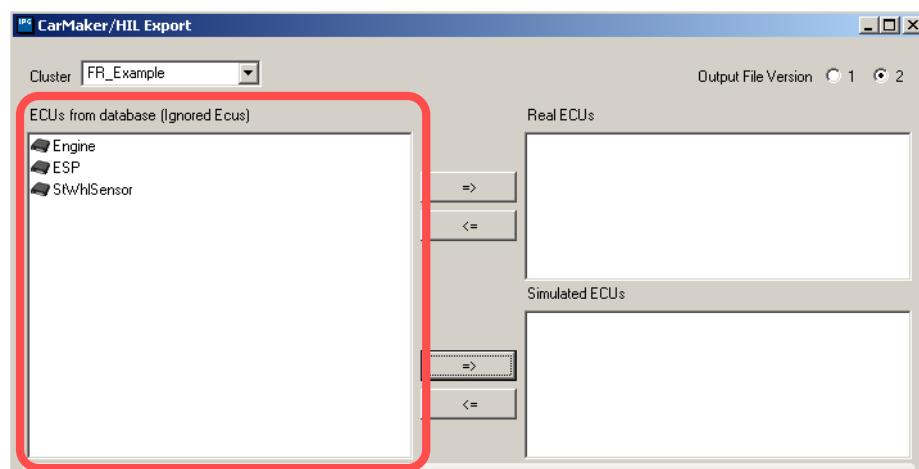


Figure 12.7: IPG CarMaker/HIL popup - Available / ignored ECUs

Setup of Real and Virtual part of a network

Using the arrow buttons, you can now move ECUs from the ignored list either to list of *Real ECUs* at the upper right part, or to the list of *Simulated ECUs* below. In our example, you would move the ECU *ESP* to the list of *Real ECUs*, while moving the *Engine* and *StWhlSensor* ECUs to the list of *Simulated ECUs*:

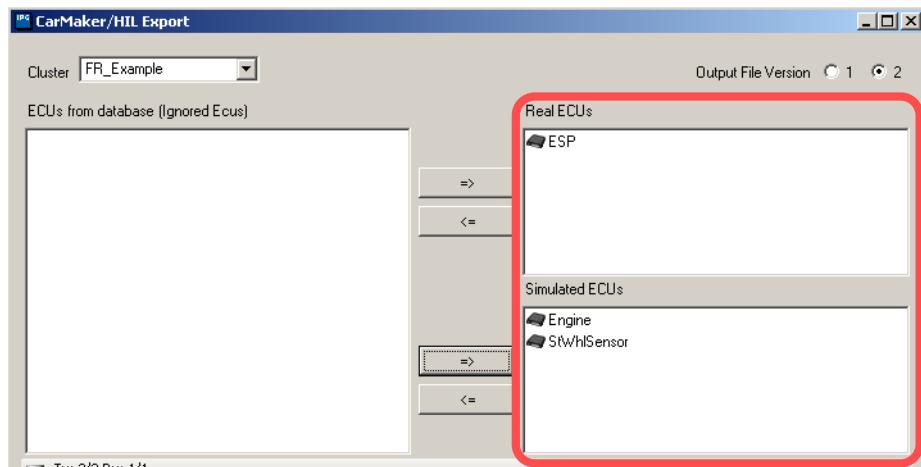


Figure 12.8: IPG CarMaker/HIL popup - Real / simulated ECUs

Once an ECU is selected either as real or simulated ECU, its Tx- and Rx-frames are added to the frame list below:

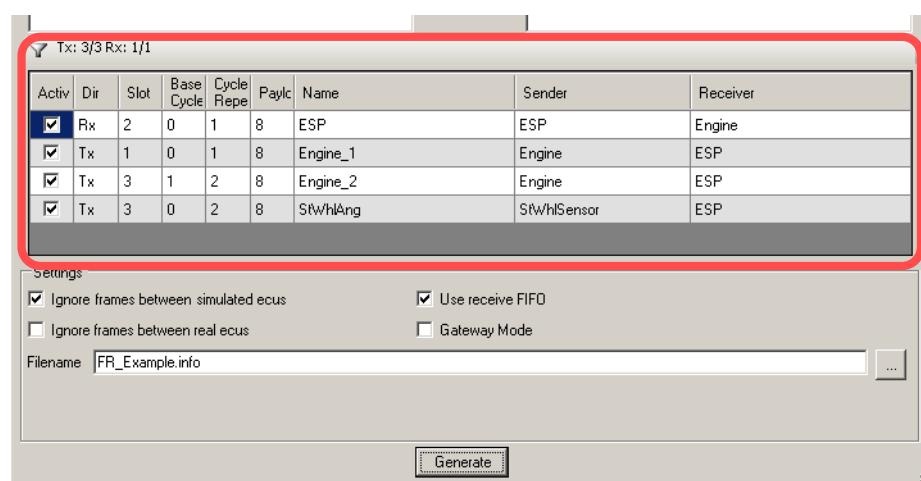


Figure 12.9: IPG CarMaker/HIL popup - List of Tx- and Rx- frames

By default, all frames will be *Active*. For different reasons, however, it might be necessary to deactivate single frames. Deactivated frames will not be generated.

Frame filter

Another way to reduce the number of generated frames effectively, is to filter out different frame types globally. This function is available when pressing the button with a funnel-shaped symbol, located above the frame list:

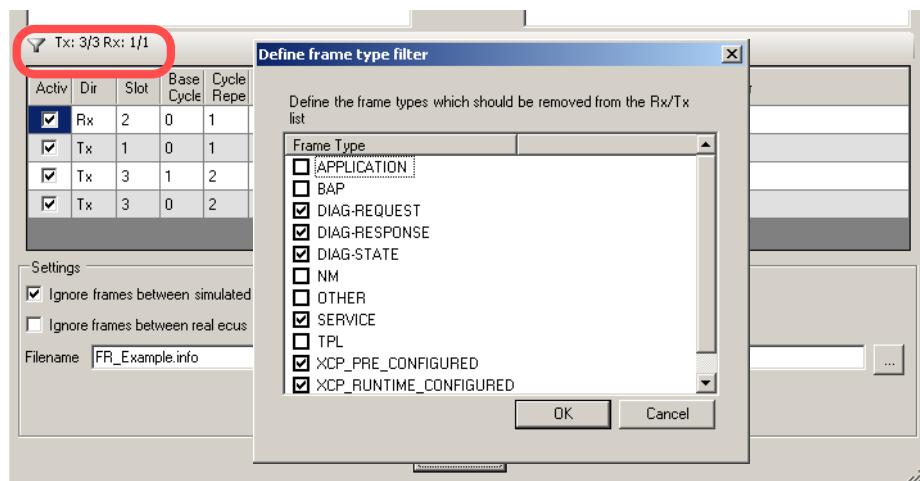


Figure 12.10: IPG CarMaker/HIL popup - Frame filter

In the filter dialog, you can select types of frames which should be opted out, i.e. will not be listed in the frame list. Typical frame types are diagnostic, service or calibration frames, which do not carry regular and vital data for other ECUs.

Coldstart capability

In order to start-up the communication on a FlexRay network, there need to be at least 2 coldstart nodes. Otherwise, the remaining FlexRay nodes will not be able to synchronize their local clock.

Depending on the capabilities of the selected real ECUs, the remaining real part of the FlexRay network may not be able to start-up. In this case, the coldstart capability needs to be provided partially or completely by the FlexCard.

When adding ECUs to the list of real or simulated ECUs, the CarMaker/HIL plug-in checks if the resulting cluster network (including the FlexCard) is coldstart capable or not. If there are not two or more coldstart nodes among the real ECUs, the plug-in tries to find a frame in the static segment which can be used as a sync frame. This can be the sync frame of another start-up node among the simulated ECUs, or any other Tx frame in the static segment, which is sent every cycle. If there is found no suitable frame, a warning will be shown at the bottom of the dialog:

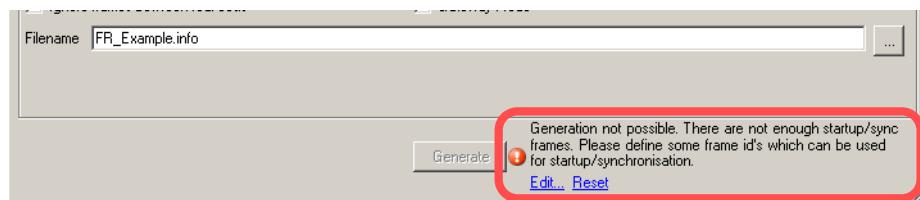


Figure 12.11: IPG CarMaker/HIL popup - Sync frame warning

In order to solve the conflict, press *Edit* and select a slot number in the following pop-up. The selected slot will be used to place a possibly empty sync frame for start-up:

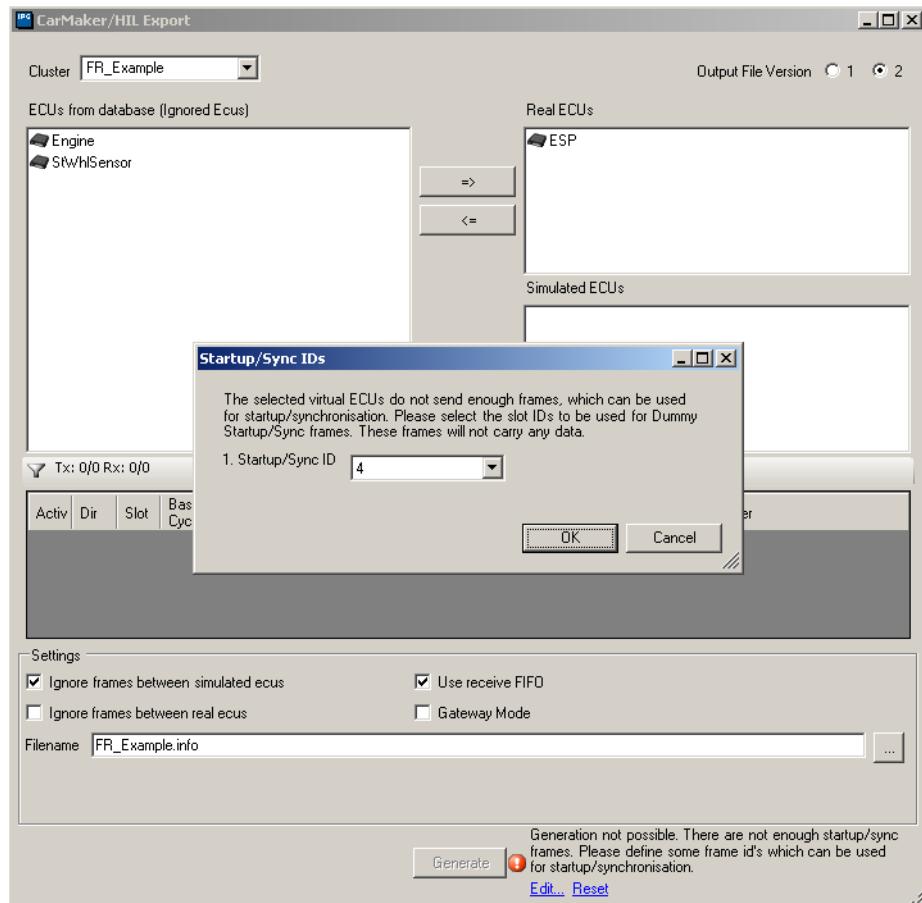


Figure 12.12: IPG CarMaker/HIL popup - Sync frame selection

Settings and Generation step

For a working rest bus simulation only those frames need to be considered, which are transferred between real ECUs and simulated ECUs. Frames, that are transferred only between real ECUs or only between simulated ECUs can be opted out. The default setting for the export is, however, to ignore only frames between simulated ECUs, but to receive all frames of the real ECUs. This allows a better integration into the test rig and CarMaker models:



Figure 12.13: IPG CarMaker/HIL popup - Settings

However, if you want to have a more realistic simulation in means of bus load, you can also generate the frames between simulated ECUs by disabling the *Ignore frames between simulated ECUs* checkbox.

The reception of FlexRay messages should always be done with the help of the receiver FIFO on the FlexCard (setting *Use receive FIFO*). Due to the limited number of message buffers on the Bosch ERay FlexRay controller, this check box should not be disabled.

Another check box called *Gateway Mode* can be selected for a rest bus simulation with Car-Maker as a FlexRay gateway between the real ECUs.

The name of the generated output files will be based on the name of the FlexConfig project and will be saved in the same directory (e.g. *Data/Misc*):

File	Description
<code><PRJ_NAME>.xml</code>	FIBEX file for import
<code><PRJ_NAME>.fpf</code>	FlexConfig project, based on imported FIBEX file
<code><PRJ_NAME>.info</code>	Main export file with frame and signal descriptions (<i>FlexRayParameters</i>)
<code><PRJ_NAME>0.chi</code> [<code><PRJ_NAME>1.chi</code>]	FlexRay controller configuration file(s). The number of files depends on the number of required controllers
<code><PRJ_NAME>.fccm</code>	Configuration of selected ECUs etc. for later reuse

The destination directory and the naming of the exported files can be changed by the *File-name* input field and the browse button.

When pressing the *Generate* button, the export step is executed and a confirmation dialog is displayed showing the names of all generated files:

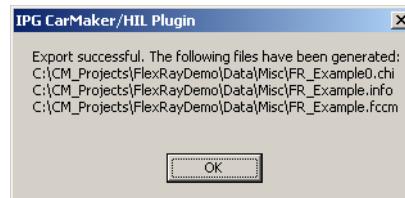


Figure 12.14: IPG CarMaker/HIL popup - Confirmation dialog

12.2.2 Export for Gateway Mode

Some application – when two or more real ECU are connected to the test rig – require to control and manipulate the data flow between real ECUs. In this case, a special setup is required, where single real ECUs are connected to different FlexRay communication controllers. This feature can be activated with the checkbox *Gateway Mode*.

In Gateway Mode, all Tx frames coming from virtual (simulated) ECUs have to be sent simultaneously on several FlexRay controllers to all real ECUs. On the other hand, Rx frames coming from a real ECU have to be routed immediately to all real ECUs, which are connected to other FlexRay controllers. This guarantees, that the bus traffic on all FlexRay busses is the same (up to some extent – mainly depending on race conditions in the dynamic segment).

Real ECUs and FlexRay Controllers

When activating the *Gateway Mode* check box, or when adding an ECU to the list of real ECUs (in gateway mode), the CarMaker/HIL plug-in needs to know, to which FlexRay controller the real ECUs are connected. However, at this point of time, it's not required to identify a FlexRay controller absolutely, e.g. by a dedicated FlexCard etc. The CarMaker/HIL plug-in just asks for a logical numbering and lets you also specify a prefix string, which can be later used as prefix for Data Dictionary entries.

Whenever the CarMaker/HIL plug-in needs to assign a real ECU to a FlexRay controller, you will see the following popup:

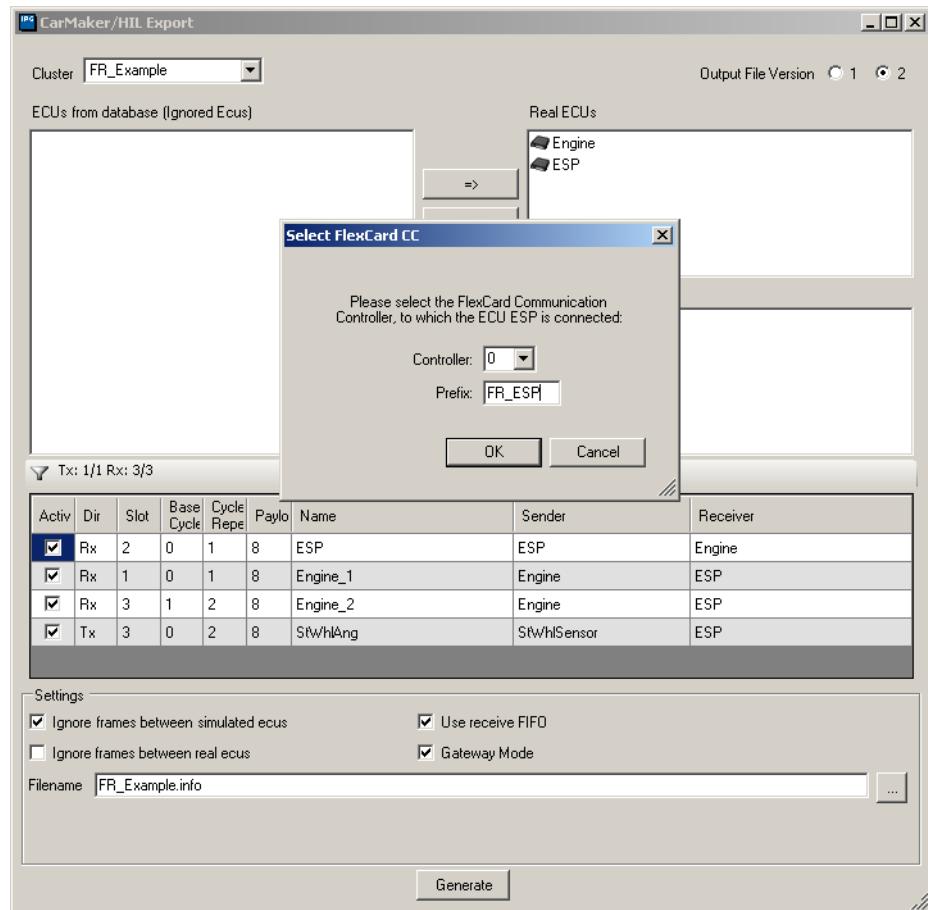


Figure 12.15: Gateway Mode – Select FlexRay controller

In order to change the FlexRay controller assignment of a real ECU, just double-click to the ECU in the list of real ECUs. The above dialog will then also appear, where you can select another FlexRay controller in the drop-down list, or change the prefix string.

Generation step

The naming of the generated output files follows the same scheme as with the standard, not gateway mode. However, for every physical FlexRay controller, an individual CHI configuration file is generated, as well.

12.3 Rest Bus Simulation in CarMaker/HIL

12.3.1 Integration of FlexRay in a CarMaker Project

The project template for CarMaker/HIL is already prepared for a FlexRay rest bus simulation. However, FlexRay is not active by default. It needs to be enabled in the Makefile, followed by a rebuild of the realtime application.

Makefile

On compilation, the preprocessor macro `WITH_FLEXRAY` is used to activate the FlexRay rest bus simulation function. In the Makefile of the project template, there is already a line which adds the macro `WITH_FLEXRAY` to the Makefile variable `DEF_CFLAGS`, but it is commented out:

```
### FlexRay
#DEF_CFLAGS += -DWITH_FLEXRAY
```

By activating the line, different code sections in the modules `IO.c` and `User.c` are enabled, which handle the initialization of the FlexCard and manage the FlexRay communication together with the rest bus simulation.

12.3.2 Initialization of FlexRay Framework

Basic Initialization

Basic initialization is done in the module `IO.c` in three steps:

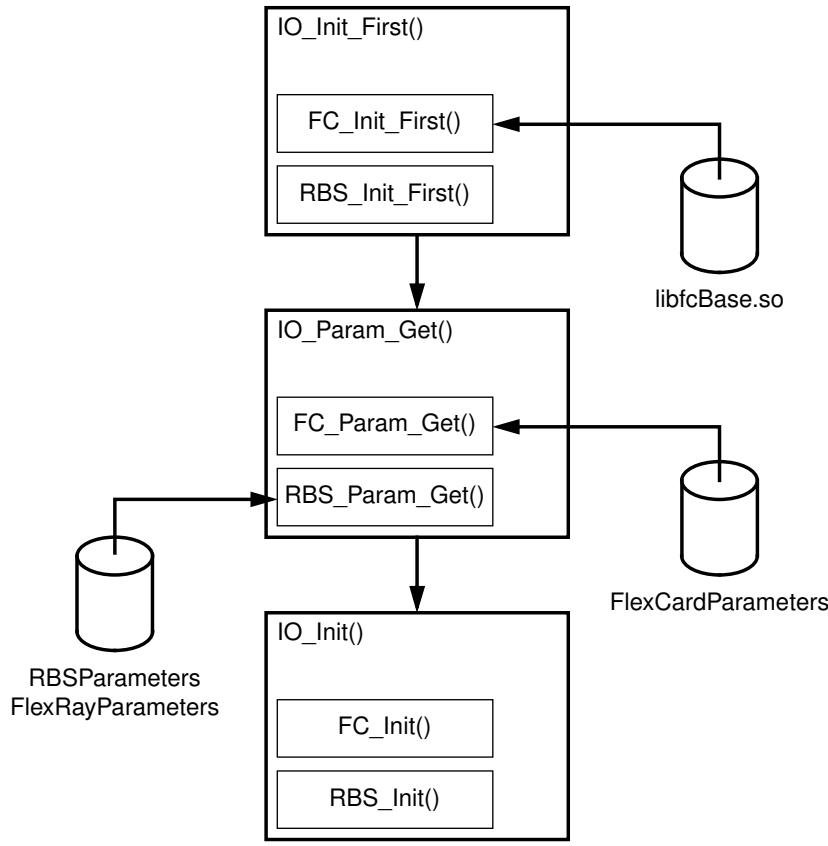


Figure 12.16: Basic Initialization

In `IO_Init_First()`, the function `FC_Init_First()` is called to load the FlexCard interface Library, which is used to access FlexRay hardware interfaces from Eberspächer Electronics. This is followed by the early initialization of the rest bus simulation by `RBS_Init_First()`.

The configuration is performed in `IO_Param_Get()`: `FC_Param_Get()` loads the FlexCard configuration (s. [section 12.5.2 'FlexCardParameters'](#)) and identifies all FlexRay interfaces which are used by the application. The function `RBS_Param_Get()` then reads the configuration for the rest bus simulation out of the `RBSParameters` file, which references a `FlexRayParameters` file for the bus configuration.

In `IO_Init()`, the functions `FC_Init()` and `RBS_Init()` are called in order to configure all used FlexRay interfaces and to prepare for start-up.

Data Dictionary Quantities

After the initialization of the FlexRay interfaces and the rest bus simulation, the function `RBS_DeclQuants()` is called in `User_DeclQuants()`. The purpose of this function is to provide access to all FlexRay signals as defined with the `FlexRayParameters` file.

By default, all generated Data Dictionary quantities follow the naming scheme below:

`<Prefix>.<ECUName>.<FrmName>.<SigName>`

Prefix	Prefix for identification of the rest bus simulation (default: FR)
ECUName	Name of the ECU, that transmits the frame
FrmName	Name of the FlexRay frame
SigName	Name of the FlexRay signal

Depending on the FlexRay configuration, the number of FlexRay signals may be very high. However, there are possibilities to limit the number of Data Dictionary quantities and to customize the naming scheme with settings in the `RBSParameters` file.

Completion of Initialization

The FlexRay initialization is completed in `IO_Init_Finalize()`:

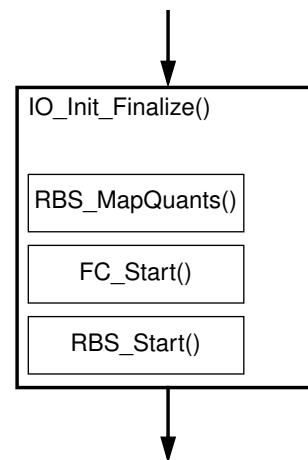


Figure 12.17: Completion of Initialization

First, mappings between CarMaker model quantities and FlexRay signals are set-up as defined in the `RBSParameters` file. This is performed by the function `RBS_MapQuants()`, which also manages special mappings like CRC or counter signals. By calling the functions `FC_Start()` and `RBS_Start()`, the FlexRay communication is started and the rest bus simulation is prepared for the cyclic management of FlexRay frames and signals.

12.3.3 Rest Bus Simulation

Input Processing

FlexRay frames are received and processed in `IO_In()` at the beginning of each simulation cycle:

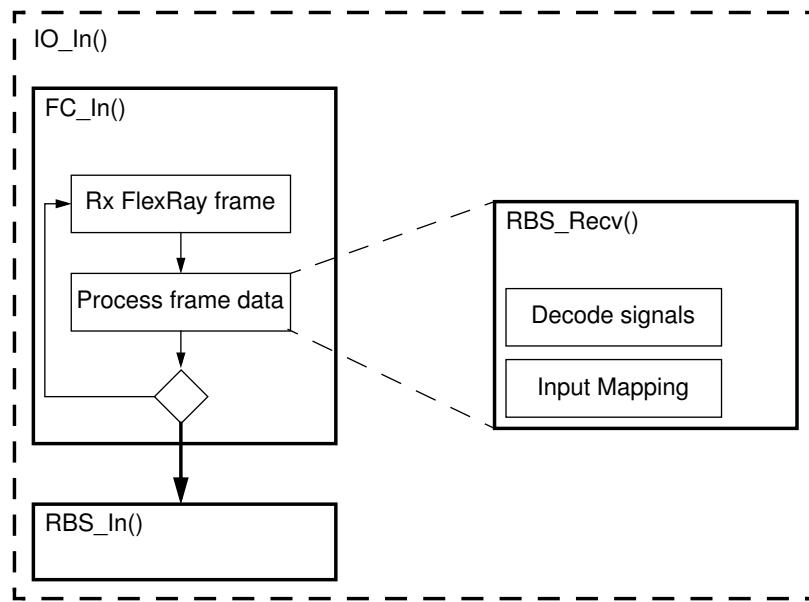


Figure 12.18: Input Processing

First, the function `FC_In()` reads all available FlexRay frames from the FlexRay interface(s). With each frame, a callback function from the rest bus simulation module is called to decode signal values and to perform input mappings. After all received FlexRay frames have been processed, `RBS_In()` is called to evaluate different input conditions.

Output Processing

For output of simulation results to the FlexRay bus, the functions `User_Out()` and `IO_Out()` are involved:

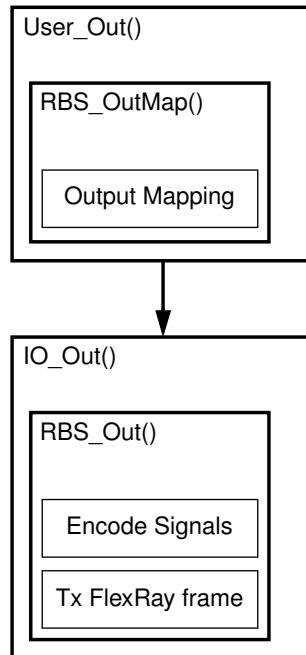


Figure 12.19: Output Processing

In `User_Out()` the function `RBS_OutMap()` is called, which performs the mapping between recently updated model quantities and the FlexRay I/O signal variables. At this point of time no signal value is encoded into the payload data of FlexRay frames, which allows dedicated signal values to be changed, e.g. for failure simulation.

At the end of the simulation cycle – in `IO_Out()` – `RBS_Out()` is called to encode signal values into the payload data of FlexRay frames and to send Tx frames to the hardware.

12.3.4 Customization

Of course, a rest bus simulation is not just sending and receiving FlexRay frames. Also the data bytes of the frames need to be filled with life. This means to set the values of signals according to the current situation and to react also on the state of received frames and signals.

While many signals can be mapped directly from and to quantities of the CarMaker models, others may be just constant.

There are also signals which have special meanings like check sums or alive counters. Or a data frame might require to be managed with low-leveled bit operations.

For CRC and rolling counter signals, there are already pre-defined calculation functions available, which can be used with standard mappings. With signals and frames, that cannot be handled with the default functionality, additional frame and signal hook functions can be registered. Supported types of hook functions are:

- Frame hook functions for low-level access to data bytes of received and sent frames (`RBS_HookFuncRx`, `RBS_HookFuncTx`)
- CRC hook functions for calculation of check sums over specific frames (`RBS_HookFuncCRC`)
- Hook functions for specific frames (`RBS_HookFuncFrame`)

- Hook functions for specific signals (`RBS_HookFuncSignal`)
- Mapping functions for CRC signals with non-standard algorithm (`RBS_MapFuncCRC`)
- Mapping functions for rolling counter signals with non-standard algorithm (`RBS_MapFuncRollCnt`)

Input Processing

On reception of a FlexRay frame it is first identified by the slot id and cycle multiplexing settings. Then, the payload data is decoded and processed:

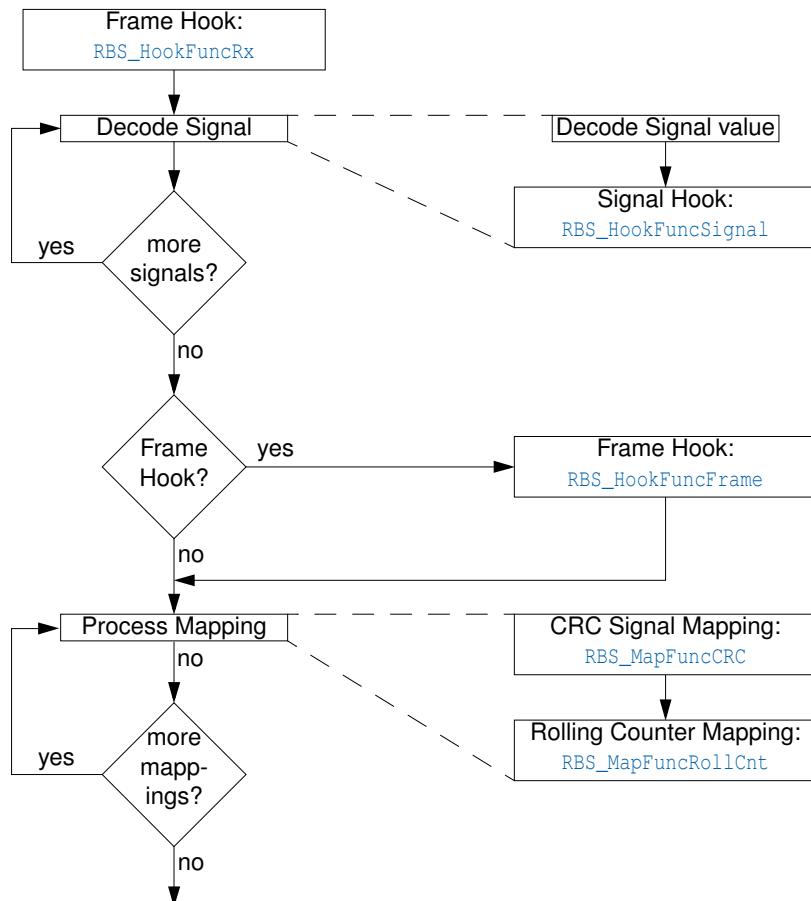


Figure 12.20: Input Processing

Output Processing

Whenever a FlexRay frame is to be sent within the current simulation cycle, the payload data needs to be updated with current values for all its signals:

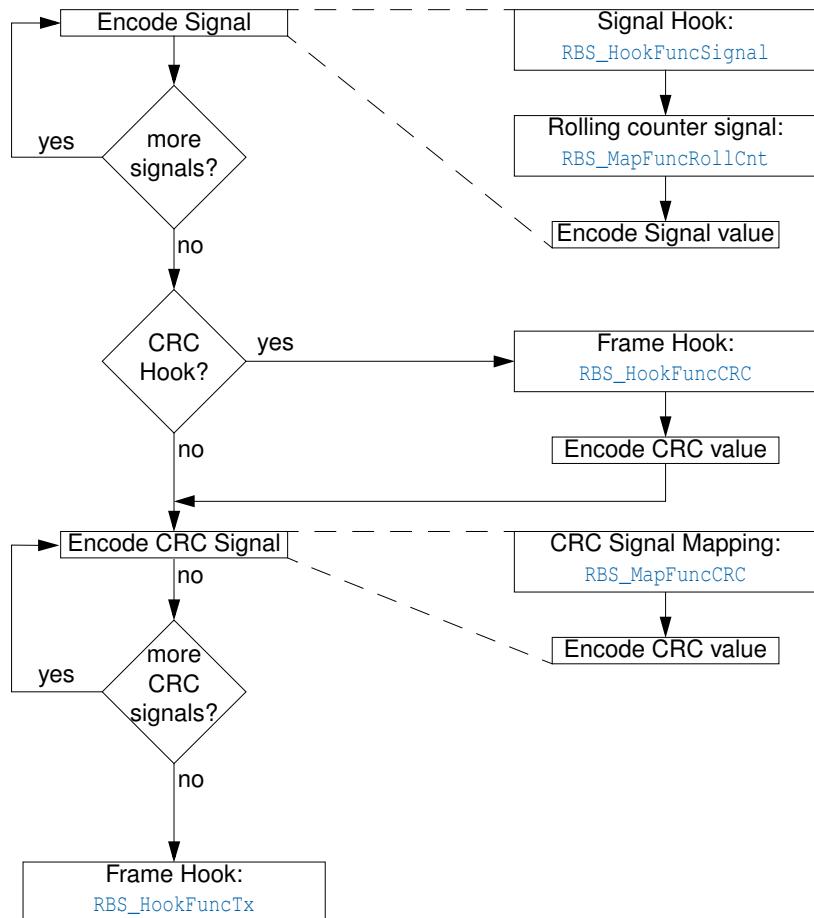


Figure 12.21: Output Processing

12.3.5 Data flow in Gateway Mode

The main benefit of the gateway mode is the possibility to control and manipulate the data flow and signal values to and from real ECUs separately. In order to have a realistic setup, some effort is necessary to keep the data flow on all FlexRay busses (connected on different FlexRay controllers) as much as possible the same. If someone uses external tools to produce bus traces on the different FlexRay busses, they should look the same.

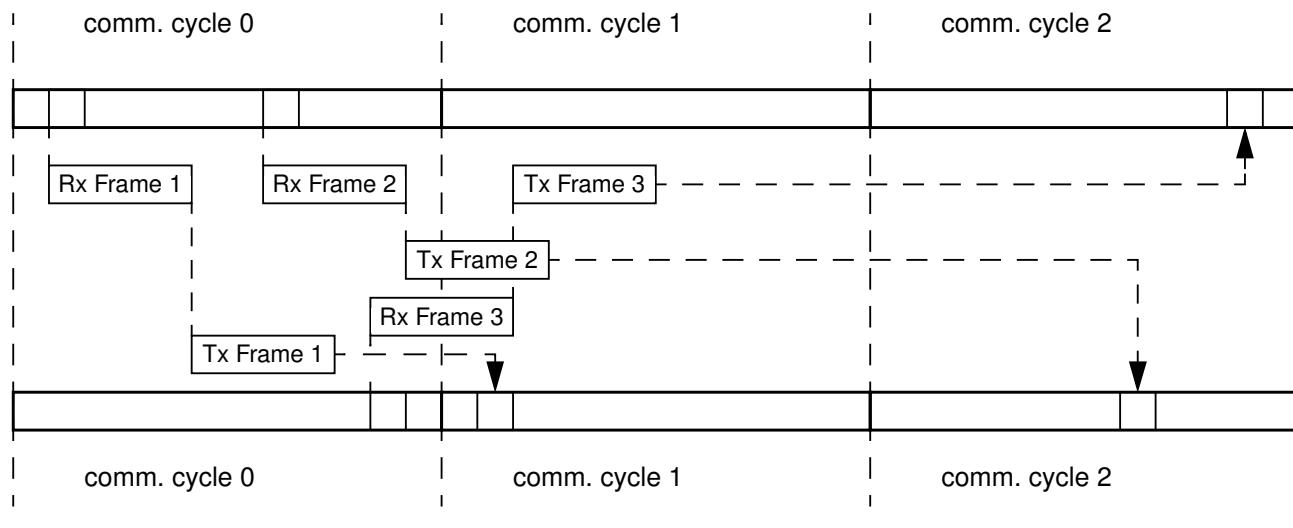
However, there are some effects and conditions that need to be considered and accepted. Splitting up a FlexRay bus into two (or more) partial busses, requires some synchronization and the user needs to know, that frames cannot be routed without delay.

Synchronization of FlexRay bus

According to the FlexRay specification, a complete bus cycle consists of 64 communication cycles. For each frame, there is a predefined condition, which defines exactly in which communication cycles (e.g. every second, fourth, etc.) within the complete bus cycle, it is possible to be sent. This condition is called the cycle multiplexing of a frame.

Synchronizing two FlexRay busses means, that on both busses, the FlexRay bus cycle will start and end always at exactly the same time, and will never drift away. Up to some extent, it may be desirable to have a constant time offset between the two busses (like e.g. 1 communication cycle).

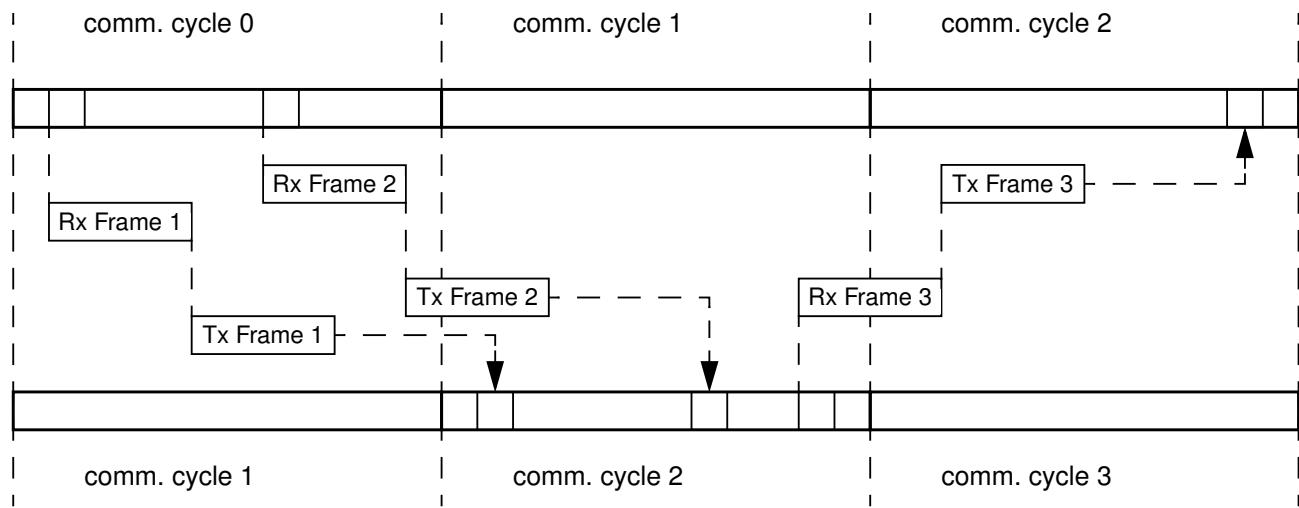
Supposed to be a frame is received from a real ECU, it needs to be sent immediately to the other bus. However, since reception, processing and transmission of the frame needs some time (even though it might be only some micro seconds), it normally cannot be transmitted within exactly the same communication cycle. For frames, which are not cycle multiplexed, they will be sent within the following communication cycle. All other frames, are additionally delayed, until the cycle number of the target bus meets the cycle multiplexing condition of the frame. In the worst case, this can be a total of 64 communication cycles later:



Frame 1: cycle multiplexing 0/1
Frame 2: cycle multiplexing 0/2
Frame 3: cycle multiplexing 0/2

Figure 12.22: Synchronization of FlexRay busses – without time offset

If you choose a constant time offset between the two busses, you can achieve a shorter delay in both directions. However, the effective delay always depends on the cycle multiplexing of a frame:



Frame 1: cycle multiplexing 0/1

Frame 2: cycle multiplexing 0/2

Frame 3: cycle multiplexing 0/2

Figure 12.23: Synchronization of FlexRay busses – constant time offset

Processing of Frames and Signal values

Whenever a flexray frame is received from a real ECU, it is immediately processed, the received signals are decoded and mapped as inputs to the model. The frame is then marked for routing. Optionally, all signal values are copied to a second set of signals, representing the output side of the gateway.

So far, the frame is prepared for transmission to the other FlexRay busses and will be sent at the end of the same simulation step. With this short, but not relevant delay, it is possible to pass a signal to the model, perform some calculations, and – if desired – manipulate its value before forwarding the frame to other real ECUs.

The data flow of a signal when routing a frame between two FlexRay busses is shown below:

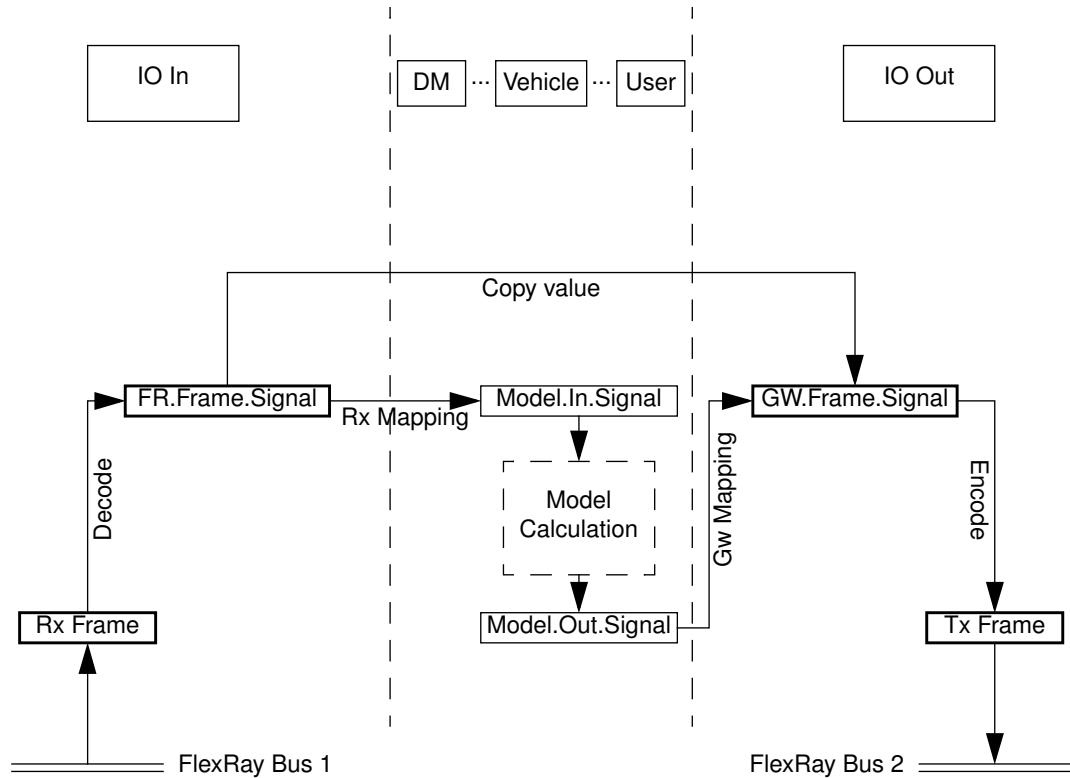


Figure 12.24: Data flow of Signal through FlexRay Gateway

Routing of unknown frames

Besides those FlexRay frames, which are relevant for and known by the rest bus simulation module in CarMaker/HIL, there are usually also other data frames on the FlexRay busses, which are ignored. Typical frames are e.g. diagnostic frames, which are only used to communicate with external diagnostics tools or bus monitors. However, these frames need to be routed, too.

Whenever a FlexRay frame is received which can not be identified by the rest bus simulation, it is transmitted immediately and without any modification on all other FlexRay busses. Since there is nothing known about the data frame – especially nothing about its cycle multiplexing condition – some effort is made in order to guess a cycle multiplexing condition, which can not collide with eventual known FlexRay frames, but still guarantees a transmission as soon as possible.

This feature can be switched off.

12.4 Functional description

12.4.1 FlexCard Function Interface

Configuration / Initialization

FC_Init_First ()

```
void FC_Init_First (void);
```

Description

First (early) initialization of the FlexCard communication library. The FlexCard library *libfcBase.so* is loaded and internal function pointers and data structures are initialized for communication with the available FlexCards.

This function has to be called prior to all other `FC_*`() functions.

FC_ShowCards ()

```
void FC_ShowCards (void);
```

Description

Collects information about all available FlexCards and prints a list of hardware and software information into the log file. This function is useful for debugging.

FC_Param_Get ()

```
int FC_Param_Get (struct tInfos *inf, const char *pfx);
```

Description

Reads the FlexCard configuration from the given info file handle `inf`. If a prefix is passed, too (`pfx != NULL`), then “`pfx.`” will be prepended to all keys which are read from the given info file handle `inf`.

First, `FC_Param_Get ()` tries to read the key `FlexCardParameters`. If `inf` is not `NULL` and the key exists, then its value is interpreted as the name of another info file with the FileIdent `FlexCardParameters`. All configuration parameters will then be read from this file. If `inf` is `NULL`, then a file with name `FlexCardParameters` will be searched. The format of the `FlexCardParameters` file is described in [section 12.5.2 ‘FlexCardParameters’](#).

If `inf` is not `NULL` and if the key `FlexCardParameters` does not exist, then the FlexCard configuration will be read directly from the info file handle `inf` (eventually prefixed with “`pfx.`”).

Return Value

- 0, if all FlexCard parameters have been successfully read.
- Otherwise: <0. Additional information about the reason is written into the session log.

FC_Init ()

```
int FC_Init (void);
```

Description

Opens all FlexCards which are configured by [FC_Param_Get \(\)](#) and the [FlexCardParameters](#) file. The FlexCards are prepared for initialization of the communication controllers. FlexCards that are not referenced by the [FlexCardParameters](#) file will be left untouched.

Return Value

- 0, if all FlexCards have been successfully opened.
- Otherwise: <0. Additional information about the reason is written into the session log.

FC_Cleanup ()

```
void FC_Cleanup (void);
```

Description

Closes all handles to configured FlexCards and unloads the library `libfcBase.so`. Before using any FlexCard again, the module needs to be reinitialized with [FC_Init_First \(\)](#).

Start-up / Stop of FlexRay Communication

FC_Start ()

```
int FC_Start (void);
```

Description

Starts the FlexRay communication on all communication controllers of all FlexCards. The communication controllers which are configured for operation (e.g. with CHI data by [FC_ConfigCC \(\)](#)) are put into monitoring mode (`fcMonitoringNormal`) by the FlexCard API function `fcbFRMonitoringStart()`. For unprepared FlexRay interfaces this function does not have any effect.

Return Value

- 0, if the communication could be successfully started on all interfaces.
- Otherwise: <0. Additional information about the reason is written into the session log.

FC_Stop ()

```
int FC_Stop (void);
```

Description

Stops the FlexRay communication on all communication controllers of all FlexCards. The communication controllers are put into configuration mode by the FlexCard API function `fcbFRMonitoringStop()`.

Handle Reception of FlexRay Frames

FC_In ()

```
void FC_In (unsigned CycleNo);
```

Description

Scans all active FlexCards and communication controllers for received FlexRay packets and processes the data. The packets are also passed to all callback functions that are registered either by [FC_RecvRegister \(\)](#) or by [FC_RecvRegister2 \(\)](#).

Parameters

- `CycleNo` is the current simulation cycle of the realtime application

Low-level access to Communication Controllers

Some operations like configuration with CHI data, registering receive callbacks or transmission of FlexRay frames require direct access to communication controllers. For this purpose, some low-level functions allow to address FlexCards and their communication controllers with indexes as defined by the FlexCard configuration in the [FlexCardParameters](#) file.

FC_ConfigCC ()

```
int FC_ConfigCC (int FC, int CC, char *CHI_data);
```

Description

Loads the CHI data from the null-terminated character string `CHI_data` to the communication controller `CC` on FlexCard `FC` and prepares it for start-up. The communication controller must be in configuration state, otherwise the function will fail.

Return Value

- 0, if the communication controller could be successfully initialized.
- Otherwise: <0. Additional information about the reason is written into the session log.

FC_RecvRegister ()

```
int FC_RecvRegister (int FC, int CC, tFC_Callback CB, void *Arg);
```

Description

Registers a callback function for the reception of FlexRay packets on communication controller `CC` of FlexCard `FC`. The callback function `CB` is of type `tFC_Callback` and has the following function prototype:

```
int FC_Callback (struct tFlexCard *, struct tCC *, struct tFC_FrameBuf *, void *, void *);
```

On reception of a FlexRay packet, the registered callback function will be called and pointers to the FlexCard configuration struct (`struct tFlexCard *`), the communication controller struct (`struct tCC *`) and the frame buffer struct (`struct tFC_FrameBuf *`) will be passed. The data structs are defined in [Listing 12.1](#), [Listing 12.2](#) and [Listing 12.3](#):

Listing 12.1: FlexCard configuration struct `tFlexCard`

```

1:  typedef struct tFlexCard {
2:      fcHandle hFlexCard;
3:      int iFlexCard;
4:      tCC CC[MAX_FLEX_CCS];
5:      int NumCC;
6:      unsigned TimeStamp;
7:      double TS_Time;
8:      float T_Receive;
9:      float T_ReceiveMin;
10:     float T_ReceiveMax;
11:     float T_MemFree;
12:     float T_RxProcess;
13:     float T_Schedule;
14:     float T_Send;
15:     float T_SendMin;
16:     float T_SendMax;
17:     float T_Update;
18:     float T_UpdateMin;
19:     float T_UpdateMax;
20:     int Num_Packets;
21:     unsigned NumPacketsNotification;
22:     unsigned NumPacketsTrigger;
23:     unsigned NumPacketsTriggerEx;
24:     float RxRate;
25:     float TxRate;
26:     unsigned LogErrCycleNo;
27: } tFlexCard;
```

Listing 12.2: Communication controller struct `tCC`

```

1:  typedef struct tCC {
2:      short FC_Id;
3:      char CC_Id;
4:      char RBS_Id;
5:      char *Name;
6:      char *CHI;
7:      char *CHI_data;
8:      int BusState;
9:      int CC_State;
10:     int SelfSynchId;
11:     unsigned Num_RxFrame;
12:     unsigned Num_TxFrame;
13:     tFC_FrameBuf **RxFrameBuf;
14:     int nRxFrameBuf;
15:     int mRxFrameBuf;
16:     tFC_FrameBuf **TxFrameBuf;
17:     int nTxFrameBuf;
18:     int mTxFrameBuf;
19:     short TxFloatIdxStat;
20:     short TxFloatIdxDyn;
21:     fcMsgBufCfg *TxFloatStatic;
22:     fcMsgBufCfg *TxFloatDynamic;
23:     unsigned TxDictDefRaw :1;
24:     unsigned TxAckEnable :1;
25:     unsigned TxAckShowNF :1;
26:     unsigned TxAckShowPl :1;
27:     unsigned RxDictDefRaw :1;
28:     unsigned RxShowNF :1;
29:     unsigned AutoRestart :1;
30:     unsigned Termination :1;
```

Listing 12.2: Communication controller struct tCC

```
31:     unsigned          TermReset      :1;
32:     unsigned          ChMask        :2;
33:     unsigned          CycleNo;
34:     unsigned          CycleStartTS;
35:     double           TMonStart;
36:     double           TPauseRestart;
37:     double           TSchedule;
38:     double           pdMicrotick;
39:     double           pdMacrotick;
40:     double           gdStaticSlot;
41:     double           gdMinislot;
42:     double           gdCycle;
43:     unsigned          gNumStaticSlots;
44:     unsigned          gPayloadLengthStatic;
45:     unsigned          gNumMiniSlots;
46:     struct tFC_CBdata *RxCB;
47:     int              nRxCB;
48:     unsigned          Num_Restarts;
49:     unsigned          NumPacketsInfo;
50:     unsigned          NumPacketsError;
51:     unsigned          NumPacketsStatus;
52:     unsigned          NumPacketsTxAck;
53:     unsigned          NumPacketsNMVector;
54:     unsigned          NumPacketsCAN;
55:     unsigned          NumPacketsCANError;
56:     struct {
57:         unsigned Type;
58:         unsigned Slot;
59:         unsigned CycleNo;
60:         unsigned Channel;
61:         unsigned FC_Overflow;
62:         unsigned SyncFrames;
63:         unsigned ClockCorrection;
64:         unsigned Parity;
65:         unsigned FIFO_OVERRUN;
66:         unsigned FIFO_EMPTY;
67:         unsigned InBuf_Illegal;
68:         unsigned OutBuf_Illegal;
69:         unsigned Syntax;
70:         unsigned SyntaxSW;
71:         unsigned SyntaxNIT;
72:         unsigned Content;
73:         unsigned SlotBoundViolation;
74:         unsigned TransAcrossBound;
75:         unsigned TransConflictSW;
76:         unsigned LastestTransViolation;
77:         unsigned SlotBoundViolationSW;
78:         unsigned SlotBoundViolationNIT;
79:     } Error;
80: } tCC;
```

Listing 12.3: Frame buffer struct tFC_FrameBuf

```

1:  typedef struct tFC_FrameBuf {
2:      unsigned     FrameId      :12;
3:      unsigned     CycleRep     :7;
4:      unsigned     CyclePos     :6;
5:      unsigned     Cyclic       :1;
6:      unsigned     FIFO         :1;
7:      unsigned     ChMask       :2;
8:      unsigned     PayloadLength:7;
9:      unsigned     DataReady    :1;
10:     unsigned    inStaticSeg :1;
11:     unsigned char CycleCount;
12:     unsigned char NF;
13:     short        BufferIdx;
14:     unsigned     MT_Offset;
15:     unsigned     TxCount;
16:     double       tTxNext;
17:     unsigned     CycleCalc;
18:     unsigned     CycleSend;
19:     unsigned short Payload[127];
20:     unsigned short PayloadAck[127];
21:     unsigned char PayloadLengthAck;
22: } tFC_FrameBuf;

```

The last argument passed to the callback function of type `tFC_Callback` is a user defined pointer as passed with the argument `Arg`. The second last argument is a pointer to the FlexRay packet of type `fcFlexRayFrame`, as defined by the FlexCard API.

Return Value

If the callback function could be successfully registered, a value ≥ 0 is returned. This handle can be used later to unregister the callback function again with [FC_RecvUnregister \(\)](#).

On error, -1 is returned.

FC_RecvRegister2 ()

```
int FC_RecvRegister2 (int FC, int CC, tFC_Callback2 CB, void *Arg);
```

Description

Registers a callback function for the reception of FlexRay packets on communication controller `CC` of FlexCard `FC`. The callback function `CB` is of type `tFC_Callback2` and has the following function prototype:

```
FC_Callback2 (struct tFlexCard *, struct tCC *, struct tFC_FrameBuf *, int Ch, void *, void *);
```

On reception of a FlexRay packet, the registered callback function will be called and pointers to the FlexCard configuration struct (`struct tFlexCard *`), the communication controller struct (`struct tCC *`) and the frame buffer struct (`struct tFC_FrameBuf *`) will be passed. The data structs are defined in [Listing 12.1](#), [Listing 12.2](#) and [Listing 12.3](#).

The fourth argument passed to the callback function is the parameter `Ch`, indicating the channel on which the frame was received. Possible values for `Ch` are:

0	No channel (not possible on reception)
1	Frame was received on channel A
2	Frame was received on channel B
3	Both channels (not used for reception)

The remaining arguments are the same as for the function [FC_RecvRegister \(\)](#).

FC_RecvUnregister ()

```
void FC_RecvUnregister (int FC, int CC, int CB_Id);
```

Description

Unregisters a receive callback function, which was previously registered with [FC_RecvRegister \(\)](#), or [FC_RecvRegister2 \(\)](#).

Return Value

If the callback function could be successfully registered, a value ≥ 0 is returned. This handle can be used later to unregister the callback function again with [FC_RecvUnregister \(\)](#).

On error, -1 is returned.

FC_FrameReadyToSend ()

```
int FC_FrameReadyToSend (int FC, int CC, tFC_FrameBuf *Frame, double t);
```

Description

Checks if the given frame buffer `Frame` is ready to be sent and should be updated with current payload data. It is checked based on the given timestamp of the last transmission, the slot number as well as the cycle time settings, compared with the current time `t`, if the frame is to be sent within the same simulation cycle.

Return Value

- !=0, if the frame is going to be sent within the current simulation cycle.
- 0, if the frame is not yet ready to be sent.

FC_FrameDataReady ()

```
int FC_FrameDataReady (int FC, int CC, tFC_FrameBuf *Frame);
```

Description

If the frame buffer `Frame` was updated with current data and was not yet sent, the function returns 1 (true). Otherwise, 0 (false) is returned.

FC_FrameSetDataRdy ()

```
void FC_FrameSetDataRdy (int FC, int CC, tFC_FrameBuf *Frame, int Ready);
```

Description

Allows to mark the frame buffer `Frame` as ready to send. If set to 1 (true), the frame will be sent as soon as possible (regarding the slot number and cycle time settings).

FC_FrameSend ()

```
int FC_FrameSend (int FC, int CC, tFC_FrameBuf *Frame);
```

Description

Takes a frame buffer `Frame` and transmits it on the specified communication controller.

Return Value

- 0, if the frame was sent successfully.
- Otherwise: <0.

FC_Send ()

```
int FC_Send (int FC, int CC, unsigned FrameId, int CycleRep, int CyclePos,  
            unsigned short *Payload, int PayloadLength);
```

Description

Allows to send a data frame with (almost) arbitrary `FrameId`, cycle multiplexing (`CycleRep`, `CyclePos`) and `PayloadLength`.

Return Value

- 0, if the frame was sent successfully.
- Otherwise: <0.

12.4.2 RBS Function Interface

Configuration / Initialization

RBS_Init_First ()

```
void RBS_Init_First (void);
```

Description

First (early) initialization of the rest bus simulation module. This function needs to be called prior to any other `RBS_*`() function.

RBS_Param_Get ()

```
int RBS_Param_Get (struct tInfos *inf, const char *pfx);
```

Description

Reads the configuration of the FlexRay rest bus simulation from the given info file handle `inf`. If a prefix is passed, too (`pfx != NULL`), then “`pfx.`” will be prepended to all keys which are read from the given info file handle `inf`.

First, `RBS_Param_Get()` tries to read the key `RBSParameters`. If `inf` is not NULL and the key exists, then its value is interpreted as the name of another info file with the Fileident `RBSParameters`. All configuration parameters will then be read from this file. If `inf` is NULL, then a file with name `RBSParameters` will be searched. The format of the `RBSParameters` file is described in [section 12.5.3 'RBSParameters'](#).

If `inf` is not NULL and if the key `RBSParameters` does not exist, then the RBS configuration will be read directly from the info file handle `inf` (eventually prefixed with "pfx.").

Return Value

- 0, if all RBS parameters have been successfully read.
- Otherwise: <0. Additional information about the reason is written into the session log.

RBS_Init ()

```
int RBS_Init (void);
```

Description

Configures all FlexCards and communication controllers that are assigned to rest bus simulations with an eventually specified CHI file and prepares them for start-up. Communication controllers, that are assigned to a rest bus simulation with automatic configuration disabled (s. [page 482](#)), will be left untouched.

RBS_DeclQuants ()

```
void RBS_DeclQuants (void);
```

Description

Defines Data Dictionary entries for frames and signals that are handled with the rest bus simulations. Depending on different settings (s. [section 12.5.3 'RBSParameters'](#)), this can be quantities for mapped signals, unmapped signals, or both. If set, quantities for raw values and for signal states can be defined, too.

RBS_MapQuants ()

```
int RBS_MapQuants (void);
```

Description

Creates and sets up the mappings for all FlexRay signals and frames, as defined in the RBS parameters file `RBSParameters`.

RBS_Register_HookFunc ()

```
int RBS_Register_HookFunc (char *Name, tRBS_HookFuncType Type, tRBS_HookFunc Func);
```

Description

Allows to register a hook function for FlexRay signals and frames. Hook functions can be registered with a name and are then available for activation in the RBS parameters file. Following types of hook functions are supported:

RBS_HookFuncRx (1)	Global hook function for received FlexRay frames
RBS_HookFuncTx (2)	Global hook function for transmission of FlexRay frames
RBS_HookFuncCRC (3)	CRC hook function, assigned to selected Tx frames
RBS_HookFuncFrame (4)	Frame hook function, assigned to selected Rx frames
RBS_HookFuncSignal (5)	Signal hook function, assigned to selected signals

A hook function is of type `tRBS_HookFunc` and has the following function prototype:

```
int RBS_HookFunc (struct tFlexCard *, struct tRBS *, struct tCC *, struct tRBS_SigFrame *,
tRBS_HookArgs *);
```

Depending on the selected Type, the hook function will be called in different situations:

RBS_HookFuncRx	Called with every received FlexRay frame: • right after reception • before decoding of payload data
RBS_HookFuncTx	Called with every transmitted FlexRay frame: • after all signals have been encoded into the payload data • immediately before sending the frame to the FlexCard
RBS_HookFuncCRC	Called on transmission of selected FlexRay frame: • after normal signals have been encoded into the payload • if connected with a signal, the signal value is updated with the calculated CRC and encoded into the payload data • before output signal mappings of type CRC are processed
RBS_HookFuncFrame	Called on reception of selected FlexRay frame: • after all signals have been decoded • before input signal mappings are processed
RBS_HookFuncSignal	Called on transmission and reception of selected FlexRay frame: • for Tx-signal, before encoding value of selected signal • for Rx-signal, after decoding value of selected signal

Whenever a registered callback function is called, pointers to the FlexCard configuration struct (`struct tFlexCard *`), the rest bus simulation struct (`struct tRBS *`) and the communication controller struct (`struct tCC *`) will be passed. The data structs are defined in [Listing 12.1](#), [Listing 12.2](#) and [Listing 12.4](#):

Listing 12.4: Rest Bus Simulation struct `tRBS`

```
1:  typedef struct tRBS {
2:      char          *Cfg;
3:      struct tInfos *Inf;
4:      char          *VarInfo;
5:      struct {
6:          struct tCC   *CC;
7:          char          *CHI;
8:          char          *CHI_data;
9:          char          *Prefix;
10:         int           CB_Id;
11:         } MapCC[MAX_FLEX_CCS];
12:         int           NumCC;
13:         char          *Prefix;
14:         char          *GwPrefix;
15:         int           Flags;
16:         int           TxInitVal;
17:         char          SigStates[RBS_SState_nStates];
18:         tRBS_SigFrame **TxFrame;
19:         int           nTxFrame;
20:         tRBS_HookFunc TxHookFunc;
21:         tRBS_HookArgs TxHookArgs;
22:         tRBS_SigFrame **RxFrame;
23:         int           nRxFrame;
24:         tRBS_HookFunc RxHookFunc;
25:         tRBS_HookArgs RxHookArgs;
26:         tRBS_SigMap   **TxMap;
27:         int           nTxMap;
28:         tRBS_SigMap   **RxMap;
29:         int           nRxMap;
30:         tRBS_SigMap   **GwMap;
31:         int           nGwMap;
32:         tRBS_SigVar   *SigVar;
33:         int           nSigVar;
34:     } tRBS;
```

With the fourth argument a pointer to the selected signal frame is passed (`struct tRBS_SigFrame`), which is defined in [Listing 12.5](#):

[Listing 12.5: Signal frame struct tRBS_SigFrame](#)

```

1:  typedef struct tRBS_SigFrame {
2:      char             *Name;
3:      unsigned         FrameId     :12;
4:      unsigned         CycleRep    :7;
5:      unsigned         CyclePos    :6;
6:      unsigned         Cyclic      :1;
7:      unsigned         ChMask      :2;
8:      unsigned         Length      :7;
9:      unsigned         ByteLength  :8;
10:     char            *GRName;
11:     char            GwRoute;
12:     tRBS_CC_BufId   ActIdx;
13:     tRBS_CC_BufId   **RecvIdx;
14:     tRBS_CC_BufId   **SendIdx;
15:     char            *Receiver;
16:     char            *Sender;
17:     tRBS_Signal     **Signal;
18:     tRBS_Signal     **GwSignal;
19:     int              nSignal;
20:     double           tTxLast;
21:     char            *CTName;
22:     int              CycleTime;
23:     int              CycleTimeStd;
24:     int              CycleTimeMin;
25:     int              CycleTimeMax;
26:     struct tRBS_Trigger *Trigger;
27:     union {
28:         tRBS_HookFunc CRCFunc;
29:         tRBS_HookFunc HookFunc;
30:     };
31:     union {
32:         tRBS_HookArgs CRCArgs;
33:         tRBS_HookArgs HookArgs;
34:     };
35: } tRBS_SigFrame;
```

The last argument passed to the hook function is of type `tRBS_HookArgs` (s. [Listing 12.6](#)). The struct contains a pointer to the selected signal (NULL, if no signal) and up to four numeric arguments of type `int`, as defined in the RBS parameters file.

[Listing 12.6: Hook Function Arguments struct tRBS_HookArgs](#)

```

1: #define RBS_MAX_HOOK_FUNC_ARGS 4
2: typedef struct tRBS_HookArgs {
3:     struct tRBS_Signal *Signal;
4:     int                ArgC;
5:     int                ArgV[RBS_MAX_HOOK_FUNC_ARGS];
6: } tRBS_HookArgs;
```

Return value of Hook Function

For hook functions of type `RBS_HookFuncRx`:

- <0: received frame is discarded, no signals are decoded.
- >=0: frame and signals are processed.

For hook functions of type `RBS_HookFuncTx`:

- <0: frame is discarded, i.e. frame will not be sent.

- ≥ 0 : frame is sent.

For hook functions of type *RBS_HookFuncCRC* and if the CRC function is connected to a signal:

- $= 0$: the selected signal will be encoded into the payload data.
- $\neq 0$: the calculated CRC value will not be encoded.

For all other types, the return value does not have any effect.

RBS_Register_MapFunc()

```
int RBS_Register_MapFunc (char *Name, tRBS_MapFuncType Type, tRBS_MapFunc Func);
```

Description

Registers a user defined function for rolling counter or CRC signals, which can be referenced in *RollCnt* or *CRC* signal mappings, just by the registered name *Name*. Following types of mapping functions are supported:

RBS_MapFuncCRC (1)	Mapping function for calculation of CRC signals
RBS_MapFuncRollCnt (2)	Mapping function for calculation of Rolling Counter signals

A mapping function is of type *tRBS_MapFunc* and has the following function prototype:

```
int RBS_MapFunc (unsigned char *Data, int FLen, int Argc, int *Argv);
```

Whenever a map function is called, a pointer to the payload data of the FlexRay frame is passed with the argument *Data*, whereas the second argument *FLen* gives the length of the payload data in bytes. The fourth argument *Argv* is a pointer to an integer array with *Argc* elements.

For functions of type *RBS_MapFuncCRC* up to 6 integer values can be passed, as defined with the signal mapping in the RBS parameters file.

For functions of type *RBS_MapFuncRollCnt* the first element in the array will be the old value of the counter. Additional up to 5 integer values can be passed, if defined with the signal mapping in the RBS parameters file.

Return value of Map Function

For mapped Rx-signals, the return value of a map function reflects the new or rather the expected CRC or Counter value. In the next step, this calculated value is compared to the received value of the mapped signal.

According to the result of comparison the signal state is set. Supported signal states are:

RBS_SState_None	no state, unknown (default: 0)
RBS_SState_Valid	signal is valid (default: 1)
RBS_SState_Invalid	signal value is invalid (default: 2)
RBS_SState_Unavailable	signal is not available (default: 3)
RBS_SState_Undefined	signal is not defined (default: 4)
RBS_SState_Error	signal is in error state (default: 5)

For Tx-signals, the new signal value is given by return. In contrast to Rx-mode the state setting is independent of the return. It defines whether the mapping is performed or not.

RBS_Cleanup ()

```
void RBS_Cleanup (void);
```

Description

Terminates the rest bus simulation and frees allocated memory. Before using any `RBS_*`() function again, the module needs to be reinitialized with [RBS_Init_First\(\)](#).

Start / Stop Rest Bus Simulation

RBS_Start ()

```
int RBS_Start (void);
```

Description

Starts-up all FlexRay communication controllers that are assigned to rest bus simulations and begins with the reception and transmission of FlexRay frames.

RBS_Stop ()

```
int RBS_Stop (void);
```

Description

Stops the reception and transmission of FlexRay frames and stops the communication on all FlexRay communication controllers, that are assigned to rest bus simulations,

Cyclic Calculations (realtime)

RBS_In ()

```
void RBS_In (unsigned CycleNo);
```

Description

Processes all received FlexRay frames. This function needs to be called at the beginning of each simulation step of the realtime application, i.e. in `IO_In()`. Whenever a frame is received, the payload data is analyzed and all signals are updated. If there are signal mappings bound to that frame, they are processed, too.

RBS_OutMap ()

```
int RBS_OutMap (unsigned CycleNo);
```

Description

After the calculation step of the model but before I/O output, this function needs to be called in each simulation step to update all signal variables with new values – either from mapped model variables or mapped functions. Call this function at the end of `User_Calc()` before calling `IO_Out()`.

After `RBS_OutMap()`, all signal variables are updated, but not yet encoded into payload data of Tx-frames. Hence, manipulations on the signal values can still take place.

RBS_Out ()

```
void RBS_Out (unsigned CycleNo);
```

Description

This function manages the transmission of FlexRay frames and needs to be called at the end of each simulation step, i.e. in `IO_Out()`.

12.5 Configuration Files

12.5.1 FlexRayParameters

The main file of the generated output from the CarMaker/HIL plug-in is an Info file with the Fileident *CarMaker-FlexRayParameters*.

Header information

Fileident = CarMaker-FlexRayParameters Ver

Identifies the info file as FlexRay parameters file of version *Ver*. Currently supported values for *Ver* are 1 and 2.

Description:**Description of file type and contents**

Short description about the type of info file and its intended use.

Example**Description:**

Generated with FlexConfig CarMaker plugin Version 0.0.0.11 for:
CarMaker/HIL
FlexRay Rest Bus Simulation

Created = %Y/%m/%d %H:%M:%S user@host

Creation information about when the file was generated, the *user* and the *host* name.
The format of the date and time stamp is as follows:

%Y	Four digit calendar year of the current date
%m	Two digit month of the year with leading zero (01 .. 12)
%d	Two digit day of the month with leading zero (01 .. 31)
%H	Two digit hour of the current time (00 .. 23)
%M	Two digit minute of the current hour (00 .. 59)
%S	Two digit second of the current minute (00 .. 59)

FlexConfigProject = Path

Path to the FlexConfig project file, that was used as base for the export.

Configuration parameters

GatewayMode = bool

Defines, whether the configuration describes a FlexRay gateway, or not. The default value for this key is 0 (false).

CC.<nCC>.Config. ECU = Name[...]

Lists the names of all real ECUs, which are connected to this communication controller. This parameter is only used, if *GatewayMode* is active.

CC.<nCC>.Config. Prefix = Prefix

Defines a prefix for data dictionary quantities, that are created for frames and signals, and which are only sent on this communication controller. These are usually routed frames, that are received from real ECUs.

This parameter is only used, if *GatewayMode* is active.

CC.<nCC>.Config. CHI_FileName = Path

Defines the *Path* to a configuration file for the FlexRay communication controller on the FlexCard. Depending on the complexity of the rest bus simulation, the configuration occupies one or more communication controllers.

The communication controllers are numbered consecutively (beginning with zero) and pre-fixed with *CC.<nCC>*.

Each controller needs its own CHI file defined by *Path* and may also comprise specific FlexRay frames and signals.

CC.<nCC>.Config. SelfSyncStartupId = Slot_Id

If there is no coldstart node among the configuration of real ECUs, the self-sync capability of the FlexCard can be used. This key defines the *Slot_Id*, which is used for the self-sync.

FlexRay frames

The description of the following keys is only valid for configurations where *GatewayMode* is not active. For frame descriptions in gateway mode, refer to section [FlexRay frames \(Gateway Mode\)](#).

[CC.<nCC>.]TxFrame.<nTx> =	Name Slot Ch Base Rep Len BLen Idx [Sender]
[CC.<nCC>.]RxFrame.<nRx> =	Name Slot Ch Base Rep Len BLen Idx [Sender]

Defines a Tx-/Rx-frame. The frames are numbered consecutively, starting with zero. Both frame counters – `nTx` and `nRx` – start with zero and are incremented independently for every new frame.

Each frame is defined by the following parameters:

Name	Name of the frame
Slot	Slot Id of the frame. Possible values are in the range of 1 and 2047
Ch	Channel, on which the frame is sent or received. Possible values are “A” for channel A, “B” for channel B, or “AB” for both channels
Base	Specifies the base cycle in which the frame is to be sent. If no cycle multiplexing is used, <code>Base</code> is always 0, otherwise in the range of 0 to 63 (depending on the cycle repetition)
Rep	Defines, if the frame is sent every communication cycle (<code>Rep</code> = 1), or only every n -th communication cycle. Possible values are $1 \leq n \leq 64$, whereas n must be a power of 2
Len	Payload length of the frame in 16-bit words (0 ... 127)
BLen	Payload length of the frame in bytes (0 ... 254). Possible values are either 2^*Len , or $2^*Len - 1$, if the length in bytes is not even
Idx	Identifier of a buffer on the FlexCard, which is used by the driver to send or receive the frame. If the frame is an Rx-frame, <code>Idx</code> may be -1, indicating that this frame is received by the receiver FIFO
Sender	Name of the ECU, that sends the frame. Multiple senders are not allowed

FlexRay frames (Gateway Mode)

The description of the following keys is only valid for configurations where *GatewayMode* is active. For frame descriptions in standard mode, refer to section [FlexRay frames](#).

Frame.<n> =	Name Slot Ch Base Rep Len BLen [Sender]
-------------	---

Defines a Tx-/Rx-frame. The frames are numbered consecutively, starting with zero. The frame counter `n` starts with zero and is incremented for every new frame.

Each frame is defined by the following parameters:

Name	Name of the frame
Slot	Slot Id of the frame. Possible values are in the range of 1 and 2047
Ch	Channel, on which the frame is sent or received. Possible values are “A” for channel A, “B” for channel B, or “AB” for both channels
Base	Specifies the base cycle in which the frame is to be sent. If no cycle multiplexing is used, <code>Base</code> is always 0, otherwise in the range of 0 to 63 (depending on the cycle repetition)
Rep	Defines, if the frame is sent every communication cycle (<code>Rep</code> = 1), or only every n -th communication cycle. Possible values are $1 \leq n \leq 64$, whereas n must be a power of 2
Len	Payload length of the frame in 16-bit words (0 ... 127)

BLen	Payload length of the frame in bytes (0 ... 254). Possible values are either 2*Len, or 2*Len - 1, if the length in bytes is not even
Sender	Name of the ECU, that sends the frame. Multiple senders are not allowed

Frame.<n>.SendIdx = <nCC,BufIdx>[...]

List of identifiers for buffers on all FlexRay communication controllers, which are used by the driver to send the frame. Each buffer is specified as a comma separated tuple of `nCC` (identifying the communication controller) and `BufIdx` (identifying the buffer index).

Frame.<n>.RecvIdx = <nCC,BufIdx>[...]

List of identifiers for buffers on all FlexRay communication controllers, on which the frame is received. Each buffer is specified as a comma separated tuple of `nCC` (identifying the communication controller) and `BufIdx` (identifying the buffer index).

Optional Frame parameters

**Frame.<n>.TxMode = Mode [dT [dT_Min [dT_Max]]]
[CC.<nCC>.]TxFrame.<nTx>.TxMode = Mode [dT [dT_Min [dT_Max]]]**

Specifies, how a Tx-frame is transmitted. Possible values for `Mode` are:

Cyclic (TimeTriggered, Continuous)	Frame is transmitted cyclic
OnEvent (EventTriggered, SingleShot)	Frame is transmitted only on special events

In the static segment, a frame is usually transmitted cyclically only. The delay between two consecutively transmitted frames is then defined by the cycle multiplexing (base cycle, cycle repetition). In the dynamic segment, frames are either sent on event (e.g. once after power up) or cyclically with a longer delay.

If `dT` is specified, it defines the typical cycle time in micro seconds. If `dT_Min` and `dT_Max` are also specified, a jitter in the range of `[dT_Min .. dT_Max]` micro seconds will be considered as tolerable.

If `Mode` is *Cyclic* and `dT` is 0 micro seconds, then the frame will be sent as fast as possible (every possible communication cycle) as defined by the base cycle and cycle repetition.

If `Mode` is *OnEvent* and `dT` is 0 micro seconds, then the frame is sent exactly one time, whenever the send condition becomes true. If `dT` is greater than 0, then the frame will be sent cyclically as long as the send condition stays true. The send condition can be either a Frame Trigger or another user defined event.

**Frame.<n>.Receiver = Receiver
[CC.<nCC>.]TxFrame.<nTx>.Receiver = Receiver
[CC.<nCC>.]RxFrame.<nRx>.Receiver = Receiver**

Name of the ECU that receives the frame. Multiple receivers can be specified and are separated by white spaces.

FlexRay signals

Frame.<n>.Signals:
[CC.<nCC>.]TxFrame.<nTx>.Signals:
[CC.<nCC>.]RxFrame.<nRx>.Signals:
 Name Len Pos ByteOrder Unit [Type [EncType]]

Defines the signals of a frame. The signals are listed consecutively, each text line defines one signal.

Each signal is defined by the following parameters:

Name	Name of the signal
Len	Gives the size of the signal in bits
Pos	Specifies the location of the signal in the payload of the data frame. <i>Pos</i> is to be interpreted as the number of the first bit, which is occupied by the signal. It is assumed, that a signal is not fragmented
ByteOrder	The byte order of the signal. Possible values are "MSB" (Most Significant Byte first) for big-endian or Motorola byte order and "LSB" (Least Significant Byte first) for little-endian or Intel byte order
Unit	Specifies the unit of the signal
Type	Defines the base (raw) data type of the signal, according to the FIBEX specification
EncType	Defines how to convert the raw value of a signal into its physical value and vice versa

Referred the FIBEX specification, the following values are supported for *Type*:

A_UINT8	unsigned integer, 8 bits
A_INT8	signed integer, 8 bits
A_UINT16	unsigned integer, 16 bits
A_INT16	signed integer, 16 bits
A_UINT32	unsigned integer, 32 bits
A_INT32	signed integer, 32 bits
A_UINT64	unsigned integer, 64 bits
A_INT64	signed integer, 64 bits
A_FLOAT32	single precision floating point, 32 bit
A_FLOAT64	double precision floating point, 64 bit
A_ASCIISTRING	ascii string, 8 bit characters
A_UNICODE2STRING	unicode string, 16 bits characters
A_BYTEFIELD	byte field
A_BITFIELD	bit field

Referred to the FIBEX specification, possible values for *EncType* are:

IDENTICAL	physical and raw value are identical
LINEAR	linear mapping, defined by factor and offset
SCALE-LINEAR	multiple linear mappings, depending on the range of the raw value

TEXTTABLE	text string, one for each, or range of raw value
TAB-NOINTP	single or range of raw values, mapped to a table of physical values
FORMULA	mathematical formula

Frame.<n>.Multiplexers:**[CC.<nCC>.]TxFrame.<nTx>.Multiplexers:****[CC.<nCC>.]RxFrame.<nRx>.Multiplexers:****Name Len Pos ByteOrder**

Defines multiplexer signals of a frame. The multiplexer signals are listed consecutively, each text line defines one multiplexer.

Each multiplexer is defined by the following parameters:

Name	Name of the multiplexer
Len	Gives the size of the multiplexer in bits
Pos	Specifies the location of the multiplexer in the payload of the data frame. Pos is to be interpreted as the number of the first bit, which is occupied by the multiplexer. It is assumed, that a multiplexer is not fragmented
ByteOrder	The byte order of the signal. Possible values are “MSB” (Most Significant Byte first) for big-endian or Motorola byte order and “LSB” (Least Significant Byte first) for little-endian or Intel byte order

Optional Signal parameters**Frame.<n>.Signal.<m>.InvalidVal = Value****[CC.<nCC>.]TxFrame.<nTx>.Signal.<m>.InvalidVal = Value****[CC.<nCC>.]RxFrame.<nRx>.Signal.<m>.InvalidVal = Value**

Defines the value, which represents an invalid signal value. The signal is referred by the counter m , indicating the m -th line of the Frame.<n>.Signals text key.

Frame.<n>.Signal.<m>.Multiplexed = MPlexName SwCode**[CC.<nCC>.]TxFrame.<nTx>.Signal.<m>.Multiplexed = MPlexName SwCode****[CC.<nCC>.]RxFrame.<nRx>.Signal.<m>.Multiplexed = MPlexName SwCode**

Specifies, that the signal which is referenced by the counter m , is a multiplexer signal. The signal is only available, if the value of the multiplexer MPlexName is equal to SwCode.

Coding of a signal

Frame.<n>.Signal.<m>.Coding =	Vn0 Vn1 Vd0
[CC.<nCC>.]TxFrame.<nTx>.Signal.<m>.Coding =	Vn0 Vn1 Vd0
[CC.<nCC>.]RxFrame.<nRx>.Signal.<m>.Coding =	Vn0 Vn1 Vd0

Defines conversion parameters for signals with an *EncType* of *LINEAR*. The parameters *Vn0*, *Vn1* and *Vd0* are used in the following formula, to convert a raw value into its physical value and vice versa:

$$\text{Phys}(x) = (Vn0 + Vn1 * x) / Vd0$$
(EQ 2)

Frame.<n>.Signal.<m>.Coding:	
[CC.<nCC>.]TxFrame.<n>.Signal.<m>.Coding:	
[CC.<nCC>.]RxFrame.<n>.Signal.<m>.Coding:	
[Min,Max] Vn0 Vn1 Vd0	

Defines conversion parameters for signals with an *EncType* of *SCALE-LINEAR*, for two or more intervals. The parameters *Vn0*, *Vn1* and *Vd0* are used in the following formula, to convert a raw value into its physical value and vice versa:

$$\text{Phys}(x) = (Vn0 + Vn1 * x) / Vd0$$
(EQ 3)

Each text line defines a new set of conversion parameters *Vn0*, *Vn1* and *Vd0* for an interval of raw values. The parameters are valid, whenever the raw value of the signal is between the *Min* and *Max* value of the interval.

Frame.<n>.Signal.<m>.Coding =	Text
[CC.<nCC>.]TxFrame.<nTx>.Signal.<m>.Coding =	Text
[CC.<nCC>.]RxFrame.<nRx>.Signal.<m>.Coding =	Text

Defines a display text string for a signal with an *EncType* of *TEXTTABLE*.

Frame.<n>.Signal.<m>.Coding:	
[CC.<nCC>.]TxFrame.<n>.Signal.<m>.Coding:	
[CC.<nCC>.]RxFrame.<n>.Signal.<m>.Coding:	
[Min,Max] Text	

Defines display text strings for a signal with an *EncType* of *TEXTTABLE*, for two or more intervals.

Each text line defines a text string *Text* for an interval of raw values. A text string is valid, whenever the raw value of the signal is between the *Min* and *Max* value of the interval.

Frame.<n>.Signal.<m>.Coding =	Value
[CC.<nCC>.]TxFrame.<nTx>.Signal.<m>.Coding =	Value
[CC.<nCC>.]RxFrame.<nRx>.Signal.<m>.Coding =	Value

Defines a *Value* which represents the value of a signal with an *EncType* of *TABNO/INTP*.

Frame.<n>.Signal.<m>.Coding:
[CC.<nCC>.]TxFrame.<n>.Signal.<m>.Coding:
[CC.<nCC>.]RxFrame.<n>.Signal.<m>.Coding:
[Min,Max] Value

Defines a list of values to represent the values of a signal with an `EncType` of `TABNOINTP`, for two or more intervals.

Each text line defines a value for an interval of raw values. A `Value` is used, if the raw value is in between the `Min` and `Max` value of the interval.

Frame.<n>.Signal.<m>.Coding = Formula
[CC.<nCC>.]TxFrame.<nTx>.Signal.<m>.Coding = Formula
[CC.<nCC>.]RxFrame.<nRx>.Signal.<m>.Coding = Formula

Defines a `Formula` for conversion between the raw and physical of a signal with an `EncType` of `FORMULA`.

The formula string is currently not interpreted.

Frame.<n>.Signal.<m>.Coding:
[CC.<nCC>.]TxFrame.<n>.Signal.<m>.Coding:
[CC.<nCC>.]RxFrame.<n>.Signal.<m>.Coding:
[Min,Max] Formula

Defines formulas for conversion between the raw and physical of a signal with an `EncType` of `FORMULA`, for more than one interval.

The formula strings are currently not interpreted.

FlexRay frames (obsolete file version 1)

[CC.<nCC>.]TxFrame.<nTx>.Name = Name
[CC.<nCC>.]RxFrame.<nRx>.Name = Name

Defines the `Name` of a Tx-/Rx-frame. The frames are numbered consecutively, starting with zero. Both frame counters – `Tx_Num` and `Rx_Num` – start with zero and are incremented independently for every new frame.

[CC.<nCC>.]TxFrame.<nTx>.ID = Slot_Id
[CC.<nCC>.]RxFrame.<nRx>.ID = Slot_Id

Slot Id `Slot_Id` of a Tx-/Rx-frame. Possible values are in the range of 1 and 2047.

[CC.<nCC>.]TxFrame.<nTx>.Length = Length
[CC.<nCC>.]RxFrame.<nRx>.Length = Length

Payload length of the frame in 16-bit words (0 ... 127).

[CC.<nCC>.]TxFrame.<nTx>.ByteLength = Length
[CC.<nCC>.]RxFrame.<nRx>.ByteLength = Length

Payload length of the frame in bytes (0 ... 254). This parameter is optional and can be used, if the length in bytes is not even and therefore not equal to the length in 16-bit words.

Example TxFrame.0.Length = 10
 TxFrame.0.ByteLength = 19

[CC.<nCC>.]TxFrame.<nTx>.BufIdx = BufIdx
[CC.<nCC>.]RxFrame.<nRx>.BufIdx = BufIdx

Identifier of a buffer on the FlexCard, which is used by the driver to send or receive the frame. If the frame is an Rx-frame, BufIdx may be -1, indicating that this frame is received by the receiver FIFO.

[CC.<nCC>.]TxFrame.<nTx>.Receiver = Receiver
[CC.<nCC>.]RxFrame.<nRx>.Receiver = Receiver

Name of the ECU that receives the frame. Multiple receivers can be specified and are separated by white spaces.

[CC.<nCC>.]RxFrame.<nRx>.Sender = Sender
[CC.<nCC>.]TxFrame.<nTx>.Sender = Sender

Name of the ECU, that sends the frame. Multiple senders are not allowed.

[CC.<nCC>.]TxFrame.<nTx>.TxMode = Mode

Specifies, how a Tx-frame is transmitted. Possible values for Mode are:

Cyclic (TimeTriggered, Continuous)	Frame is transmitted cyclic
OnEvent (EventTriggered, SingleShot)	Frame is transmitted only on special events

In the static segment, a frame is usually transmitted cyclically only. The delay between two consecutively transmitted frames is then defined by the cycle multiplexing (base cycle, cycle repetition). In the dynamic segment, frames are either sent on event (e.g. once after power up) or cyclically with a longer delay.

[CC.<nCC>.]TxFrame.<nTx>.TxCycleTime = Time [Min [Max]]

Defines the delay between two consecutively transmitted frames. *Time* specifies the typical cycle time in micro seconds. If *Min* and *Max* are also specified, a jitter in the range of [*Min* .. *Max*] micro seconds will be considered as tolerable.

If the *TxMode* of the frame is *Cyclic* and *Time* is 0 micro seconds, then the frame will be sent as fast as possible (every possible communication cycle) as defined by the base cycle and cycle repetition.

If the *TxMode* is *OnEvent* and *Time* is 0 micro seconds, then the frame is sent exactly one time, whenever the send condition becomes true. If *Time* is greater than 0, then the frame will be sent cyclically as long as the send condition stays true. The send condition can be either a Frame Trigger or another user defined event.

[CC.<nCC>.]TxFrame.<nTx>.TxCycleTimeMin = Time

Defines the minimum cycle time for a cyclically transmitted frame in micro seconds.

[CC.<nCC>.]TxFrame.<nTx>.TxCycleTimeMax = Time

Defines the maximum cycle time for a cyclically transmitted frame in micro seconds.

[CC.<nCC>.]TxFrame.<nTx>.Channel = Channel
[CC.<nCC>.]RxFrame.<nRx>.Channel = Channel

Channel, on which the frame is sent or received. Possible values are “A” for channel A, “B” for channel B, or “AB” for both channels (Tx-frames).

[CC.<nCC>.]TxFrame.<nTx>.BaseCycle = BaseCycle
[CC.<nCC>.]RxFrame.<nRx>.BaseCycle = BaseCycle

Specifies the base cycle in which the frame is to be sent. If no cycle multiplexing is used, *BaseCycle* is always 0, otherwise in the range of 0 to 63 (depending on the cycle repetition).

[CC.<nCC>.]TxFrame.<nTx>.CycleRepetition = Repetition
[CC.<nCC>.]RxFrame.<nRx>.CycleRepetition = Repetition

Defines, if the frame is sent every communication cycle (*Repetition* = 1), or only every *n*-th communication cycle. Possible values are $1 \leq n \leq 64$, whereas *n* must be a power of 2.

FlexRay signals (obsolete file version 1)

[CC.<nCC>.]TxFrame.<nTx>.Signal.<Num>.Name = Name
[CC.<nCC>.]RxFrame.<nRx>.Signal.<Num>.Name = Name

Defines the `Name` of a signal in a Tx-/Rx-frame. The signals are numbered consecutively within the frame, starting with zero. The signal counter `Num` starts with zero and is incremented for every new signal.

[CC.<nCC>.]TxFrame.<nTx>.Signal.<Num>.Unit = Unit
[CC.<nCC>.]RxFrame.<nRx>.Signal.<Num>.Unit = Unit

Specifies the unit of the signal.

[CC.<nCC>.]TxFrame.<nTx>.Signal.<Num>.Size = Size
[CC.<nCC>.]RxFrame.<nRx>.Signal.<Num>.Size = Size

Gives the size of the signal in bits.

[CC.<nCC>.]TxFrame.<nTx>.Signal.<Num>.Start = Start
[CC.<nCC>.]RxFrame.<nRx>.Signal.<Num>.Start = Start

Specifies the location of the signal in the payload of the data frame. `Start` is to be interpreted as the number of the first bit, which is occupied by the signal. It is assumed, that a signal is not fragmented.

[CC.<nCC>.]TxFrame.<nTx>.Signal.<Num>.ByteOrder = ByteOrder
[CC.<nCC>.]RxFrame.<nRx>.Signal.<Num>.ByteOrder = ByteOrder

The byte order of the signal. Possible values are “MSB” (Most Significant Byte first) for big-endian or Motorola byte order and “LSB” (Least Significant Byte first) for little-endian or Intel byte order.

[CC.<nCC>.]TxFrame.<nTx>.Signal.<Num>.Type = Type
[CC.<nCC>.]RxFrame.<nRx>.Signal.<Num>.Type = Type

Defines the base (raw) data type of the signal, according to the FIBEX specification:

A_UINT8	unsigned integer, 8 bits
A_INT8	signed integer, 8 bits
A_UINT16	unsigned integer, 16 bits
A_INT16	signed integer, 16 bits
A_UINT32	unsigned integer, 32 bits

A_INT32	signed integer, 32 bits
A_UINT64	unsigned integer, 64 bits
A_INT64	signed integer, 64 bits
A_FLOAT32	single precision floating point, 32 bit
A_FLOAT64	double precision floating point, 64 bit
A_ASCIISTRING	ascii string, 8 bit characters
A_UNICODE2STRING	unicode string, 16 bits characters
A_BYTEFIELD	byte field
A_BITFIELD	bit field

[CC.<nCC>.]TxFrame.<nTx>.Signal.<Num>.EncType =	Type
[CC.<nCC>.]RxFrame.<nRx>.Signal.<Num>.EncType =	Type

Defines how to convert the raw value of a signal into its physical value and vice versa. According to the FIBEX specification, possible values are:

IDENTICAL	physical and raw value are identical
LINEAR	linear mapping, defined by factor and offset
SCALE-LINEAR	multiple linear mappings, depending on the range of the raw value
TEXTTABLE	text string, one for each, or range of raw value
TAB-NOINTP	single or range of raw values, mapped to a table of physical values
FORMULA	mathematical formula

Coding of a signal

Coding of a signal (obsolete file version 1)

For the conversion between raw and physical values of a signal, one or more codings can be defined. The most simple type of conversion is *IDENTICAL*, which means the physical value is the same as the raw value. Depending the value of the parameter *EncType*, additional conversion parameters (...Signal.<Num>.Coding.<CNum>...) may be defined.

[CC.<nCC>.]TxFrame.<nTx>.Signal.<Num>.Coding.<CNum>.Min =	Min
[CC.<nCC>.]RxFrame.<nRx>.Signal.<Num>.Coding.<CNum>.Min =	Min

For a signal coding that is defined only for a specific range of the raw value, this defines the lower border of the interval.

[CC.<nCC>.]TxFrame.<nTx>.Signal.<Num>.Coding.<CNum>.Max =	Max
[CC.<nCC>.]RxFrame.<nRx>.Signal.<Num>.Coding.<CNum>.Max =	Max

For a signal coding that is defined only for a specific range of the raw value, this defines the upper border of the interval.

<code>[CC.<nCC>.]TxFrame.<nTx>.Signal.<Num>.Coding.<CNum>.Text =</code>	<code>Text</code>
<code>[CC.<nCC>.]RxFrame.<nRx>.Signal.<Num>.Coding.<CNum>.Text =</code>	<code>Text</code>

For codings of type *TEXTTABLE*, this defines the display text for the specified interval.

<code>[CC.<nCC>.]TxFrame.<nTx>.Signal.<Num>.Coding.<CNum>.Vn0 =</code>	<code>Vn0</code>
<code>[CC.<nCC>.]RxFrame.<nRx>.Signal.<Num>.Coding.<CNum>.Vn0 =</code>	<code>Vn0</code>

Codings of type LINEAR and SCALE-LINEAR define a linear conversion for the specified range of raw values, according to the formula

$$\text{Phys}(x) = (Vn0 + Vn1 * x) / Vd0 \quad (\text{EQ } 4)$$

This parameter defines the value for `Vn0`.

<code>[CC.<nCC>.]TxFrame.<nTx>.Signal.<Num>.Coding.<CNum>.Vn1 =</code>	<code>Vn1</code>
<code>[CC.<nCC>.]RxFrame.<nRx>.Signal.<Num>.Coding.<CNum>.Vn1 =</code>	<code>Vn1</code>

Codings of type LINEAR and SCALE-LINEAR define a linear conversion for the specified range of raw values, according to the formula

$$\text{Phys}(x) = (Vn0 + Vn1 * x) / Vd0 \quad (\text{EQ } 5)$$

This parameter defines the value for `Vn1`.

<code>[CC.<nCC>.]TxFrame.<nTx>.Signal.<Num>.Coding.<CNum>.Vd0 =</code>	<code>Vd0</code>
<code>[CC.<nCC>.]RxFrame.<nRx>.Signal.<Num>.Coding.<CNum>.Vd0 =</code>	<code>Vd0</code>

Codings of type LINEAR and SCALE-LINEAR define a linear conversion for the specified range of raw values, according to the formula

$$\text{Phys}(x) = (Vn0 + Vn1 * x) / Vd0 \quad (\text{EQ } 6)$$

This parameter defines the value for `Vd0`.

<code>[CC.<nCC>.]TxFrame.<nTx>.Signal.<Num>.Coding.<CNum>.Value =</code>	<code>Value</code>
<code>[CC.<nCC>.]RxFrame.<nRx>.Signal.<Num>.Coding.<CNum>.Value =</code>	<code>Value</code>

For codings of type *TAB-NOINTP*, this defines the physical value for raw values in the specified range.

12.5.2 FlexCardParameters

The FlexCardParameters Info file provides some information and configuration for the installed FlexCards. Each FlexCard can be identified by an unique Id. This Id is being used to identify the FlexCard within CarMaker/HIL.

A list of all installed FlexCards can be obtained from the driver via the `/proc` file system:

Listing 12.7: List of FlexCards

```

1: [rt1-hil] 1) cat /proc/xeno_flexcard
2:
3: =====
4: * X E      FlexCard FlexRay interfaces (www.eberspaecher.com)
5: * N O          Version 5.3.0.2
6: * M A I          1 RT-interfaces for XENOMAI found
7: =====
8: * n use irq ----Serial---- ID -Firmware- -Hardware- FlexCardID UserCardID W L X*
9: 00      19 30000009810026 09 0x00050300 0x00010100 0x01ed906a 0x00000000 1 1 1
10:        BusType LIC SUP ----CC---- -Protocol- ----BG---- TinyStates
11:        FlexRay 4 2 0x00010300 0x00020100 0x00000000 0x0000000f
12:        CAN     8 4 0x00010000 0x00020000 0x0000000f

```

The Listing 12.7 above shows the output of the driver for a FlexCard PMC II with two Flex-Tiny II modules for FlexRay. The Id can be found in the column *FlexCardID*, here it is 0x01ed906a.

Header information

FileId = CarMaker-FlexCardParameters	Ver
---	------------

Identifies the info file as FlexCard parameters file of version *Ver*. Currently supported value for *Ver* is 1.

Configuration parameters

FlexCard.LogCards = bool

Activates the output of a list of available FlexCards into the CarMaker log file. This parameter can be used for debugging purpose. The default value is 0 (false).

FlexCard.LogFrames.Tx = bool
FlexCard.LogFrames.Rx = bool

Activates log messages for each received or sent FlexRay frame. These are debugging parameters and should be used with care. The default value is 0 (false).

FlexCard parameters

FlexCard.<nFC>.Id = Id

Defines the FlexCard from the view of CarMaker with the specified *Id* as the FlexCard with number *FC_Num*. This allows to distinguish between multiple FlexCards. All FlexCards have to be numbered consecutively, starting with 0. Depending on the number of available communication controllers on the specified FlexCard, additional settings can be defined. Info file

keys, that define communication controller specific settings will have the prefix `FlexCard.<nFC>.CC.<nCC>`, with $0 \leq nCC \leq NumCC$ (`NumCC` = number of available communication controllers on the FlexCard).

FlexCard.<nFC>.CC.<nCC>.Name = Name

Specifies an unique name for a FlexRay communication controller. This `Name` is used by the rest bus simulation, to find proper assignments to communication controllers, defined by the `Prefix` key in the [FlexRayParameters](#) file. This identification is especially important, when using the `GatewayMode`.

FlexCard.<nFC>.CC.<nCC>.TxAcknowledge = bool

Specifies, if acknowledge packages should be generated by the FlexCard for each Tx-frame. If set to 1 (true), a counter for each Tx-frame will be incremented as soon as the FlexCard driver reports the successful transmission of the frame. The default value is 0 (false).

FlexCard.<nFC>.CC.<nCC>.TxAckShowNullFrames = bool

Activates Tx acknowledge packages for NULL frames (frames without payload data within the static segment). The default value is 0 (false).

FlexCard.<nFC>.CC.<nCC>.TxAckShowPayload = bool

Specifies, if the FlexCard driver shall also report the payload with every Tx acknowledge package. The default value is 0 (false).

FlexCard.<nFC>.CC.<nCC>.RxShowNullFrames = bool

Specifies, if NULL frames (frames without payload data within the static segment) should be received by the driver. The default value is 0 (false).

FlexCard.<nFC>.CC.<nCC>.AutoRestart = bool

Enables the automatic recovery feature, if the FlexRay controller disable normal operation because of errors (e.g. message buffer overflow). If set to 1 (true), the FlexCard will re-synchronize automatically and return to normal operation after e.g. the cable was cut-off and re-plugged again. The default value is 0 (false).

FlexCard.<nFC>.CC.<nCC>.TPauseRestart = Time

Specifies how many seconds the driver should wait before trying to re-synchronize again, if the FlexRay controller disables normal operation because of errors (e.g. message buffer overflow). The default value is 1.0 (false).

FlexCard.<nFC>.CC.<nCC>.Term = bool

If set to 0 (false), this parameter disables the bus terminator on the FlexCard. The default value is 1 (true).

FlexCard.<nFC>.CC.<nCC>.TermReset = bool

Specifies, if the bus termination should be reset to 0 after the given communication controller is disabled. The default value is 0 (false).

FlexCard.<nFC>.CC.<nCC>.DictDefRawRx = bool
FlexCard.<nFC>.CC.<nCC>.DictDefRawTx = bool

Specifies, if Data Dictionary should be generated for each payload word of all Tx- or Rx-frame buffers configured on the FlexCard. The default value is 0 (false).

FlexCard.<nFC>.CC.<nCC>.TxFloatStatic = bool
FlexCard.<nFC>.CC.<nCC>.TxFloatDynamic = bool

Activates additional Tx buffers on the FlexRay controller, which will be used for transmission of FlexRay frames on arbitrary slots.

The default value is 0 (false).

Enabling this feature may cause collisions with Tx frames of other ECUs.



12.5.3 RBSParameters

The FlexRay rest bus simulation is configured with an Info file of type *RBSParameters*. This type of file can list sets for one or more rest bus simulations. Parameter keys for rest bus simulations must have the prefix *RBS.<nRBS>*, whereas *RBS_Num* consecutively numbers the configurations, starting with 0.

Header information

FileIdent = CarMaker-RBSParameters Ver

Identifies the info file as RBS parameters file of version *Ver*. Currently supported value for *Ver* is 1.

Configuration parameters

RBS.<nRBS>.ConfigFile = Path

Specifies the path to a *FlexRayParameters* info file, created by the CarMaker/HIL FlexConfig plug-in. This file will be used as source for frame and signal definitions.

RBS.<nRBS>.CC.<nCC> = FC_Id CC_Id RBS.<nRBS>.CC.<nCC> = Name

Defines, which FlexRay communication controllers are to be used by the rest bus simulation. The communication controllers are to be defined by the FlexCard number **FC_Id** and communication controller number **CC_Id**, as defined in the *FlexCardParameters* configuration file.

Instead of identifying communication controllers by numeric indices, you can also specify a **Name**. In this case, the communication controller will be identified by comparing the name with the definitions in the *FlexRayParameters* and *FlexCardParameters* files.

RBS.<nRBS>.Prefix = Prefix

Defines the prefix to be used when declaring Data Dictionary variables for FlexRay signals and other informations. If no prefix is defined, "FR" will be used as default.

RBS.<nRBS>.GwPrefix = Prefix

If the *GatewayMode* is active, this defines the prefix to be used when declaring Data Dictionary variables for FlexRay signals and other informations on the output side of the gateway. If no prefix is defined, "GW" will be used as default.

RBS.<nRBS>.AutoStart = bool

Configures the rest bus simulation to start-up the FlexRay communication automatically right after initialization. The default value is 0 (false).

RBS.<nRBS>.AutoConfig = bool

Configures the rest bus simulation to load the CHI configuration file to the FlexRay communication controller automatically. The default value is 1 (true).

RBS.<nRBS>.TxCyclicMapped = bool

Specifies if FlexRay frames with mapped signals are considered as to be sent cyclic. The default value is 1 (true).

RBS.<nRBS>.TxCyclicUnMapped = bool

Specifies if FlexRay frames where no signals are mapped, should be considered as to be sent cyclic. The default value is 0 (false).

RBS.<nRBS>.InitTxDataValue = Val

For those parts of the payload in a FlexRay frame which are not occupied by signals, this parameter defines what should be used as default raw value. The default value is 0.

RBS.<nRBS>.SigState.None =	Val
RBS.<nRBS>.SigState.Valid =	Val
RBS.<nRBS>.SigState.Invalid =	Val
RBS.<nRBS>.SigState.Unavailable =	Val
RBS.<nRBS>.SigState.Undefined =	Val
RBS.<nRBS>.SigState.Error =	Val

These parameters allow to customize the possible values of state variables, that indicate the different signal states. Possible are values in the range of -128 and 127:

Signal state	Value of state variable, if Signal is
None	of unknown state (default: 0)
Valid	of valid state (default: 1)
Invalid	of invalid state (default: 2)
Unavailable	unavailable (default: 3)
Undefined	undefined (default: 4)
Error	in error state (default: 5)

RBS.<nRBS>.DictDefMapped = bool

If set to 0 (false), mapped signals will not be added to the Data Dictionary. By default, this parameter is 1 (true), i.e. mapped signals will be always added to the Data Dictionary.

RBS.<nRBS>.DictDefUnMapped = bool

Defines, if signals should be added to the Data Dictionary even if they are not mapped. The default is 0 (false).

RBS.<nRBS>.DictDefTx = bool
RBS.<nRBS>.DictDefRx = bool
RBS.<nRBS>.DictDefGw = bool

If set to 0 (false), no Tx-/Rx- or Gateway signals will be added to the Data Dictionary. By default this parameter is 1 (true) for Rx-/Tx-signals, but 0 (false) for Gateway signals. Tx-/Rx-signals will be added to the Data Dictionary.

RBS.<nRBS>.DictDefTxRaw = bool
RBS.<nRBS>.DictDefRxRaw = bool
RBS.<nRBS>.DictDefGwRaw = bool

Defines, if also raw values of Tx-/Rx- or Gateway signals should be added to the Data Dictionary. The default is 0 (false).

RBS.<nRBS>.DictDefTxState = bool
RBS.<nRBS>.DictDefRxState = bool
RBS.<nRBS>.DictDefGwState = bool

Defines, if also the states of Tx-/Rx- or Gateway signals should be added to the Data Dictionary. The default is 0 (false).

RBS.<nRBS>.DictDefPrefixFrameId = bool
RBS.<nRBS>.DictHexIds = bool

When signal variables are added to the Data Dictionary, the name of the FlexRay frame will be included in the name of the Data Dictionary variable. If `DictDefPrefixFrameId` is set to 1 (true), the name of the frame will be replaced by a combination of the slot id and the cycle multiplexing settings. If `DictHexIds` is set to 1 (true), the slot id will be in hexadecimal notation, otherwise in decimal notation. The default value for both parameters is 0 (false).

RBS.<nRBS>.DictDefPrefixECU = bool

If set to 0 (false), the name of the sender ECU will not be included in the names of generated Data Dictionary quantities. The default value is 1 (true).

RBS.<nRBS>.DictDefPrefixTx = bool
RBS.<nRBS>.DictDefPrefixRx = bool
RBS.<nRBS>.DictDefPrefixGw = bool

Defines if the names of generated Data Dictionary quantities shall also include a prefix "Tx", "Rx", or "Gw" for Tx-/Rx- or Gateway signals. The default value is 0 (false).

RBS.<nRBS>.WarnDictErrors = bool

When analyzing the mappings for FlexRay signals, a warning message is written to the session log whenever a mapped Data Dictionary variable does not exist. If set to 0 (false), a normal log message will be written instead. The default value is 1 (true).

RBS.<nRBS>.GwRouteUnknown = bool

When *GatewayMode* is active, this parameter defines, whether unknown frames should be routed, or not. The default value is 1 (true).

Tx- / Rx- / Gw- Signal Mappings

RBS.<nRBS>.TxMap[.<ECUName>]:
RBS.<nRBS>.RxMap[.<ECUName>]:
RBS.<nRBS>.GwMap[.<ECUName>]:
 <FrmName>.<SigName> MapVar [Factor [Offset [Min [Max]]]]
 <FrmName>.<SigName> Const Value

This defines a mapping to a Data Dictionary variable or to a constant value. The Data Dictionary variable must exist. Ideally, the FlexRay signal and the Data Dictionary variable have the same unit. If not, optional parameters can be specified for conversion or signal conditioning. If a signal shall be mapped to a constant value, the keyword *Const* is to be used instead of the name of a Data Dictionary variable followed by the desired value:

ECUName	Name of Sender ECU
FrmName	Name of FlexRay frame
SigName	Name of signal
MapVar	Name of Data Dictionary variable
Factor	Factor for conversion, optional
Offset	Offset for conversion, optional
Min	Minimum possible value, optional
Max	Maximum possible value, optional
Const	Keyword, indicating mapping to constant value
Value	Desired value for mapping to constant

In combination with the parameter *DictDefPrefixECU* set to 0, the *ECUName* needs to be omitted.

Rolling Counters

RBS.<nRBS>.TxMap[.<ECUName>]:			
RBS.<nRBS>.RxMap[.<ECUName>]:			
RBS.<nRBS>.GwMap[.<ECUName>]:			
<FrmName>.<SigName>	RollCnt	Std	[Min [Max [Incr]]]
<FrmName>.<SigName>	RollCnt	<Type>	[Args]

Defines a signal to be a message counter. Following types are supported:

ECUName	Name of Sender ECU
FrmName	Name of FlexRay frame
SigName	Name of signal
RollCnt	Keyword to define message counter signal
Std	Standard Rolling Counter: <ul style="list-style-type: none">• starts with minimum value (default 0)• increments with each sent message• wraps back to 0 after maximum reached
Min	Minimum value for Standard Rolling Counter, optional
Max	Maximum value for Standard Rolling Counter, optional
Incr	Step size (increment) for Standard Rolling Counter, optional
Type	User defined keyword for user defined Counter function
Args	Optional / possible arguments for user defined Counter function

User defined Rolling Counter functions must have been registered with the function [RBS_Register_MapFunc\(\)](#).

Checksum Calculation

RBS.<nRBS>.TxMap[.<ECUName>]:				
RBS.<nRBS>.RxMap[.<ECUName>]:				
RBS.<nRBS>.GwMap[.<ECUName>]:				
<FrmName>.<SigName>	CRC	J1850	[Start [Count]]	
<FrmName>.<SigName>	CRC	<Type>	[Args]	

Defines a CRC mapping. Following types of CRCs are supported:

ECUName	Name of Sender ECU
FrmName	Name of FlexRay frame
SigName	Name of signal
CRC	Keyword to define CRC signal
J1850	CRC signal, as defined by SAE J1850
Start	Start byte for CRC calculation in message, optional
Count	Number of data bytes for CRC calculation, optional
Type	User defined keyword for user defined CRC function
Args	Optional / possible arguments for user defined CRC function

User defined CRC functions must have been registered with [RBS_Register_MapFunc\(\)](#).

Frame Trigger

RBS.<nRBS>.TxMap[.<ECUName>]:				
RBS.<nRBS>.GwMap[.<ECUName>]:				
<FrmName>	Trigger	RisingEdge	[Condition]	
<FrmName>	Trigger	FallingEdge	[Condition]	
<FrmName>	Trigger	LevelHigh	[Condition]	
<FrmName>	Trigger	LevelLow	[Condition]	

Defines a Frame trigger mapping. Following trigger types are supported:

ECUName	Name of Sender ECU
FrmName	Name of FlexRay frame
Trigger	Keyword to define frame trigger
Condition	Condition (Realtime Expression)
RisingEdge	Trigger on rising edge: fires, if condition changes from false to true
FallingEdge	Trigger on falling edge: fires, if condition changes from true to false
LevelHigh	Trigger on high level: fires, if condition is true
LevelLow	Trigger on low level: fires, if condition is false

Hook Functions

For operations that have to be executed right on the reception or transmission of a frame or signal, hook functions of different types can be mapped. A hook function mapping can refer to any function that has been registered with [RBS_Register_HookFunc \(\)](#).

RBS.<nRBS>.TxHook =	Name
RBS.<nRBS>.RxHook =	Name

Assigns the hook function registered with the name **Name**, as global Tx- / Rx- hook function. Global hook functions will be called with every frame right after reception and before decoding the data (**RxHook**), or on transmission, after all signals have been encoded (**TxHook**).

RBS.<nRBS>.Tx.[<ECUName>.<FrmName>.<SigName>.HookFunc =	Name
RBS.<nRBS>.Rx.[<ECUName>.<FrmName>.<SigName>.HookFunc =	Name

To map a hook function to a given signal. For Rx- signals, the hook function is called after the signal value has been decoded from the payload data of a frame. For Tx- signals, the hook function is called right before encoding the signal value into the payload data.

ECUName	Name of Sender ECU (wildcard allowed)
FrmName	Name of FlexRay frame
SigName	Name of signal
Name	Name of registered signal hook function

In combination with the parameter `DictDefPrefixECU` set to 0, the `ECUName` can be omitted. With `DictDefPrefixECU` set to 1 (default), a wildcard (*) allows to map the hook function to all frames of name `FrmName` with a signal of name `SigName`.

RBS.<nRBS>.Tx.[<ECUName>.<FrmName>[.<SigName>].CRCFunc =	Name
---	-------------

To calculate checksums on the payload data of frames, CRC hook functions can be mapped. CRC hook functions are called after all signals have been updated and encoded into the payload data of a Tx-frame.

ECUName	Name of Sender ECU (wildcard allowed)
FrmName	Name of FlexRay frame
SigName	Name of signal
Name	Name of registered CRC hook function

If the CRC hook function is mapped to a frame, the calculated CRC will not be evaluated, i.e. it is up to the user to encode it into the payload data. When mapped to a signal, the calculated CRC will be assigned to the signal variable and the payload data of the frame will be updated with the signal value.

RBS.<nRBS>.Rx.[<ECUName>.<FrmName>.FrameFunc =	Name
---	-------------

This parameter allows to assign a hook function to a Rx-frame. The hook function will be called after all payload data has been decoded and all signal values have been updated.

ECUName	Name of Sender ECU (wildcard allowed)
FrmName	Name of FlexRay frame
Name	Name of registered frame hook function

Chapter 13

SOME/IP

With the need for more flexible and more efficient communication solutions, the automotive industry has been researched the possibility of using ethernet technology in automotive applications. Ethernet is fast (100 Mbit/s, 1 Gbit/s and more), supports reliable (TCP/IP) and unreliable (UDP/IP) communication; and it is cheap.

However, implementing a new communication standard which has the potential of replacing the traditional CAN bus, is not as easy. Compared to e.g. FlexRay, one of the main advantages of CAN is still the simple wiring – CAN uses a single unshielded twisted pair. You won't find new friends with replacing a single unshielded twisted pair as part of the wiring harness in a car, by two or more shielded twisted pairs – not only because of the additional weight.

The solution for this dilemma is SOME/IP (**S**calable service-**O**riented **M**iddlewar**E** over **I**P) as communication protocol, in combination with a special physical layer: BroadR-Reach. Using BroadR-Reach as physical layer, allows full duplex communication over a 100 Mbits/s over a single unshielded twisted pair.

13.1 Overview

13.1.1 Basic information

The SOME/IP implementation for CarMaker/HIL is based on the AUTOSAR release 4.1 and FIBEX 4.1.0 and follows the following specifications:

- AUTOSAR R4.1 Rev 3: Example for a Serialization Protocol (SOME/IP) Version 1.1.0 (http://www.autosar.org/fileadmin/files/releases/4-1/software-architecture/communication-stack/auxiliary/AUTOSAR_TR_SomelpExample.pdf)
- AUTOSAR R4.1 Rev 3: Specification of Service Discovery Version 1.2.0 (http://www.autosar.org/fileadmin/files/releases/4-1/software-architecture/system-services/standard/AUTOSAR_SWS_ServiceDiscovery.pdf)
- ASAM MCD-2 NET (FIBEX) Version 4.1.0 (<http://www.asam.net>)

13.1.2 Features

- Import of FIBEX version 4.1 data bases
- Export of lists of services and events for configuration of the SOME/IP rest bus simulation in CarMaker/HIL:
- No need to recompile the CarMaker/HIL realtime application. The rest bus simulation is configured with CarMaker Info files and can be changed even without restarting the realtime application.

13.2 Rest Bus Simulation in CarMaker/HIL

13.2.1 Integration of SOME/IP in a CarMaker Project

The project template for CarMaker/HIL is already prepared for a SOME/IP rest bus simulation. However, SOME/IP is not active by default. It needs to be enabled in the Makefile, followed by a rebuild of the realtime application.

Makefile

On compilation, the preprocessor macro `WITH_SOMEIP` is used to activate the SOME/IP rest bus simulation function. In the Makefile of the project template, there is already a line which adds the macro `WITH_SOMEIP` to the Makefile variable `DEF_CFLAGS`, but it is commented out:

```
### SOME/IP
#DEF_CFLAGS += -DWITH_SOMEIP
```

By activating the line, different code sections in the modules `IO.c` and `User.c` are enabled, which manage the SOME/IP communication together with the rest bus simulation.

13.2.2 Initialization of SOME/IP Framework

Basic Initialization

Basic initialization is done in the module `IO.c` in three steps:

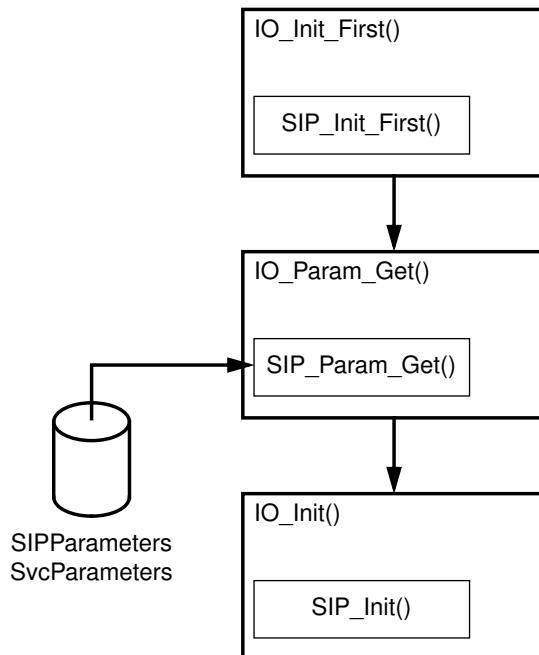


Figure 13.1: Basic Initialization

In `IO_Init_First()`, the function `SIP_Init_First()` is called to perform an early initialization of the SOME/IP rest bus simulation.

The configuration is performed in `IO_Param_Get()`: `SIP_Param_Get()` reads the configuration for the rest bus simulation out of the `SIPParameters` file, which references a `SvcParameters` file for the bus configuration (service and ECU descriptions).

In `IO_Init()`, the functions `SIP_Init()` is called in order to configure all used network interfaces and to prepare for start-up.

Data Dictionary Quantities

After the initialization of the network interfaces and the SOME/IP rest bus simulation, the function [SIP_DeclQuants\(\)](#) is called in `User_DeclQuants()`. The purpose of this function is to provide access to all SOME/IP signals as defined with the [SIPParameters](#) file.

By default, all generated Data Dictionary quantities follow the naming scheme below:

```
<Prefix>.<ECUName>.<SvcName>.<EventName>.<ParName>
<Prefix>.<ECUName>.<SvcName>.<FieldName>
```

Prefix	Prefix for identification of the rest bus simulation (default: SIP)
ECUName	Name of the ECU, that provides the service
SvcName	Name of the Service
EventName	Name of the Event
ParName	Parameter Name of Event (Input or Return)
FieldName	Field Name

Depending on the configuration, the number of SOME/IP signals may be very high. However, there are possibilities to limit the number of Data Dictionary quantities and to customize the naming scheme with settings in the [SIPParameters](#) file.

Completion of Initialization

The SOME/IP initialization is completed in `IO_Init_Finalize()`:

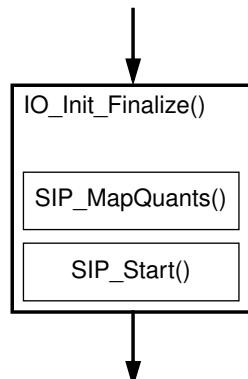


Figure 13.2: Completion of Initialization

First, mappings between CarMaker model quantities and SOME/IP signals are set-up as defined in the [SIPParameters](#) file. This is performed by the function [SIP_MapQuants\(\)](#), which also manages special mappings like CRC or counter signals. By calling the function [SIP_Start\(\)](#), the SOME/IP communication is started and the rest bus simulation is prepared for the cyclic management of SOME/IP messages.

13.2.3 Rest Bus Simulation

Input Processing

SOME/IP messages are received and processed in `IO_In()` at the beginning of each simulation cycle:

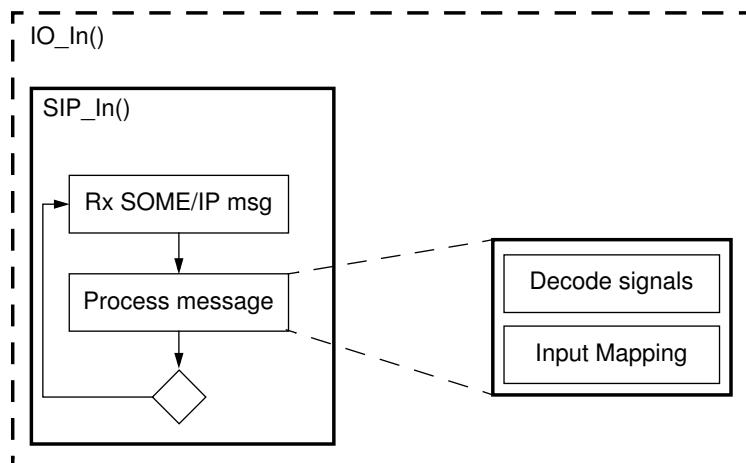


Figure 13.3: Input Processing

First, the function `SIP_In()` reads all available SOME/IP messages from the network interface(s). With each frame, a callback function from the rest bus simulation module is called to decode signal values and to perform input mappings.

Output Processing

For output of simulation results to the SOME/IP network, the functions `User_Out()` and `IO_Out()` are involved:

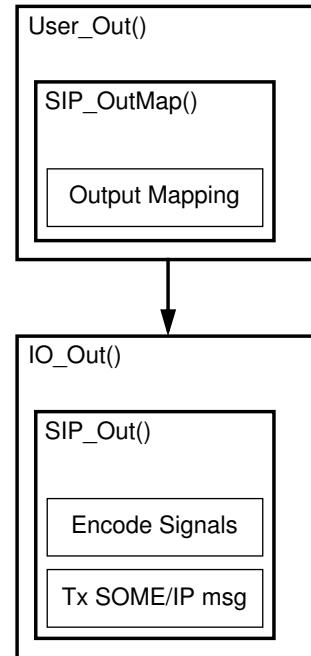


Figure 13.4: Output Processing

In `User_Out()` the function `SIP_OutMap()` is called, which performs the mapping between recently updated model quantities and the SOME/IP I/O signal variables. At this point of time no signal value is encoded into the payload data of SOME/IP messages, which allows dedicated signal values to be changed, e.g. for failure simulation.

At the end of the simulation cycle – in `IO_Out()` – `SIP_Out()` is called to encode signal values into SOME/IP messages and to send Tx messages to the hardware.

13.2.4 Customization

Of course, a rest bus simulation is not just sending and receiving SOME/IP messages. Also the data bytes of the frames need to be filled with life. This means to set the values of signals according to the current situation and to react also on the state of received frames and signals.

While many signals can be mapped directly from and to quantities of the CarMaker models, others may be just constant.

There are also signals which have special meanings like check sums or alive counters. Or a data frame might require to be managed with low-leveled bit operations.

For CRC and rolling counter signals, there are already pre-defined calculation functions available, which can be used with standard mappings. With signals and frames, that cannot be handled with the default functionality, additional frame and signal hook functions can be registered. Supported types of hook functions are:

- Frame hook functions for low-level access to data bytes of received and sent frames (`SIP_HookFuncRx`, `SIP_HookFuncTx`)
- CRC hook functions for calculation of check sums over specific frames (`SIP_HookFuncCRC`)
- Mapping functions for CRC signals with non-standard algorithm (`SIP_MapFuncCRC`)
- Mapping functions for rolling counter signals with non-standard algorithm (`SIP_MapFuncRollCnt`)

Input Processing

On reception of a SOME/IP it is first identified the addressed service interface and event. Then, the payload data is decoded and processed:

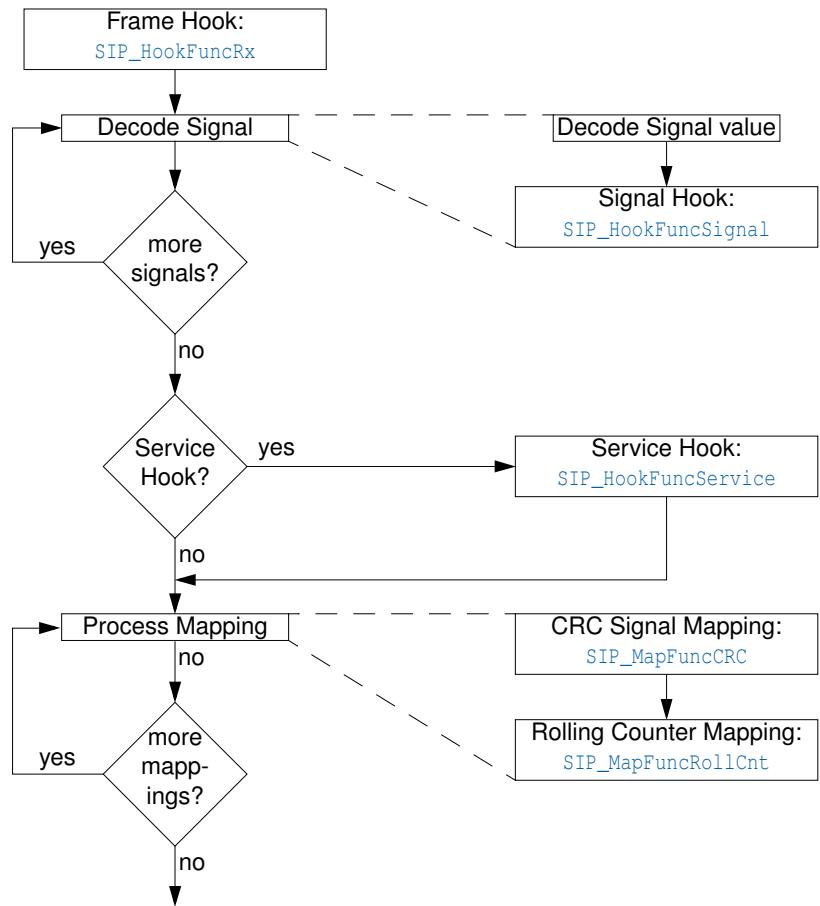


Figure 13.5: Input Processing

Output Processing

Whenever a SOME/IP message is to be sent within the current simulation cycle, the payload data needs to be updated with current values for all its signals:

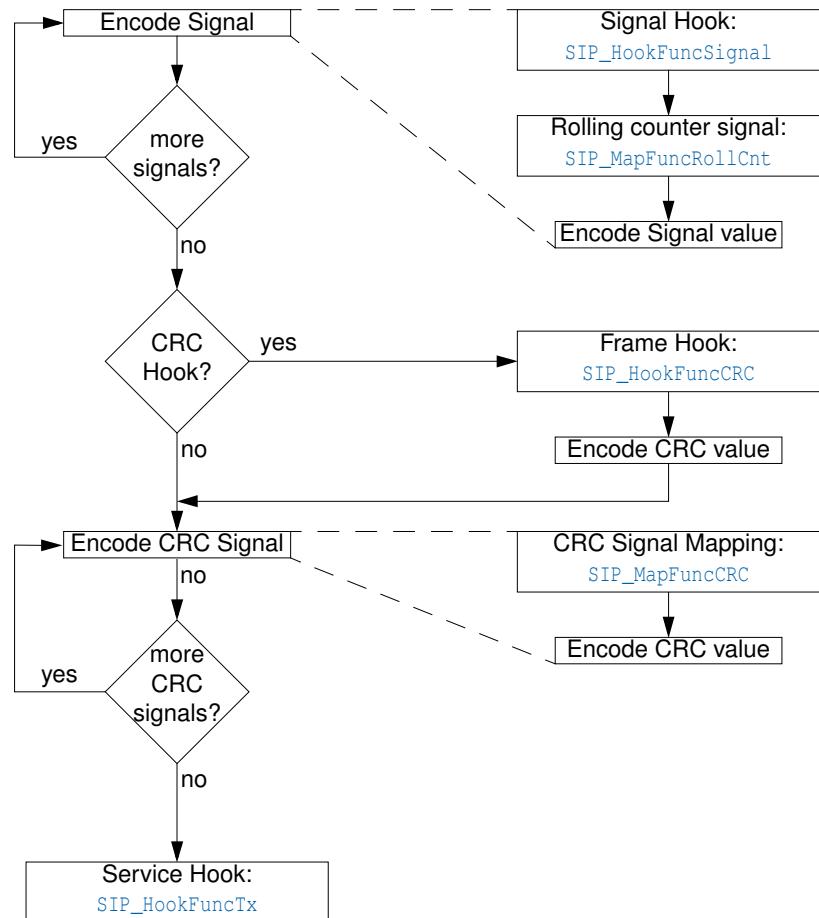


Figure 13.6: Output Processing

13.3 Functional description

13.3.1 SOME/IP RBS Function Interface

Configuration / Initialization

SIP_Init_First ()

```
void SIP_Init_First (void);
```

Description

First (early) initialization of the SOME/IP rest bus simulation module. This function needs to be called prior to any other `SIP_*`() function.

SIP_Param_Get ()

```
int SIP_Param_Get (struct tInfos *inf, const char *pfx);
```

Description

Reads the configuration of the SOME/IP rest bus simulation from the given info file handle `inf`. If a prefix is passed, too (`pfx != NULL`), then “`pfx.`” will be prepended to all keys which are read from the given info file handle `inf`.

First, `SIP_Param_Get ()` tries to read the key `SIPParameters`. If `inf` is not `NULL` and the key exists, then its value is interpreted as the name of another info file with the Fileident `SIPParameters`. All configuration parameters will then be read from this file. If `inf` is `NULL`, then a file with name `SIPParameters` will be searched. The format of the `SIPParameters` file is described in [section 13.4.2 'SIPParameters'](#).

If `inf` is not `NULL` and if the key `SIPParameters` does not exist, then the SOME/IP rest bus configuration will be read directly from the info file handle `inf` (eventually prefixed with “`pfx.`”).

Return Value

- 0, if all SOME/IP rest bus parameters have been successfully read.
- Otherwise: <0. Additional information about the reason is written into the session log.

SIP_Init ()

```
int SIP_Init (void);
```

Description

Configures all instances of provided and consumed service interfaces for all simulated ECUs. The service interface instances are registered to the service discovery module, each virtual ECU is bound to its own virtual ethernet interface.

SIP_DeclQuants ()

```
void SIP_DeclQuants (void);
```

Description

Defines Data Dictionary entries for events and fields of all (provided and consumed) service instances that are handled with the rest bus simulations. Depending on different settings (s. [section 13.4.2 'SIPParameters'](#)), this can be quantities for mapped signals, unmapped signals, or both.

SIP_MapQuants ()

```
int SIP_MapQuants (void);
```

Description

Creates and sets up the mappings for all SOME/IP events and field of service instances, as defined in the SOME/IP rest bus parameters file [SIPParameters](#).

SIP_Register_HookFunc ()

```
int SIP_Register_HookFunc (char *Name, tSIP_HookFuncType Type, tSIP_HookFunc Func);
```

Description

Allows to register a hook function for SOME/IP services. Hook functions can be registered with a name and are then available for activation in the SOME/IP rest bus parameters file. Following types of hook functions are supported:

SIP_HookFuncRx (1)	Global hook function for received SOME/IP messages
SIP_HookFuncTx (2)	Global hook function for transmission of SOME/IP messages
SIP_HookFuncCRC (3)	CRC hook function, assigned to selected Tx frames
SIP_HookFuncEvent (4)	Event hook function, assigned to selected Rx frames
SIP_HookFuncField (5)	Field hook function, assigned to selected Rx frames
SIP_HookFuncSvc (6)	Service hook function, assigned to selected service interfaces

A hook function is of type `tSIP_HookFunc` and has the following function prototype:

```
int SIP_HookFunc (struct tSIP *, struct tSIP_Service *, tSIP_HookArgs *);
```

Depending on the selected `Type`, the hook function will be called in different situations:

SIP_HookFuncRx	Called with every received SOME/IP message: • right after reception • before decoding of payload data
SIP_HookFuncTx	Called with every transmitted SOME/IP message: • after all signals have been encoded into the payload data • immediately before sending the message

SIP_HookFuncCRC	Called on transmission of selected SOME/IP message: <ul style="list-style-type: none"> • after normal signals have been encoded into the payload • if connected with a signal, the signal value is updated with the calculated CRC and encoded into the payload data • before output signal mappings of type CRC are processed
SIP_HookFuncEvent	Called on reception of selected SOME/IP event: <ul style="list-style-type: none"> • after all signals have been decoded • before input signal mappings are processed
SIP_HookFuncField	Called on transmission and reception of selected SOME/IP field message: <ul style="list-style-type: none"> • for Tx-signal, before encoding value of selected signal • for Rx-signal, after decoding value of selected signal

Whenever a registered callback function is called, pointers to the SOME/IP rest bus simulation struct (`struct tSIP *`) and a pointer to the current service interface instance struct (`struct tService *`) will be passed. The data structs are defined in [Listing 13.1](#) and [Listing 13.2](#):

[Listing 13.1: SOME/IP Rest Bus Simulation struct tSIP](#)

```

1:  typedef struct tSIP {
2:      char          *Cfg;
3:      struct tInfos *Inf;
4:      char          *VarInfo;
5:      char          *Prefix;
6:      char          *IfName;
7:      int           Flags;
8:      int           TxInitVal;
9:      char          SigStates[SIP_SState_nStates];
10:     char          **ProvSvcNames;
11:     char          **ConsSvcNames;
12:     tSIP_ECU      *ECU;
13:     int           nECU;
14:     tSIP_Service   **Service;
15:     int           nService;
16:     tSIP_Service   **TxService;
17:     int           nTxService;
18:     tSIP_Service   **RxService;
19:     int           nRxService;
20:     tSIP_SvcMap   **TxMapping;
21:     int           nTxMapping;
22:     int           mTxMapping;
23:     tSIP_SvcMap   **RxMapping;
24:     int           nRxMapping;
25:     int           mRxMapping;
26:     char          **DictFilter;
27: } tSIP;
```

With the fourth argument a pointer to the selected signal frame is passed (`struct tSIP_Service`), which is defined in [Listing 13.2](#):

Listing 13.2: Service interface instance struct `tSIP_Service`

```

1:  typedef struct tSIP_Service {
2:      struct tSIP          *SIP;
3:      char                 *Name;
4:      unsigned             SvcId;
5:      unsigned             InstId;
6:      unsigned             Prio;
7:      unsigned short       MajVer;
8:      unsigned short       MinVer;
9:      tSIP_SvcEP           Provider;
10:     tSIP_SvcEP           *Consumer;
11:     int                  nConsumer;
12:     eSIP_SvcState        State;
13:     tSIP_Field           **Field;
14:     int                  nField;
15:     tSIP_Event            **Event;
16:     int                  nEvent;
17:     tSIP_EventGroup       **EventGroup;
18:     int                  nEventGroup;
19:     int                  SubscriptionCnt;
20:     double               TS;
21:     int                  StartupDelay;
22:     int                  CycleTime;
23:     int                  CycleTimeStd;
24:     int                  CycleTimeMin;
25:     int                  CycleTimeMax;
26:     struct tSIP_Trigger  *Trigger;
27: } tSIP_Service;
```

The last argument passed to the hook function is of type `tSIP_HookArgs` (s. [Listing 13.3](#)). The struct contains a pointer to the selected signal (NULL, if no signal) and up to four numeric arguments of type `int`, as defined in the SOME/IP rest bus parameters file.

Listing 13.3: Hook Function Arguments struct `tSIP_HookArgs`

```

1: #define SIP_MAX_HOOK_FUNC_ARGS 4
2: typedef struct tSIP_HookArgs {
3:     struct tSIP_Signal  *Signal;
4:     int                  ArgC;
5:     int                  ArgV[SIP_MAX_HOOK_FUNC_ARGS];
6: } tSIP_HookArgs;
```

Return value of Hook Function

For hook functions of type `SIP_HookFuncRx`:

- <0: received message is discarded, no signals are decoded.
- >=0: message and signals are processed.

For hook functions of type `SIP_HookFuncTx`:

- <0: message is discarded, i.e. message will not be sent.
- >=0: message is sent.

For hook functions of type `SIP_HookFuncCRC` and if the CRC function is connected to a signal:

- =0: the selected signal will be encoded into the payload data.
- !=0: the calculated CRC value will not be encoded.

For all other types, the return value does not have any effect.

SIP_Register_MapFunc()

```
int SIP_Register_MapFunc (char *Name, tSIP_MapFuncType Type, tSIP_MapFunc Func);
```

Description

Registers a user defined function for rolling counter or CRC signals, which can be referenced in *RollCnt* or *CRC* signal mappings, just by the registered name *Name*. Following types of mapping functions are supported:

SIP_MapFuncCRC (1)	Mapping function for calculation of CRC signals
SIP_MapFuncRollCnt (2)	Mapping function for calculation of Rolling Counter signals

A mapping function is of type *tSIP_MapFunc* and has the following function prototype:

```
int SIP_MapFunc (unsigned char *Data, int DLen, int Argc, int *Argv);
```

Whenever a map function is called, a pointer to the payload data of the SOME/IP message is passed with the argument *Data*, whereas the second argument *DLen* gives the length of the payload data in bytes. The fourth argument *Argv* is a pointer to an integer array with *Argc* elements.

For functions of type *SIP_MapFuncCRC* up to 6 integer values can be passed, as defined with the signal mapping in the SOME/IP rest bus parameters file.

For functions of type *SIP_MapFuncRollCnt* the first element in the array of *Argv* will be the old value of the counter. Additional up to 5 integer values can be passed, if defined with the signal mapping in the SIPParameters file.

Return value of Map Function

For mapped Rx-signals, the return value of a map function reflects the state of the signal. Supported signal states are:

SIP_SState_None	no state, unknown (default: 0)
SIP_SState_Valid	signal is valid (default: 1)
SIP_SState_Invalid	signal value is invalid (default: 2)
SIP_SState_Unavailable	signal is not available (default: 3)
SIP_SState_Undefined	signal is not defined (default: 4)
SIP_SState_Error	signal is in error state (default: 5)

For Tx-signals, the return value does not have any effect, but the state of a signal defines whether the mapping is performed or not.

SIP_Cleanup ()

```
void SIP_Cleanup (void);
```

Description

Terminates the SOME/IP rest bus simulation and frees allocated memory. Before using any *SIP_**() function again, the module needs to be reinitialized with [SIP_Init_First \(\)](#).

Start / Stop Rest Bus Simulation

SIP_Start ()

```
int SIP_Start (void);
```

Description

Starts-up all SOME/IP communication, starting with service discovery (announce / find service) and service description, that are assigned to rest bus simulations and begins with the reception and transmission of SOME/IP events.

SIP_Stop ()

```
int SIP_Stop (void);
```

Description

Stops the reception and transmission of SOME/IP events, unsubscribes services, and terminates the activities of the SOME/IP service discovery.

Cyclic Calculations (realtime)

SIP_In ()

```
void SIP_In (unsigned CycleNo);
```

Description

Receives SOME/IP messages and processes received events. This function needs to be called at the beginning of each simulation step of the realtime application, i.e. in `IO_In()`. Whenever a SOME/IP message is received, the payload data is analyzed and all signals are updated. If there are signal mappings bound to that message, they are processed, too.

SIP_OutMap ()

```
int SIP_OutMap (unsigned CycleNo);
```

Description

After the calculation step of the model but before I/O output, this function needs to be called in each simulation step to update all signal variables with new values – either from mapped model variables or mapped functions. Call this function at the end of `User_Calc()` before calling `IO_Out()`.

After [SIP_OutMap \(\)](#), all signal variables are updated, but not yet encoded into payload data of Tx-messages. Hence, manipulations on the signal values can still take place.

SIP_Out ()

```
void SIP_Out (unsigned CycleNo);
```

Description

This function manages the transmission of SOME/IP messages and needs to be called at the end of each simulation step, i.e. in `IO_Out()`.

13.4 Configuration Files

13.4.1 SvcParameters

The main file of the generated output from the CarMaker/HIL FIBEX import tool *fibex2sip* is an Info file with the Fileident *CarMaker-SvcParameters*.

Header information

Fileident = CarMaker-SvcParameters Ver

Identifies the info file as SOME/IP service parameters file of version *Ver*. Currently supported value for *Ver* is 1.

Description:**Description of file type and contents**

Short description about the type of info file and its intended use.

Example Description:

This file was automatically generated by fibex2sip 1.0 for:
CarMaker/HIL – Version 5.0
SOME/IP Rest Bus Simulation

Created = %Y/%m/%d %H:%M:%S user@host

Creation information about when the file was generated, the *user* and the *host* name.
The format of the date and time stamp is as follows:

%Y	Four digit calendar year of the current date
%m	Two digit month of the year with leading zero (01 .. 12)
%d	Two digit day of the month with leading zero (01 .. 31)
%H	Two digit hour of the current time (00 .. 23)
%M	Two digit minute of the current hour (00 .. 59)
%S	Two digit second of the current minute (00 .. 59)

FIBEXdb = Path

Path to the FIBEX file, that was used as base for the import.

ECU descriptions

ECUs = Name[...]

Lists the names of all ECUs (real and simulated) that are part of the rest bus simulation. Each of these ECUs either provides or consumes at least one service instance.

ECU.<Name>.Address: AddrType AddrMode Addr [Mask] [VLAN-Id]

Lists all network addresses (NETWORK-ENDPOINT) of the ECU with name `Name`. Each address is described by its address type `AddrType`, addressing mode `AddrMode` and the address `Addr`. Optionally, an address mask `Mask` and a virtual LAN identifier `VLAN-Id` can be defined, too.

Possible values for `AddrType` are:

- ip: IPv4 address (e.g. 172.16.169.1)
- ip6: IPv6 address (e.g. fe80::c23f:d5ff:fe62:9541)
- mac: MAC address (e.g. 0c:00:01:ab:03:02)

The following addressing modes are supported:

- ucast: Unicast
- mcast: Multicast
- bcast: Broadcast

ECU.<Name>.AppPort: ProtType Port Discovery Serializer Remoting AddrId [Label]

Defines the application ports (APPLICATION-ENDPOINT) of the ECU with name `Name`. An application port is mainly characterized by its protocol `ProtType` (`udp` or `tcp`) and `Port`. The following DISCOVERY-TECHNOLOGY (`Discovery`), SERIALIZATION-TECHNOLOGY (`Serializer`) and REMOTING-TECHNOLOGY (`Remoting`) describe the function of the application endpoint, which is in most cases SOME/IP.

The parameter `AddrId` associates the application endpoint with a network address of the same ECU. The value is an index into the list of defined network addresses (counting starts with 0), as defined with the parameter `ECU.<Name>.Address`.

An optional `Label` helps to identify, if the application port is used for service discovery (SD), or other functions.

Service parameters

Service.Provide = Name[...]

Lists names of services, that are provided by simulated ECUs of the rest bus simulation. Real ECUs can subscribe to events of provided services.
Each service needs to be described in details by `Service<nSvc>`* parameters.

Service.Consume = Name[...]

Lists names of services, that are consumed by simulated ECUs of the rest bus simulation. These services will be consumed from real ECUs by subscription to associated events. Each service needs to be described in details by *Service<nSvc>** parameters.

Service.<n> = Id Name Major Minor InstId Prio Provider

Defines a Service Instance. The services are numbered consecutively, starting with zero. Each service instance is defined by the following parameters:

Id	Service identifier
Name	Name of the service
Major	Major version number
Minor	Minor version number
InstId	Instance Id of service instance
Prio	Priority
Provider	Name of service provider (ECU)

Service.<n>.Provider = Name PortId CycDelay QRDelay [TTL [MinDelay MaxDelay [MaxRep [RepDelay]]]]

Specifies the provider of the service and parameters for service discovery:

Name	Name of service provider (ECU)
PortId	Index of application port
CycDelay	Delay for cyclic announce during main phase [ms]
QRDelay	Delay between query and response [ms]
TTL	Time To Live
MinDelay	Minimum delay for initial announce [ms]
MaxDelay	Maximum delay for initial announce [ms]
MaxRep	Maximum number of repetitions for initial announce
RepDelay	Delay between two repeated announce messages [ms]

**Service.<n>.Consumer:
Name PortId [TTL [MinDelay MaxDelay [MaxRep [RepDelay]]]]**

Specifies the consumers of the service and parameters for service discovery:

Name	Name of consumer (ECU)
PortId	Index of application port
TTL	Time To Live
MinDelay	Minimum delay for initial query [ms]

MaxDelay	Maximum delay for initial query [ms]
MaxRep	Maximum number of repetitions for initial query
RepDelay	Delay between two repeated queries [ms]

Events

Service.<n>.Events:

Id	Name	[CycleTime]
-----------	-------------	--------------------

Lists all events of a service instance:

Id	Event identifier
Name	Name of event
CycleTime	Cycle time of event [s]

Service.<n>.Event.<nEv>.Inputs:**Service.<n>.Event.<nEv>.Returns:**

Pos	Name	ByteOrder	Unit	DataType	[Signal]
Pos	Name[Dim,Min,Max]	ByteOrder	Unit	DataType	[Signal]

Lists all input / return parameters of an event. The event number *nEv* refers to an event, listed with *Service.<n>.Events*, whereas *nEv* counts consecutively from zero.

Each input /return parameter is defined by the following parameters:

Pos	Position of parameter
Name	Name of input / return parameter
Dim	Definition of array dimension (multi-dimensional parameters)
Min	Minimum array size (multi-dimensional parameters)
Max	Maximum array size (multi-dimensional parameters)
ByteOrder	Byte order of parameter (possible values: msb lsb)
MinDelay	Minimum delay for initial announce [ms]
Unit	Unit of parameter (use “-” if no unit, or with complex data types)
DataType	Data type of parameter (see Data type definitions for details)
Signal	Name of signal, which might be connected to that parameter

Fields

Service.<n>.Fields:

Name	NotId GetId SetId Acc[:CT]	BO Unit dType	[Sig]
Name[Dim,Min,Max]	NotId GetId SetId Acc[:CT]	BO Unit dType	[Sig]

Defines the fields of a service instance. The fields are listed consecutively, each text line defines one field.

Each field is defined by the following parameters:

Name	Name of the field
Dim	Definition of array dimension (multi-dimensional fields)
Min	Minimum array size (multi-dimensional fields)
Max	Maximum array size (multi-dimensional fields)
NotId	Notification Identifier
GetId	Getter Identifier (Method Id)
SetId	Setter Identifier (Method Id)
Acc	Access Permissions
CT	Cycle Time
BO	Byte order of field (possible values: msb lsb)
Unit	Unit of field (use “-” if no unit, or with complex data types)
dType	Data type of field (see Data type definitions for details)
Sig	Name of signal, which might be connected to that field

Event Groups

Service.<n>.EventGroups:

Id	EvGroup
----	---------

Defines event groups of a service. Each event group is described by its `Id` and name (`EvGroup`).

The signal is referred by the counter `m`, indicating the `m`-th line of the `Service.<n>.Signals` text key.

Service.<n>.EventGroup.<m>:

Type	EvGrpMbr
------	----------

Specifies the members (events) of the event group with index `m`. The event group referred by the counter `m`, indicating the `m`-th line of the `Service.<n>.EventGroups` text key. Each member of the event group is defined by the `Type` (`event` | `field`) and the name `EvGrpMbr`.

Data type definitions

According to the AUTOSAR SOME/IP specification, the following common data types are supported:

bool	Boolean (8 bits, “false” = 0, “true” != 0)
uint8	unsigned integer, 8 bits
sint8	signed, 8 bits
uint16	unsigned integer, 16 bits
sint16	signed integer, 16 bits
uint32	unsigned integer, 32 bits
sint32	signed integer, 32 bits
uint64	unsigned integer, 64 bits
sint64	signed integer, 64 bits
float32	single precision floating point, 32 bit
float64	double precision floating point, 64 bit

Complex data types (enums, typedefs, structs, unions) are defined with `DataType` keys, followed by hierarchical package notation, separated with dots “.”.

DataType.<Pkg>[.<SubPkg>].<Name> = enum dType ByteOrder Unit

DataType.<Pkg>[.<SubPkg>].<Name>.Enum:

Value	Description
-------	-------------

Defines an enumeration data type with `Name`, as part of the package `Pkg.SubPkg`. The name and structure of the package description is of variable length and depth. The number of dots is not evaluated. Fields and input / return parameters use always the full package path, e.g. `Pkg.SubPkg.Name` in order to reference the data type.

The data type `dType` of the enum must be a common integer data type (e.g. `uint8`).

`Value` and `Description` pairs can be defined with a followed text key, being extended by the keyword `Enum` (`DataType.<Pkg>[.<SubPkg>].<Name>.Enum`).

DataType.<Pkg>[.<SubPkg>].<Name> = complex typedef

DataType.<Pkg>[.<SubPkg>].<Name>.TypeDef = dName ByteOrder Unit dType

Definition of a complex data type `Name` as `typedef` named `dName`. The original data type is referenced with `dType`, which can either be a common data type, or another complex data type.

DataType.<Pkg>[.<SubPkg>].<Name> = complex struct

DataType.<Pkg>[.<SubPkg>].<Name>.Struct:

Pos	MbrName	ByteOrder	Unit	DataType
Pos	MbrName[Dim,Min,Max]	ByteOrder	Unit	DataType

Definition of complex data type, here: structure with members `MbrName` at position `Pos` and individual data type `DataType`. `DataType` can either be a common data type, or another complex data type.

DataType.<Pkg>[.<SubPkg>].<Name> = complex union
DataType.<Pkg>[.<SubPkg>].<Name>.Union:

UnId	UnName	ByteOrder	Unit	DataType
	UnName[Dim,Min,Max]	ByteOrder	Unit	DataType

Definition of complex data type, here: union with members `UnName` with identifier `UnId` and individual data type `DataType`. `DataType` can either be a common data type, or another complex data type.

13.4.2 SIPParameters

The SOME/IP rest bus simulation is configured with an Info file of type *SIPParameters*. This type of file can list sets for one or more rest bus simulations. Parameter keys for rest bus simulations must have the prefix `SIP.<nSIP>`, whereas `nSIP` consecutively numbers the configurations, starting with 0.

Header information

FileIdent = CarMaker-SIPParameters **Ver**

Identifies the info file as SOME/IP parameters file of version `Ver`. Currently supported value for `Ver` is 1.

Configuration parameters

SIP.<nSIP>.ConfigFile = **Path**

Specifies the path to a *SvcParameters* info file, created by the CarMaker/HIL FIBEX import tool (*fibex2sip*). This file will be used as source for service definitions.

SIP.<nSIP>.IfName = **IFace**

Defines, which network interface should be used for the communication.

SIP.<nSIP>.Prefix = **Prefix**

Defines the prefix to be used when declaring Data Dictionary variables for SOME/IP signals and other informations. If no prefix is defined, "SIP" will be used as default.

SIP.<nSIP>.InitTxDataValue = **Val**

For those parts of the payload in a SOME/IP message which are not occupied by signals, this parameter defines what should be used as default raw value. The default value is 0.

SIP.<nSIP>.SigState.None =	Val
SIP.<nSIP>.SigState.Valid =	Val
SIP.<nSIP>.SigState.Invalid =	Val
SIP.<nSIP>.SigState.Unavailable =	Val
SIP.<nSIP>.SigState.Undefined =	Val
SIP.<nSIP>.SigState.Error =	Val

These parameters allow to customize the possible values of state variables, that indicate the different signal states. Possible are values in the range of -128 and 127:

Signal state	Value of state variable, if Signal is
None	of unknown state (default: 0)
Valid	of valid state (default: 1)
Invalid	of invalid state (default: 2)
Unavailable	unavailable (default: 3)
Undefined	undefined (default: 4)
Error	in error state (default: 5)

SIP.<nSIP>.DictDefFields = **bool**

If set to 0 (false), fields will not be added to the Data Dictionary. By default, this parameter is 1 (true), i.e. fields will be always added to the Data Dictionary.

SIP.<nSIP>.DictDefEvents = **bool**

Defines, if events input / return parameters should be added to the Data Dictionary. The default is 1 (true).

SIP.<nSIP>.DictFilter: **EventName** **ServiceName**

Defines, which services or events should not be added to the Data Dictionary. Wildcards are allowed.

Tx- / Rx- Signal Mappings

SIP.<nSIP>.TxMap.<SvcName>:		
SIP.<nSIP>.RxMap.<SvcName>:		
<Event>.<Param>[.Member...]	MapVar	[Factor [Offset [Min [Max]]]]
<Field>[.Member...]	MapVar	[Factor [Offset [Min [Max]]]]
<Event>.<Param>[.Member...]	Const	Value
<Field>[.Member...]	Const	Value

This defines a mapping to a Data Dictionary variable or to a constant value. The Data Dictionary variable must exist. Ideally, the SOME/IP signal and the Data Dictionary variable have the same unit. If not, optional parameters can be specified for conversion or signal conditioning. If a signal shall be mapped to a constant value, the keyword *Const* is to be used instead of the name of a Data Dictionary variable followed by the desired value:

SvcName	Name of Service
Event	Name of Event
Param	Name of Input parameter of event
Member	Hierarchical, dot separated name of signal
Field	Name of Field
Factor	Factor for conversion, optional
Offset	Offset for conversion, optional
Min	Minimum possible value, optional
Max	Maximum possible value, optional
Const	Keyword, indicating mapping to constant value
Value	Desired value for mapping to constant

Rolling Counters

SIP.<nSIP>.TxMap.<SvcName>: SIP.<nSIP>.RxMap.<SvcName>: <Event>.<Param>[.Member...] <Field>[.Member...]	RollCnt	Std	[Min [Max [Incr]]] [Args]
--	---------	-----	------------------------------

Defines a signal to be a message counter. Following types are supported:

SvcName	Name of Service
Event	Name of Event
Param	Name of Input parameter of event
Member	Hierarchical, dot separated name of signal
Field	Name of Field
RollCnt	Keyword to define message counter signal
Std	Standard Rolling Counter: <ul style="list-style-type: none">starts with minimum value (default 0)increments with each sent messagewraps back to 0 after maximum reached
Min	Minimum value for Standard Rolling Counter, optional
Max	Maximum value for Standard Rolling Counter, optional
Incr	Step size (increment) for Standard Rolling Counter, optional
Type	User defined keyword for user defined Counter function
Args	Optional / possible arguments for user defined Counter function

User defined Rolling Counter functions must have been registered with the function [SIP_Register_MapFunc\(\)](#).

Chapter 14

CCP / XCP

Modern HIL test benches provide a complete virtual environment to intelligent ECUs, like an ESP controller, etc. Typically, such an ECU cannot distinguish anymore, whether it is mounted in a real car, or if it is just laying around on a table.

On the other hand, an ECU which controls certain functions of a car, implements complex algorithms and runs more or less autonomously. Testing its internal functions, without knowing anything about its internal states and variables, is nearly impossible.

Based on this problem, two standards have been developed to provide an interface to the internals of an intelligent ECU, allowing access to internal parameters, variables and states. These are the *CAN Calibration Protocol* (CCP) and its successor, the *Universal Measurement and Calibration Protocol* (XCP). While CCP is exclusively designed for use with the CAN protocol, the XCP standard supports many different transport layers, like CAN, Ethernet (UDP/IP and TCP/IP), SxI (SPI, SCI), FlexRay or USB.

Typically, CCP or XCP is part of the on-board measurement and diagnostics system in a prototype vehicle. It is used to measure a set of parameters or variables during a TestRun. The measurement data is usually recorded for being analyzed in an additional post processing step after the TestRun.

Now, since we are able to transfer a real world TestRun seamlessly into a virtual world, while the tested ECUs are not able to notice a difference, why not using the CCP or XCP capabilities of an ECU within virtual tests in CarMaker? A number of applications arise from this opportunity. We would be able to compare and check internal vehicle dynamics with appropriate CarMaker model variables, or even use the measurement data online within the CarMaker simulation.

14.1 Overview

CCP, the *CAN Calibration Protocol*, was originally developed by “Ingenieurbüro Helmut Kleinknecht” (now Kleinknecht Automotive GmbH – <http://www.kleinknecht.com>). Later it was assumed by the ASAM group (<http://www.asam.net>) and extended with additional optional functions.

XCP, the *Universal Measurement and Calibration Protocol*, is based on the experiences with CCP, and is meant to be an improved and generalized version of CCP. The main difference is the separation of protocol- and transport layer, allowing XCP to be used not only with CAN, but also with Ethernet (TCP/IP and UDP/IP), FlexRay, SxI (SPI, SCI), USB and potentially any other serial transfer protocol.

Both, CCP and XCP, implement a master slave communication between a service tool (master) and an ECU (slave). They are used during the development and test of ECUs for motor vehicles, to allow on-board tests and measurements in pre-production cars.

This chapter explains how the CCP protocol and the XCP protocol have been integrated in CarMaker/HIL and describes the available functions within the CarMaker GUI and Script-Control.

14.1.1 Basic information

The CCP and XCP implementation in CarMaker/HIL follows the CCP specification version 2.1 and XCP specification version 1.0. But, since the main focus is on data acquisition, not all features of the specifications are implemented. Especially those sets of commands, which allow to flash new ECU software, are not provided to the end user. However, most of the commands can still be implemented with the help of the basic function to send and receive arbitrary CCP / XCP commands.

For the configuration of the master, ASAP2 files are used, the common description format for protocol parameters, ECU parameters, variables and conversion functions. With the help of this file, CarMaker is able to supply an easy to use interface to configure, start and stop data acquisitions.

The current implementation of XCP in CarMaker/HIL allows to use XCP with CAN, Ethernet and FlexRay.

CCP Communication Model

The communication model of CCP defines two sets of commands:

- A generic set of commands for the connection handling, calibration, programming, etc.
- DAQ (data acquisition) commands for data measurement

The CCP slave receives commands from the Calibration Tool by so called *Command Receive Objects* (CRO) and replies with *Data Transmit Objects* (DTO).

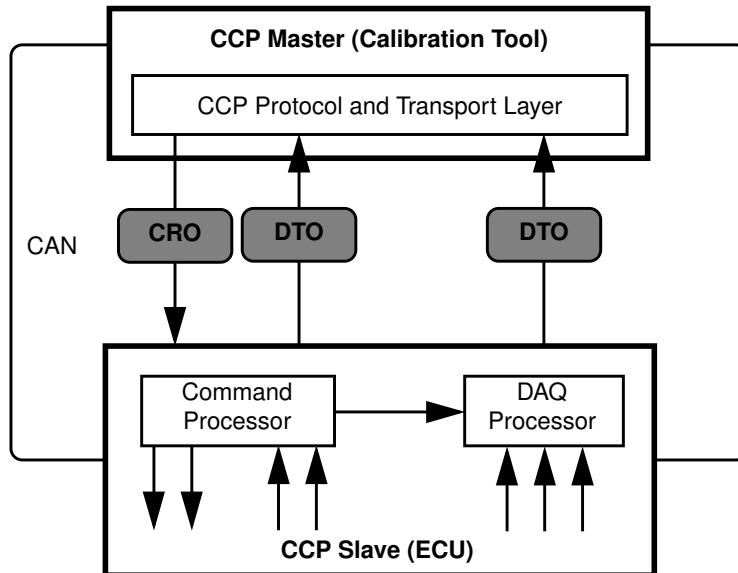


Figure 14.1: CCP Communication Model

When receiving DAQ commands, the Command Processor of the ECU programs the DAQ Processor for the transmission of requested DAQ lists and defined data rate.

A DAQ list comprises one or more *Object Descriptor Tables* (ODT) which need to be transmitted at the same data rate. The data elements of an ODT are the measured quantities, identified by their address. Each ODT is identified by a unique *Process Identifier* (PID) and transmitted in a single CAN frame:

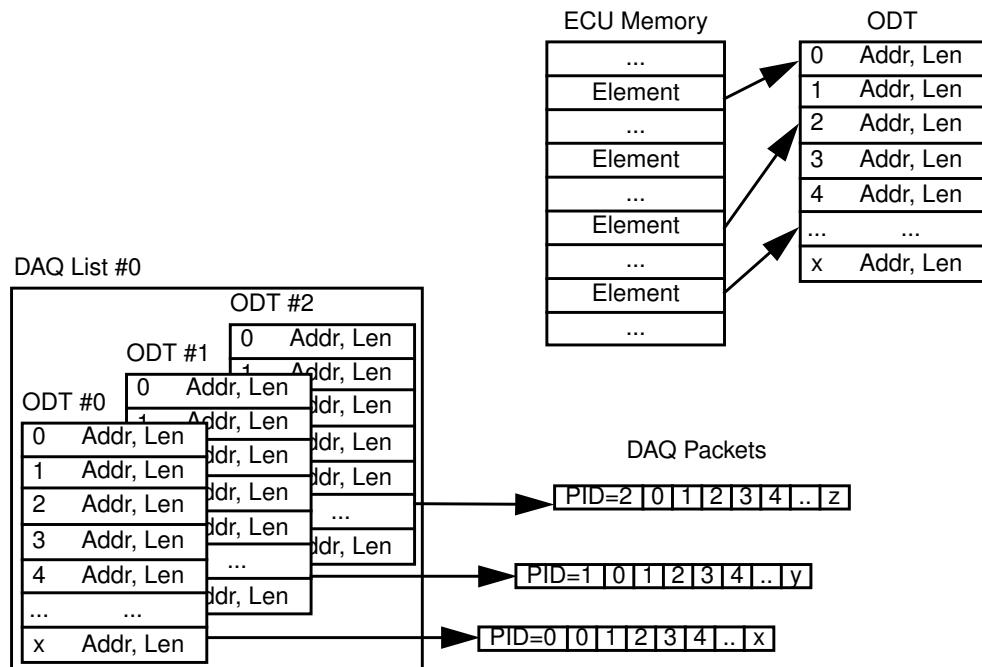


Figure 14.2: DAQ list

Once the DAQ processor is programmed, data measurement can be started. The DAQ processor will then continuously send DAQ packets with the requested data rate.

XCP Communication Model

The specification of the XCP protocol extends and subdivides the two sets of CCP commands into five groups:

- Standard commands (STD)
- Calibration commands (CAL)
- Page switching commands (PAG)
- Data acquisition commands (DAQ)
- Programming commands (PGM)

Similar to CCP, XCP also defines two packet types:

- CTO packets for transferring control commands (*Command Transmit Object*)
- DTO packets for synchronous data (*Data Transmit Object*)

A CTO packet can be a command request from the master to the slave (CMD), a command response (RES), an error packet (ERR), an event packet (EV), or a service request (SERV). During a measurement, the slave usually sends DAQ packets (DTO) continuously to the master. However, the XCP standard also defines a special stimulation packet (STIM), which can be send by the master to request a specific DAQ packet.

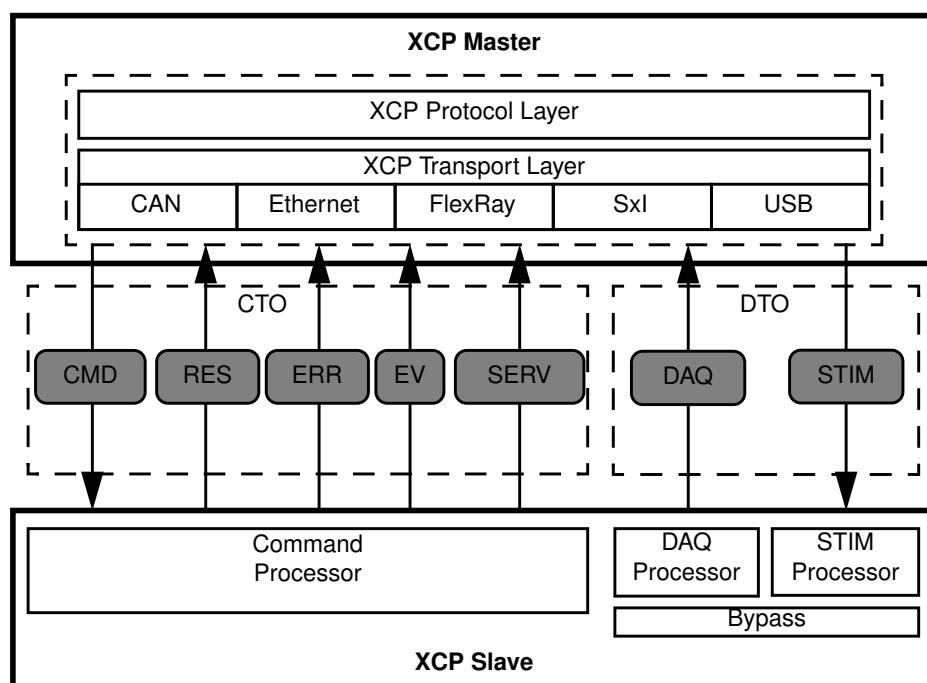


Figure 14.3: XCP Communication Model

DAQ lists are defined – similar to CCP – by ODTs (*Object Descriptor Table*) and single data elements. Unlike to CCP, where the size of an ODT is limited to seven bytes (in order to fit into a CAN data frame), XCP allows numerous different ways to pack DAQ data into DTO packets. The simplest packet format – which is usually used for XCP on CAN – is equal to the format used with CCP (s. [Figure 14.2](#)). Depending on the transport layer, XCP supports also:

- Timestamps
- DAQ identifier
- Variable size of ODTs (up to 254 entries of variable size)
- Variable number of ODTs (up to 252 per DAQ list)

- Variable number of DAQ lists (up to 65536)

With CarMaker/HIL, all types of DAQ lists are supported, fixed as well as dynamic. All protocol parameters, like message format, type of DAQ lists, available events, possible data rates etc. are imported from the appropriate ASAP2 file.

14.1.2 Features

The main focus of the CCP and XCP implementation in CarMaker/HIL is the measurement of data. This allows to watch measurement data online and to evaluate it right away together with other model variables. Saving the measurement data to disk for a post-processing step is possible with the data storage mechanism of CarMaker, but not necessary in many cases (unlike to typical test scenarios with real cars).

An easy to use interface has been developed and integrated in the CarMaker GUI, which allows to select available parameters and assign to up to 6 sample groups. Each sample group can be configured with its own sample rate. A measurement can then be started either manually, or automatically with a TestRun.

In order to program DAQ lists for data measurement, a number of commands have been used. Generally, it is expected, that the connected ECU implements the set of mandatory DAQ commands according to the CCP and/or XCP specification. Additionally – if supported by the ECU – also some optional commands might be used.

Besides the DAQ programming, any other command can be send to a connected ECU, either with the “Send CRO/CTO” button in the GUI, or via ScriptControl.

The set of implemented and used commands are listed below.

Used CCP commands

Table 14.1: Used CCP commands

Id	Name	Description
0x01	CONNECT	Open CCP connection to slave
0x06	START_STOP	Start / Stop data transmission
0x07	DISCONNECT	Close CCP connection to slave
0x08	START_STOP_ALL	Start / Stop synchronized data transmission
0x14	GET_DAQ_SIZE	Get size of DAQ list
0x15	SET_DAQ_PTR	Set DAQ list pointer
0x16	WRITE_DAQ	Write DAQ list entry

The current implementation of CCP does not use the GET_SEED / UNLOCK commands and does not provide an automated mechanism to read and set parameters via UPLOAD and DNLOAD.

Used XCP commands

Table 14.2: Used XCP commands

Id	Name	Description
0xff	CONNECT	Open XCP connection to slave
0xfe	DISCONNECT	Close XCP connection to slave
0xfd	GET_STATUS	Get current session status from slave
0xfc	SYNCH	Synchronize command execution
0xf8	GET_SEED	Get seed for unlocking a protected source
0xf7	UNLOCK	Send key for unlocking a protected source
0xf6	SET_MTA	Set Memory Transfer Address in slave
0xf5	UPLOAD	Upload from slave to master
0xf2	TRANSPORT_LAYER_COMMAND	Transport layer specific command (used for XCP on FlexRay only)
0xf0	DOWNLOAD	Download from master to slave
0xe3	CLEAR_DAQ_LIST	Clear DAQ list configuration
0xe2	SET_DAQ_PTR	Set pointer to ODT entry
0xe1	WRITE_DAQ	Write element in ODT entry
0xe0	SET_DAQ_LIST_MODE	Set mode for DAQ list
0xde	START_STOP_DAQ_LIST	Start / Stop / Select DAQ list
0xdd	START_STOP_SYNCH	Start / Stop DAQ lists (synchronously)
0xda	GET_DAQ_PROCESSOR_INFO	Get general information on DAQ processor
0xd9	GET_DAQ_RESOLUTION_INFO	Get general information on DAQ processing resolution
0xd8	GET_DAQ_LIST_INFO	Get specific information for a DAQ list
0xd7	GET_DAQ_EVENT_INFO	Get specific information for an event channel
0xd6	FREE_DAQ	Clear dynamic DAQ configuration
0xd5	ALLOC_DAQ	Allocate DAQ lists
0xd4	ALLOC_ODT	Allocate ODTs to a DAQ list
0xd3	ALLOC_ODT_ENTRY	Allocate ODT entries to an ODT

Table 14.3: Used Transport Layer Sub-Commands with XCP on FlexRay

Sub-Id	Name	Description
0xff	FLX_ASSIGN	Assign / Deassign FlexRay LPDU-Identifiers to buffers
0xfe	FLX_ACTIVATE	Activate Communication of a FlexRay buffer

The GET_SEED / UNLOCK commands are used for the UPLOAD and DOWNLOAD commands to read and set parameters on the slave. To compute the key from the requested seed, a Seed & Key Library can be included according the requirements of the XCP specification. Depending on the operating system, this would be either a Windows-DLL, or a Linux Shared Object.

Storage of Measurement data

Optionally, the measurement data of XCP measurements can be saved into a MDF file for post-processing with other measurement and analysis tools. The produced MDF files are compliant to the MDF standard version 4.1.0, specified by the ASAM Group (<http://www.asam.net>). All MDF files have the following structure:

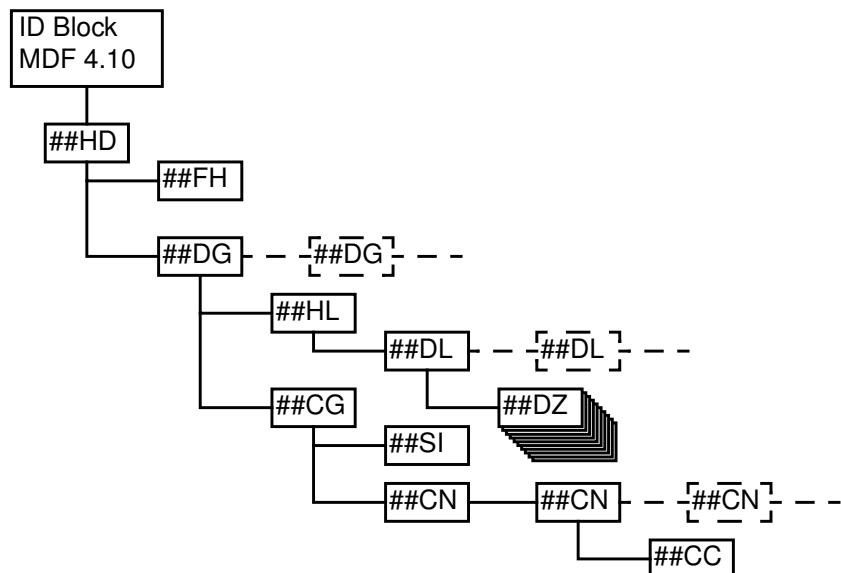


Figure 14.4: Structure of MDF file

- If uncompressed output format is desired, the “##HL” block is optimized out.
- Each DAQ list is represented by one single “##DG” block.
- The first channel (“##CN”) of a DAQ list is always a time stamp, stored as 4-byte unsigned Integer (UINT32). The resolution depends on the ECU. If the ECU does not support the transmission of time stamps, an internal time stamp with a resolution of 1 micro second is generated.
- The time stamp channel of a DAQ list will be named always “DAQ.<n>.Timestamp”, whereas <n> represents the DAQ list number.
- The time stamp channel of a DAQ list is marked as master channel (`cn_type = 2`) for time synchronization (`cn_sync_type = 1`). In order to meet the requirements for this type of synchronization channel (SI unit “s”, physical values relative to start value in “##HD” block), a “##CC” block of linear conversion type is added to the channel group.
- Data blocks (“##DZ”, “##DT”) are of equal length. The length of data blocks is always a multiple of the size of the assigned DAQ list (including time stamp). The absolute size varies, but is chosen to hold as many samples as possible that fit into a block with a size of 1 MB.
- All data lists (“##DL”) reference 10 data blocks (“##DZ” or “##DT”). The last data list may reference less than 10 data blocks.
- All data groups (“##DG”) are sorted.
- Each “##CN” block represents a measurement quantity (^{MEASUREMENT}) as specified in the ASAP2 file. The associated `COMPU_METHOD` is represented by a “##CC” block.

14.1.3 Limitations

With CarMaker/HIL for dSPACE (Processor boards DS1005 and DS1006), XCP is only available for the transport layer CAN.

14.2 Architectural Overview

14.2.1 Overview of the CCP protocol integration into CarMaker

The following scheme shows how the CCP protocol has been integrated into CarMaker:

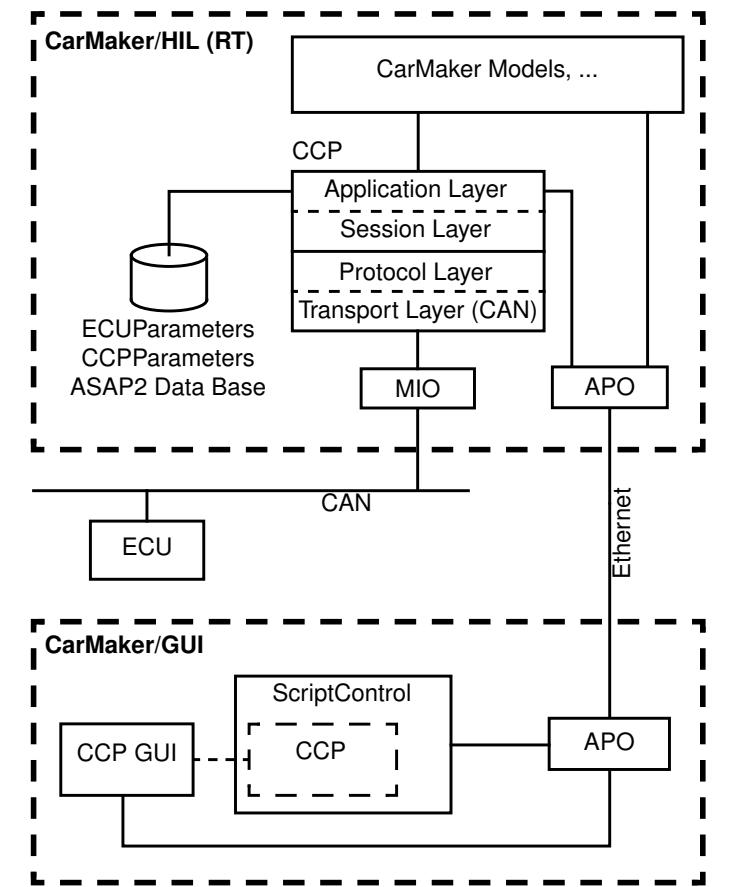


Figure 14.5: How CCP fits into simulation framework of CarMaker/HIL

14.2.2 Overview of the XCP protocol integration into CarMaker

The following scheme shows how the XCP protocol has been integrated into CarMaker:

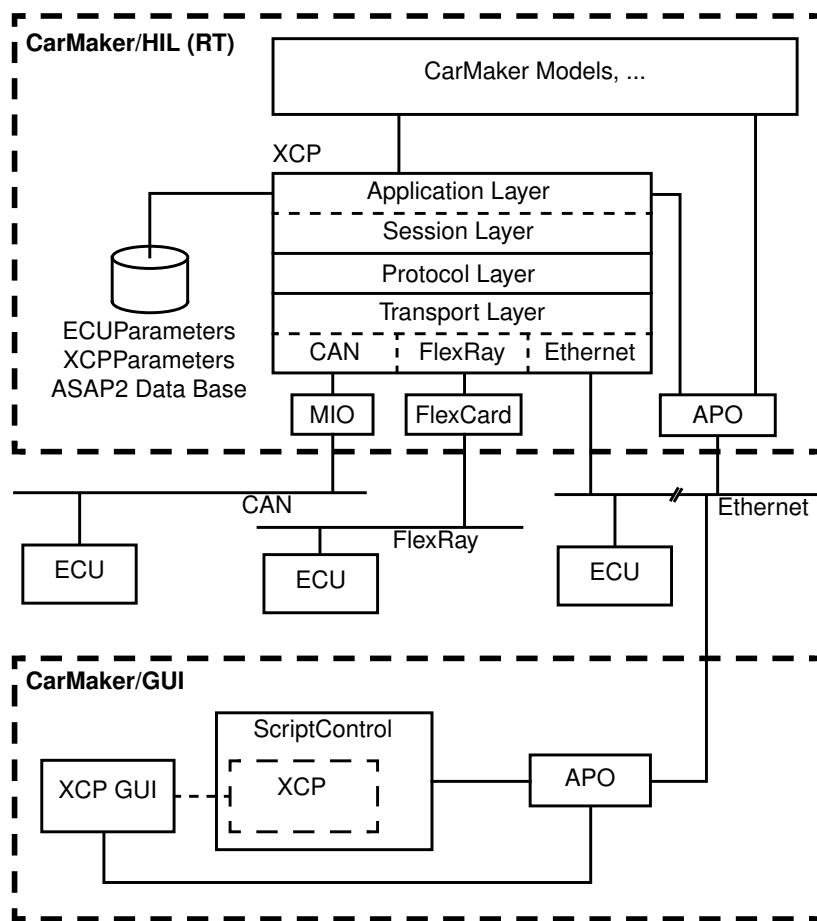


Figure 14.6: How XCP fits into simulation framework of CarMaker/HIL

14.3 Integration in a CarMaker Project

The CarMaker project template shipped with CarMaker/HIL is already prepared for use with CCP and XCP. However, CCP and XCP are not enabled by default but must be activated through preprocessor macros in the *Makefile*. The CarMaker real-time application must then be rebuilt.

The activation of CCP/XCP in a CarMaker project can be divided into 3 steps:

- Modifying source files and rebuilding CarMaker executable (s. [section 14.3.1](#))
- Tuning the CCP / XCP GUI (s. [section 14.3.2](#))
- Configuring the CCP / XCP communication module (s. [section 14.3.3](#))

Optionally, measurement quantities can also be mapped to CarMaker variables, to be used e.g. as input for customized models. This is described in [section 14.3.4](#).

14.3.1 (Re-)Building the CarMaker Executable

The *Makefile* of a CarMaker/HIL project template is already prepared for the activation of CCP and XCP. In order to activate CCP or XCP, you just have to comment out one (or both) of the following lines:

Listing 14.1: Activation of CCP / XCP in *Makefile*

```
1: ### XCP / CCP
2: #DEF_CFLAGS += -DWITH_XCP
3: #DEF_CFLAGS += -DWITH_CCPC
```

CCP and XCP can be activated and used simultaneously. There is no general limitation, which would not allow the two protocols to be used in parallel.

After the activation of CCP or XCP in the *Makefile*, you have to make sure, that all object files (*CM_Main.o*, *CM_Vehicle.o*, *User.o*, *IO.o*) will be truly rebuilt (e.g. by using *make clean*). Otherwise, the CCP or XCP feature might stay inactive. Normally, there is no need for any further modifications to any other source files, since *CM_Main.c* and *IO.c* are already prepared with the necessary blocks of code. However, the following pages explain the CCP and XCP specific lines of code for advanced users.

The CCP and the XCP module provide a very similar C-function interface, defined in the header files *CM_CCP.h* and *CM_XCP.h*. For the integration into CarMaker/HIL, the required set of function calls is nearly the same. They only differ in the prefix, which is *CM_CCP* for CCP functions, and *CM_XCP* for XCP functions. Therefore, we limit ourselves on the following description to the case of using XCP.

Modifications to CM_Main.c

Most of the XCP functionality is handled in the file *CM_Main.c*. XCP specific code sequences are located among include instructions for header files and in the functions *App_Init_First()*, *App_Init()*, *App_DeclQuants()*, *App_TestRun_Start()*, *MainThread_BeginCycle()*, *App_TestRun_Calc()*, *ProcessApoMessages()*, *MainThread_FinishCycle()*, *App_End()* and *App_Cleanup()*.

Including XCP header

First, the header file `CM_XCP.h` has to be included, ideally before the include instruction for `User.h`, but after `IO.h`:

Listing 14.2: Including XCP header

```

1: ...
2: #include "IO.h"
3: ...
4: #if defined(WITH_XCP)
5: # include <CM_XCP.h>
6: #endif
7: #include "User.h"
8: ...

```

`App_Init_First()`

A very first initialization of the XCP module takes place in the function `App_Init_First()`:

Listing 14.3: `App_Init_First()` with XCP

```

1: static int
2: App_Init_First (int argc, char **argv)
3: {
4:     ...
5:     User_Init_First ();
6: #if defined(WITH_XCP)
7:     CM_XCP_Init_First(NULL);
8: #endif
9:     return 0;
10: }

```

At this point of time, the XCP module needs to know how to send CAN messages. The function `CM_XCP_Init_First()` expects a single parameter `CAN_Send`. This can be a pointer to a customized CAN send function, which meets the following function declaration:

```
int CAN_Send(int, int, const struct CAN_Msg *);
```

Here, a NULL pointer is passed, which tells the XCP module to use the system default `MIO_M51_Send()` (or `DSIO_CAN_Send()` on CarMaker/HIL for dSPACE).

`App_Init()`

The main initialization takes place in the function `App_Init()`. The return value of `CM_XCP_Init()` is always checked. If the return value is not zero, the initialization of the application fails (FATAL ERROR):

Listing 14.4: `App_Init()` with XCP

```

1: static int
2: App_Init (void)
3: {
4:     ...
5:     if (IO_Init() < 0 || User_Init() < 0)
6:         return -1;
7: #if defined(WITH_XCP)
8:     if (CM_XCP_Init() != 0)
9:         return -1;
10: #endif
11:     ...
12: }

```

App_DeclQuants()

In order to allow, that all available measurement quantities will be also added to the Data Dictionary, `CM_XCP_DeclQuants()` is called in the function `App_DeclQuants()`:

Listing 14.5: `App_DeclQuants()` with XCP

```

1: static int
2: App_DeclQuants (void)
3: {
4:     ...
5:     User_DeclQuants ();
6: #if defined(WITH_XCP)
7:     CM_XCP_DeclQuants ();
8: #endif
9:     ...
10: }
```

App_TestRun_Start()

Communication parameters, ECU definitions and protocol parameters are read through the function `CM_XCP_Param_Get()`, which is called in `App_TestRun_Start()`, whenever the `ECUParameters` file has been changed:

Listing 14.6: `App_TestRun_Start()` with XCP

```

1: static void *
2: App_TestRun_Start (void *arg)
3: {
4:     ...
5:     if (SimCore.TestRig.ECUParam.WasRead) {
6: #if defined(WITH_XCP)
7:         if (CM_XCP_Param_Get(SimCore.TestRig.ECUParam.Inf, "XCP") != 0)
8:             rv = -6;
9: #endif
10:    }
11:    ...
12:    if (User_TestRun_Start_atEnd() < 0) {
13:        rv = -15;
14:        goto ErrorReturn;
15:    }
16: #if defined(WITH_XCP)
17:     if (CM_XCP_TestRun_Start(NULL, NULL) != 0) {
18:         rv = -16;
19:         goto ErrorReturn;
20:     }
21: #endif
22:     ...
23: }
```

The function `CM_XCP_Param_Get()` expects two arguments: an info file handle (a pointer of type: `struct tInfos *`), and a prefix for all read keys. If the prefix is not `NULL`, all XCP configuration parameters, which are read from the given info file will be appended to the given prefix.

If there is a data measurement configured in the CarMaker GUI, a data acquisition will be started automatically as soon as a new TestRun is started. The function `CM_XCP_TestRun_Start()`, called after `User_TestRun_Start_atEnd()`, will reread the XCP configuration (if changes were detected) and prepare the start for the data acquisition by adding measurement quantities to the Data Dictionary.

The Data Dictionary entries will be named according to the scheme

`<Prefix>.<ModuleName>.<QuName>`

with the following meanings:

- **Prefix:** Prefix, as defined in XCPParameters file.
- **ModuleName:** Name of the module section(s), as defined in the ASAP2 file.
- **QuName:** Name of the measurement variable, as defined in the ASAP2 file.

MainThread_BeginCycle()

XCP packets are received and processed in the function `MainThread_BeginCycle()`. `CM_XCP_In()` is called before `User_In()`:

Listing 14.7: `MainThread_BeginCycle()` with XCP

```

1: static int
2: MainThread_BeginCycle (unsigned CycleNo)
3: {
4: ...
5:     /*** Input from hardware */
6:     IO_In(CycleNo);
7: #if defined(WITH_XCP)
8:     CM_XCP_In();
9: #endif
10:    ...
11:    DVA_HandleWriteAccess(DVA_IO_In);
12:    User_In(CycleNo);
13: ...
14: }
```

App_TestRun_Calc()

In `App_TestRun_Calc()`, the function `CM_XCP_Calc()` is called. The return value indicates, if a requested measurement is already started, or not. During the preparation phase of a TestRun, this information is used to delay the start of the simulation, until a configured data measurement is started (i.e. `SimCore.Start.IsReady` is kept zero):

Listing 14.8: `App_TestRun_Calc()` with XCP

```

1: static int
2: App_TestRun_Calc (double dt)
3: {
4: ...
5:     if (User_Calc(dt) < 0)                  /* optional */
6:         rv = -4;
7: #if defined(WITH_XCP)
8:     if (CM_XCP_Calc() != 0) {
9:         if (SimCore.State == SCState_StartSim)
10:             SimCore.Start.IsReady = 0;
11:     }
12: #endif
13: ...
14: }
```

ProcessApoMessages()

In the function `ProcessApoMessages()`, all received APO messages are evaluated. For XCP specific commands, `CM_XCP_ApoMsg_Eval()` is called:

Listing 14.9: `ProcessApoMessages()` with XCP

```

1: static void
2: ProcessApoMessages (void)
3: {
4:     int ch, len, who;
5:     char MsgBuf[APO_ADMMAX];
6:
7:     while (len==sizeof(MsgBuf), AposGetAppMsgFrom(&ch,&MsgBuf,&len,&who) != 0) {
8: #if defined(WITH_XCP)
9:         if (CM_XCP_ApoMsg_Eval(ch, MsgBuf, len) >= 0)
10:             continue;
11: #endif
12:         if (User_ApoMsg_Eval(ch, MsgBuf, len, who) < 0
13:             && SimCore_ApoMsg_Eval(ch, MsgBuf, len, who) < 0) {
14:             SimCore_UnknownApoMsgWarn(ch, MsgBuf, len);
15:         }
16:     }
17: }
```

The function `CM_XCP_ApoMsg_Eval()` takes three arguments, the APO communication channel of type `int`, a pointer to the received APO message (type `char *`) and the length in bytes. Only messages which were received on the channel 6 and start with the character sequence “XCPx” are evaluated. In this case, the return value will be greater or equal to zero. A return value lower than zero means, that the message was not recognized as XCP message and not processed.



In order to prevent conflicts with the CMDIAG module (KWP2000/UDS diagnostics), the function `CM_XCP_ApoMsg_Eval()` is called before `User_ApoMsg_Eval()`. In contrast to the XCP module, the CMDIAG module does not check for a specific header, but will recognize an XCP message as CMDIAG message of unknown format, resulting in a return value greater than zero. If the CMDIAG module comes first, the following test on the return value would result in a lost APO message.

MainThread_FinishCycle()

Communication back to the XCP GUI is managed in the function `CM_XCP_ApoMsg_Send()`. It is called in `MainThread_FinishCycle()`, right after `User_Apo_Msg_Send()`:

Listing 14.10: `MainThread_FinishCycle()` with XCP

```

1: static void
2: MainThread_FinishCycle (unsigned CycleNo)
3: {
4:     ...
5:     User_ApoMsg_Send    (TimeGlobal, CycleNo);
6: #if defined(WITH_XCP)
7:     CM_XCP_ApoMsg_Send(T, CycleNo);
8: #endif
9:     ...
10: }
```

App_End()

In `App_End()`, the function `CM_XCP_End()` is called to terminate the communication to all slaves:

Listing 14.11: `App_End()` with XCP

```

1: static int
2: App_End (void)
3: {
4: #if defined(WITH_XCP)
5:     CM_XCP_End();
6: #endif
7:     User_End ();
8:     ...
9: }
```

App_Cleanup()

When CarMaker/HIL is terminated, the function `App_Cleanup()` calls `CM_XCP_Cleanup()`, in order to free unneeded resources:

Listing 14.12: `App_Cleanup()` with XCP

```

1: static void
2: App_Cleanup (void)
3: {
4:     ...
5: #if defined(WITH_XCP)
6:     CM_XCP_Cleanup();
7: #endif
8:     IO_Cleanup ();
9:     User_Cleanup ();
10:    ...
11: }
```

Modifications to `IO.c`

When using XCP on CAN (or CCP), CAN messages have to be received and evaluated by the XCP module. Normally, the CAN bus will not be used exclusively for XCP. Therefore, the XCP module cannot manage the reception of all CAN traffic on a given CAN interface independently. Instead, CAN messages have to be received from the CAN interface in the Module `IO.c` and should be passed to the XCP module for evaluation. In `IO.c`, changes are required regarding include instructions for header files and to the function `IO_In()`.

Including XCP header

First, the header file `CM_XCP.h` is included, after the include instruction for `IO.h`:

Listing 14.13: Including XCP header

```

1: ...
2: #include "IO.h"
3: ...
4: #if defined(WITH_XCP)
5: # include <CM_XCP.h>
6: #endif
7: ...
```

IO_In()

On reception of a CAN message, the CAN message should be passed to the function `CM_XCP_Dispatch_CAN_Msg()`. If the CAN message is recognized as an XCP message, `CM_XCP_Dispatch_CAN_Msg()` returns zero, otherwise -1:

Listing 14.14: `IO_In()` with XCP

```

1: void
2: IO_In (unsigned CycleNo)
3: {
4:     CAN_Msg    Msg;
5:
6:     IO.DeltaT =  SimCore.DeltaT;
7:     IO.T =      TimeGlobal;
8:
9:     if (IO_None)
10:        return;
11:
12:    /*** FailSafeTester messages */
13:    if (FST_IsActive()) {
14:        while (MIO_M51_Recv(FST_CAN_Slot, FST_CAN_Ch, &Msg) == 0)
15:            FST_MsgIn (CycleNo, &Msg);
16:    }
17:
18:    if (IO_DemoApp) {
19:        /* process Rx CAN messages */
20:        while (1) {
21:            if (MIO_M51_Recv(Slot_CAN, 0, &Msg) != 0)
22:                break;
23: #if defined(WITH_XCP)
24:            if (CM_XCP_Dispatch_CAN_Msg(&Msg) == 0)
25:                continue;
26: #endif
27:        }
28:
29: #if defined(WITH_FLEXRAY)
30:     FC_In(CycleNo);
31:     RBS_In(CycleNo);
32: #endif
33:    }
34: }
```

Compiling CarMaker/HIL

The last step is to compile and link the CarMaker/HIL executable:

- Open a Shell and change to the `src` directory in your CarMaker project (on Windows, you will start the MSYS Development Environment – MSYS 2003):

 > cd /.../CM_Project/CM_XCP_Project/src
- Start the Build process:

 > make clean

 > make

If the compilation step does not report an error, the CarMaker realtime executable is ready to start.

14.3.2 Tuning the CarMaker GUI

Whenever the CarMaker GUI connects to the realtime system, it verifies, if the realtime system is prepared for the use of CCP and/or XCP. Depending on the result, the *CCP Configuration* and/or *XCP Configuration* dialogs are made accessible through the *Realtime System* menu of the main dialog. CCP and/or XCP are then ready for use.

However, when using XCP, the following features are not available or enabled by default:

- XCP calibration access
- ECU Unlock service (Seed&Key)

The following changes can be made to the `.CarMaker.tcl` file in the CarMaker project directory, in order to allow access to calibration functions and/or to enable permanently the automatic ECU Unlock service:

Listing 14.15: Changes to `.CarMaker.tcl` for XCP

```

1: ...
2: ### Universal Calibration Protocol - XCP
3: source XCP_Cal.tcl
4:
5: # Select Seed&Key DLL for XCP (optional)
6: set XCP::XCP(UnlockRequired) 1
7: set XCP::XCP(UnlockDLL)      "bin/SeedKey.dll"
8: ...

```

14.3.3 Configuring the Communication Module

Before you can connect to an ECU using CCP or XCP, the available ECUs, protocol type and other parameters have to be configured, first:

- `bootrc` file `.../xeno_rt/etc/bootrc.rt1` of the realtime system (for XCP on Ethernet).
- `hosts` file `.../xeno_rt/etc/hosts` of the realtime system (for XCP on Ethernet).
- `ECUParameters` for activation of CCP / XCP and to specify the path to `CCPParameters / CCPParameters` file with detailed ECU and protocol definitions.

Hosts and Bootrc file of Realtime System for XCP on Ethernet

When using XCP on Ethernet, the XCP slave ECU accepts connections on a certain IP address, typically 192.168.1.1 – or any other IP address in the range of non routed addresses. In order to have a successful connection, the IP address of the XCP master needs to be in the same sub network.

For best results, it is recommended to use the second (unused) ethernet interface of the realtime system and assign it an IP address in the required range. Supposed to be, the host name of the realtime system is `rt1`, the IP address of the realtime system is 192.168.0.241 and the IP address of the ECU is 192.168.1.1. You need to change the hosts data base file `$IPGHOME/hil/linux-xeno-$CM_VERSION/xeno_rt/etc/hosts` as follows:

Listing 14.16: Changing `.../xeno_rt/etc/hosts` for XCP on Ethernet

```

1: ...
2: 192.168.0.1      srpvpc xenosrv
3: 192.168.0.241    rt1
4: ...
5: 192.168.1.1      xcp-ecu
6: 192.168.1.241    xcp-rt1
7: ...

```

Next, you have to add some lines to the bootrc file for *rt1*. Open an editor and load the file `$IPGHOME/hil/linux-xeno-$CM_VERSION/xeno_rt/etc/bootrc.rt1` (Create it, in case it does not exist):

Listing 14.17: Changing `.../xeno_rt/etc/bootrc.rt1` for XCP on Ethernet

```
1: # configure 2nd ethernet interface for XCP communication
2: ifconfig eth1 xcp-rt1 up
```

Care must be taken, that the IP addresses of the host PC and of the realtime system do not collide with the one of the ECU.



The ECU can also reside in the same physical network as the realtime system and the host PC, as long as the IP address of the ECU does not collide with any other IP address in the network. In this case, only one of the ethernet interface will be used, and there are no changes necessary in the file `bootrc.rt1`.



In case, the ECU has an IP address in a different subnet, but is connected to the same ethernet interface as the rest of the network, a second IP address has to be assigned to this ethernet interface:

Listing 14.18: Changes in `bootrc.rt1` for XCP on Ethernet in same physical network

```
1: # configure 1st ethernet interface for XCP communication
2: ifconfig eth0 add xcp-rt1
```

The realtime system should be rebooted after the changes.

ECUParameters

On CarMaker start-up and with the start of every TestRun, the CCP or XCP configuration needs to be checked for changes and re-read, if necessary. The first file to be checked is the file `ECUParameters`. This file can either list all CCP / XCP configuration parameters directly, with the specified prefix (e.g. “XCP”), as passed to the function `CM_XCP_Param_Get()` (`CM_CCP_Param_Get()`, respectively); or it can specify a file name where to read the ECU, protocol and communication parameters. For example, the following line

```
XCP.XCPParameters = XCPParameters
```

will tell the XCP module to search for the file `XCPParameters` (either in `Data/Config` or in `Data/Misc`), and parse it for XCP parameters. When using CCP, the name of the key would be `CCP.CCPParameters` (with “CCP” as prefix, passed to `CM_CCP_Param_Get()`).

The referenced `XCPParameters` file must then carry an `FileIdent` key with the value “CarMaker-XCPParameters 1”, or “CarMaker-CCPParameters 1” when using CCP.

For a complete reference of possible configuration parameters, refer to [section 14.4.1 ‘CCPParameters’](#) and [section 14.4.2 ‘XCPParameters’](#).

14.3.4 Mapping Measurement Quantities to the CarMaker Model

Measurement quantities can be routed back to the CarMaker model by using a mapping between ASAP2 quantities and C-variables. The CCP and the XCP module provide a special mechanism for this purpose.

The idea is, that if you have a model integrated in CarMaker – e.g. your own brake controller, which takes a special ASAP2 quantity as input, you can switch between different ASAP2 files, without the need to recompile. Since the model stays the same, but only the name of

the ASAP2 quantity might change (because of using another ECU or firmware version), it is preferable to adapt just a configuration file only (s. [section 14.4.5 'ASAP2 variable Mappings'](#)).

In order to use mappings, the module *User.c* has to be modified.

Modifications to User.c

Before you can use a mapping, it has to be created by registering an info file key. This info file key will then be read with other CCP / XCP parameters and used to determine the name of the ASAP2 quantity to map. This needs to be done in the function `User_Register()`. In the function `User_In()`, the actual values can be obtained after `CM_XCP_In()`.

Again, the following description shows the necessary changes for XCP. For CCP, the prefix of the used functions and data type has to be changed from `CM_XCP` to `CM_CCP`.

User_Register()

To register an info file key for an ASAP2 variable mapping, `CM_XCP_A21Var_Register()` needs to be called:

Listing 14.19: `User_Register()` with ASAP2 variable mapping

```
1: #if defined(WITH_XCP)
2: struct tXCP_VarMap *MapWhlSpdFL, *MapWhlSpdFR;
3: #endif
4:
5: int
6: User_Register (void)
7: {
8: #if defined(WITH_XCP)
9:     MapWhlSpdFL = CM_XCP_A21Var_Register("ECU.0.Map.WheelSpeed_FL");
10:    MapWhlSpdFR = CM_XCP_A21Var_Register("ECU.0.Map.WheelSpeed_FR");
11: #endif
12:
13:     return 0;
14: }
```

The function `CM_XCP_A21Var_Register()` uses the Prefix `ECU.<n>` in the given key to find the corresponding ECU. The value of the given info file key is then used to identify the variable in the linked ASAP2 file. If the key was successfully linked to a configured ECU, the mapping will be created and a pointer of type `struct tXCP_VarMap *` will be returned as a handle. This handle is needed for access to the value of the variable. On error, the function returns the `NULL` pointer.

User_In()

During a measurement, all measured quantities are updated with every received DAQ packet within the function `CM_XCP_In()`. The value of a registered mapping, can then be obtained with the function `CM_XCP_A2lVar_In()`:

Listing 14.20: User_In() with ASAP2 variable mapping

```
1: void
2: User_In (const unsigned CycleNo)
3: {
4: ...
5: #if defined(WITH_IO_CAN)
6:     IO_CAN_User_In(CycleNo);
7: #endif
8: ...
9: #if defined(WITH_XCP)
10:    CM_XCP_In();
11:    MyModel.Wh1Spd[0] = CM_XCP_A2lVar_In(MapWh1SpdFL);
12:    MyModel.Wh1Spd[1] = CM_XCP_A2lVar_In(MapWh1SpdFR);
13: #endif
14: ...
15: }
```

The return value of `CM_XCP_A2lVar_In()` is the value of the mapped variable, converted to double precision floating point.

14.4 Configuration Parameters

14.4.1 CCPParameters

ECU Configuration

nECU = NumberOfECU

Defines the number of used / available ECUs with CCP. All following ECU configuration parameters must have the prefix “ECU.<n>”, with n being the index of the ECU in the range of $0 \leq n < nECU$.

ECU.<n>.Name = Name

Name of the ECU. This string will be displayed in the *CCP Configuration* dialog of the Car-Maker GUI.

ECU.<n>.Addr = StationAddress

Address identifier of the ECU. This address (possible values: $0 \leq StationAddress < 65536$) will be used to connect to the ECU.

ECU.<n>.ASAP2_File = Path

Gives the file name / path to the ASAP2 file with the description of available measuring quantities. If no, or a relative path is specified, the directories *Data/Config* and *Data/Misc* will be searched in that order.

ECU.<n>.CAN_Slot =	Slot
ECU.<n>.CAN_Ch =	Channel

Used to identify the CAN bus, on which the ECU is connected. When using the M51 module, it will be most probably Slot 2 and Channel 0.

ECU.<n>.CAN_ID_CRO =	CAN_Identifier
ECU.<n>.CAN_ID_DTO =	CAN_Identifier

CAN message identifier which should be used to send CRO commands to the ECU (*CAN_ID_CRO*) and which is used by the ECU to respond to commands and for DTO packets during measurement (*CAN_ID_DTO*).

ECU.<n>.OnlyByteODT = bool

Tells, whether the ECU supports only ODT entries with the size of one byte, or not. If set to true (!= 0), then multibyte parameters will consume several ODT entries (one for each data byte).

ECU.<n>.SampleRateMin = Rate_ms

Defines the smallest possible time between two samples for measurements in milliseconds.

ECU.<n>.EventChannel = Ch_Id

Selects the event channel number to be used for data measurements. Currently, there is only one event channel supported for all DAQ lists.
Optional. Default 0.

ECU.<n>.DAQ_Timeout = enabled | disabled | MaxMissingPackets

Specifies how to deal with missing / lost DAQ packets during data measurements:

- **enabled**: if no DAQ packets are received within the duration of 3 times the cycle time of the DAQ list with the lowest sample rate, the measurement is considered as being terminated by the ECU.
- **disabled**: there is no special action, if any DAQ packets are lost, or even no DAQ packets are received.

- $1 < \text{MaxMissingPackets} < n$: enables the check for DAQ timeouts. Similar to `enabled`, except that `MaxMissingPackets` allows to customize the timeout.

Optional. Default `enabled`.

14.4.2 XCPPParameters

ECU Configuration

nECU = NumberOfECU

Defines the number of used / available ECUs with XCP. All following ECU configuration parameters must have the prefix “`ECU.<n>`”, with n being the index of the ECU in the range of $0 \leq n < n\text{ECU}$.

ECU.<n>.Name = Name

Name of the ECU. This string will be displayed in the *XCP Configuration* dialog of the Car-Maker GUI.

ECU.<n>.ASAP2_File = Path

Gives the file name / path to the ASAP2 file with the description of available measuring quantities. If no path, or a relative path is specified, the directories `Data/Config` and `Data/Misc` will be searched in that order.

ECU.<n>.ErrorHandling.Repeat2xMax = MaxRep ECU.<n>.ErrorHandling.RepeatNxMax = MaxRep

The number of retransmit, if there is an error during a request.
Optional. Default 2 (`Repeat2xMax`), -1 (`RepeatNxMax`).

ECU.<n>.ErrorHandling.DAQ_Timeout = enabled | disabled | MaxMissingPackets

Specifies how to deal with missing / lost DAQ packets during data measurements:

- `enabled`: if no DAQ packets are received within the duration of 3 times the cycle time of the DAQ list with the lowest sample rate, the measurement is considered as being terminated by the ECU.
- `disabled`: there is no special action, if any DAQ packets are lost, or even no DAQ packets are received.
- $1 < \text{MaxMissingPackets} < n$: enables the check for DAQ timeouts. Similar to `enabled`, except that `MaxMissingPackets` allows to customize the timeout.

Optional. Default `enabled`.

ECU.<n>.SaveMode = off | on | zip [transpose]

Allows to enable saving of measurement data to MDF files. If set to `on` or `zip`, all measured data will be stored into a MDF file according to the MDF specification version (s. <http://www.asam.net>). If set to `zip`, the measured data will be compressed with the deflate algorithm (ZIP format), otherwise the data is stored without compression.

If set to `zip transpose`, the measured data will be transposed before compression, which is equivalent to a column wise compression of the measurement data.

The structure of the produced MDF files is described in [section 'Storage of Measurement data' on page 520](#).

Optional. Default `off`.

ECU.<n>.SaveCtrl = auto | manual

When `ECU.<n>.SaveMode` is not `off`, this switch controls how measurement data is written to disk.

With the default setting `auto`, all the measured data is written to disk; starting with the first sample after start of DAQ measurement, up to the last sample at the end of the measurement. However, if the data acquisition takes long, the resulting MDF file will be very large. This may be a big disadvantage, if you are only interested in a short period of time (e.g. during a special driving maneuver).

An alternative is the setting `manual`. With this setting, the MDF result file will be created and prepared on start of DAQ measurement, but no samples will be written to disk, until triggered by the ScriptControl command [XCP::SaveDAQ_Start](#).

Optional. Default `auto`.

ECU.<n>.SaveFile = Path

Path and file name to store the measurement data. If `Path` is relative, then the output file will be saved to `SimOutput/<hostname>/<Path>`. If no file name is specified, the output file will be saved to the folder `SimOutput/<hostname>` and will be named `<A2L_Project>_<A2L_Module>_%Y%m%d_%H%M%S`, with `A2L_Project` and `A2L_Module` being the name and the module of the ASAP2 project, as specified in the ASAP2 file.

Optional. Only valid, if `ECU.<n>.SaveMode` is not `off`.

ECU.<n>.DAQ_Mode = single | block

Selects the DAQ programming algorithm.

If set to `single`, the DAQ lists will be programmed according to the selected ASAP2 variables, with each selected measurement quantity being represented by one ODT entry.

If set to `block`, then the DAQ lists will be programmed to measure a complete memory block, starting with the minimum address specified by the parameter `ECU.<n>.DAQ_MinAddr`, up to the maximum address `ECU.<n>.DAQ_MaxAddr`. However, in block mode, only the selected measurement quantities can be watched online in IPGControl. If `ECU.<n>.SaveMode` is not `off`, the produced MDF file will contain sample group and channel information for all imported ASAP2 variables within the measured address range.

Optional. Default `single`.

ECU.<n>.DAQ_MinAddr = Address

Specifies the minimal address for the DAQ programming for measuring in block mode.
Optional. Default 0. Only valid, if ECU.<n>.DAQ_Mode is block.

ECU.<n>.DAQ_MaxAddr = Address

Specifies the maximum address for the DAQ programming for measuring in block mode.
Optional. Default ECU.<n>.DAQ_MinAddr + 0x1000. Only valid, if ECU.<n>.DAQ_Mode is block.

ECU.<n>.OverrideASAP2 = bool

Defines, whether concurrent settings in the *XCPParameters* file override ASAP2 parameters, or not.

Optional. Default 0.

ECU.<n>.Protocol.Timings = T1 T2 T3 T4 T5 T6 T7

Allows to set the protocol timing parameters T1 to T7, as defined with the XCP specification (s. <http://www.asam.net>).

ECU.<n>.Protocol.Params = MAX_CTO MAX.DTO ByteOrder AddrGran

Sets the XCP protocol parameters MAX_CTO, MAX.DTO, the byte order of the XCP device and its limitations regarding the address granularity for memory access:

- MAX_CTO: maximum length of CTO requests.
- MAX.DTO: maximum length of DTO packets.
- ByteOrder: byte order, which is used by the XCP ECU when transferring multi-byte parameters:
 - MSB_FIRST: most significant byte is transferred first
 - MSB_LAST: most significant byte is transferred last
- AddrGran: address granularity for memory access:
 - ADDRESS_GRANULARITY_BYTE: access to memory addresses not aligned
 - ADDRESS_GRANULARITY_WORD: address parameters need to be aligned to even addresses (word aligned)
 - ADDRESS_GRANULARITY_DWORD: address parameters need to be aligned to 4-bytes (double word aligned)

**ECU.<n>.Protocol.CommMode = BLOCK [SLAVE] [MASTER MAX_BS MIN_ST]
ECU.<n>.Protocol.CommMode = INTERLEAVED QueueSize**

Selects the supported communication mode:

- BLOCK: block transfer mode ist supported by:

- SLAVE: by slave (ECU)
- MASTER: by master, with maximum block size MAX_BS and minimum separation time between two consecutive packets of MIN_ST
- INTERLEAVED: interleaved communication model is supported by the ECU (s. XCP specification Part 1 for meaning of parameter QueueSize)

Currently, the interleaved communication model is not used.

ECU.<n>.Protocol.OptCmd = OptCommand

ECU.<n>.Protocol.OptCmd:

- OptCommand_1**
- OptCommand_2**

Used to specify a list of optional commands, that are supported by the ECU. Command identifiers can be specified either as numeric value (0 .. 255), or as string according to the command identifiers listed in the XCP specification Part 2 (Protocol Layer Specification) (s. <http://www.asam.net>).

ECU.<n>.DAQ.Config = DAQ_Type MaxDAQ MaxEventCh MinDAQ

Defines configuration parameters for the DAQ processor of the XCP ECU:

- DAQ_Type: use DYNAMIC for dynamic DAQ configuration, or STATIC for static DAQ configuration.
- MaxDAQ: total number of available DAQ lists (s. XCP parameter MAX_DAQ).
- MaxEventCh: total number of available event channels (s. XCP parameter MAX_EVENT_CHANNEL).
- MinDAQ: total number of predefined DAQ lists (s. XCP parameter MIN_DAQ).

ECU.<n>.DAQ.Opt = OptType AddrExt IdentFieldType

Sets DAQ optimization options:

- OptType: optimisation type. Possible values are:
 - OPTIMISATION_TYPE_DEFAULT
 - OPTIMISATION_TYPE_ODT_TYPE_16
 - OPTIMISATION_TYPE_ODT_TYPE_32
 - OPTIMISATION_TYPE_ODT_TYPE_64
 - OPTIMISATION_TYPE_ODT_TYPE_ALIGNMENT
 - OPTIMISATION_TYPE_MAX_ENTRY_SIZE
- AddrExt: restrictions regarding the address extension of all entries within one ODT or within one DAQ. Possible values:
 - ADDRESS_EXTENSION_FREE
 - ADDRESS_EXTENSION_ODT
 - ADDRESS_EXTENSION_DAQ
- IdentFieldType: type of identification field the ECU will use when transferring DAQ packets. Possible values are:
 - IDENTIFICATION_FIELD_TYPE_ABSOLUTE

- IDENTIFICATION_FIELD_TYPE_RELATIVE_BYTE
- IDENTIFICATION_FIELD_TYPE_RELATIVE_WORD
- IDENTIFICATION_FIELD_TYPE_RELATIVE_WORD_ALIGNED

Please refer to XCP specification Part 1 for further information.

ECU.<n>.DAQ.Settings = OvldInd PrescSupp ResSupp PID_OffSupp

Sets the DAQ parameter OvldInd, and specifies if prescaler, resume and PID_off are supported. Possible values for OvldInd are:

- NO_OVERLOAD_INDICATION
- OVERLOAD_INDICATION_PID
- OVERLOAD_INDICATION_EVENT

Use the following keywords to indicate, that the ECU supports the prescaler, resume and PID_off:

- PrescSupp: PRESCALER_SUPPORTED
- ResSupp: RESUME_SUPPORTED
- PID_OffSupp: PID_OFF_SUPPORTED

ECU.<n>.DAQ.OptODT = EntrySizeGran MaxEntrySize

Sets ODT optimization parameters. Use one of the following values for EntrySizeGran:

- GRANULARITY_ODT_ENTRY_SIZE_DAQ_BYTE
- GRANULARITY_ODT_ENTRY_SIZE_DAQ_WORD
- GRANULARITY_ODT_ENTRY_SIZE_DAQ_DWORD
- GRANULARITY_ODT_ENTRY_SIZE_DAQ_DLONG

The parameter MaxEntrySize specifies the maximum size for ODT entries in bytes.

ECU.<n>.DAQ.Timestamp = Ticks Size Unit Fixed

If specified, this parameter indicates, that the ECU supports time stamps with DAQ packets:

- Ticks: s. description for XCP parameter TIMESTAMP_TICKS.
- Size: size of the timestamp field. Possible values are:
 - NO_TIME_STAMP
 - SIZE_BYTE
 - SIZE_WORD
 - SIZE_DWORD
- Unit: timestamp resolution. Supported values:
 - UNIT_1NS: 1 nano second per tick.
 - UNIT_10NS: 10 nano second per tick.
 - UNIT_100NS: 100 nano second per tick.
 - UNIT_1US: 1 micro second per tick.
 - UNIT_10US: 10 micro second per tick.

- UNIT_100US: 100 micro second per tick.
- UNIT_1MS: 1 milli second per tick.
- UNIT_10MS: 10 milli second per tick.
- UNIT_100MS: 100 milli second per tick.
- UNIT_1S: 1 second per tick.
- Fixed: If `TIMESTAMP_FIXED` is specified, the time stamp feature cannot be turned off.

ECU.<n>.DAQ.Events = Name EvChNo Dir MaxDAQ Cycle Unit Prio**ECU.<n>.DAQ.Events:**

Name	EvChNo	Dir	MaxDAQ	Cycle	Unit	Prio
------	--------	-----	--------	-------	------	------

Defines one or more DAQ event, that are available on the ECU for DAQ lists:

- **Name:** name of event channel.
- **EvChNo:** event channel number (s. XCP parameter `EVENT_CHANNEL_NUMBER`)
- **Dir:** direction of event. Supported values are:
 - DAQ
 - STIM
 - DAQ_STIM
- **MaxDAQ:** maximum number of DAQ lists in this event channel
- **Cycle:** event channel cycle time (s. XCP parameter `TIME_CYCLE`)
- **Unit:** event channel time unit (s. XCP parameter `TIME_UNIT`)
- **Prio:** priority of the event channel (not supported=0, lowest=1.. highest=255)

ECU.<n>.TransportLayer = TranspProto

This parameter defines the transport layer, which should be used for the communication to the ECU. Possible values are:

- **CAN:** for XCP on CAN
- **Ethernet:** for XCP on TCP/IP or UDP/IP
- **FlexRay:** for XCP on FlexRay

Transport Protocol Parameters for XCP on CAN

ECU.<n>.TransportLayer.CAN.CAN_Slot =	Slot
ECU.<n>.TransportLayer.CAN.CAN_Ch =	Channel

Used to identify the CAN bus, on which the ECU is connected. When using the M51 module, it will be most probably Slot 2 and Channel 0.

Transport Protocol Parameters for XCP on Ethernet

ECU.<n>.TransportLayer.Ethernet.Preferred = Protocol

This optional parameter allows to chose whether UDP or TCP should be used as preferred protocol, if the connected ECU allows both connection types. Possible values are:

- `udp`: prefer XCP on UDP/IP (default)
- `tcp`: prefer XCP on TCP/IP

Transport Protocol Parameters for XCP on FlexRay

ECU.<n>.TransportLayer.FlexRay.FR_CC = FlexCard_Id CC_Id

Used to identify the FlexRay Bus on which the ECU is connected. `FlexCard_Id` gives the number of the FlexCard, with 0 being the first FlexCard in the system, 1 for the second, and so forth. The parameter `CC_Id` is used as an index among the available Communication Controllers on the FlexCard, starting with 0 for the first.

ECU.<n>.TransportLayer.FlexRay.StripCmdMsg = bool

When transmitting an XCP command (CTO message), this parameter controls if the length of the transmitted frame should be automatically reduced to length of the message. In case the used slot which is used for CTO messages is located in the dynamic segment, this can help to save bandwidth.

14.4.3 Limiting the number of generated Data Dictionary variables

According to our experience, ASAP2 files for XCP tend to define a very large number of measurement quantities (even more than 10000), whereas there are only a very small amount of special interest (typically 10 to 50). However, importing thousands of unused variables will surely reduce operability.

In order to limit the amount of generated Data Dictionary variables, a list of variable names and patterns can be defined. These patterns will be used to add only those quantities to the Data Dictionary, which match the patterns.

ECU.<n>.ASAP2_Keys = A2I_Var_Name

ECU.<n>.ASAP2_Keys:
 A2I_Var_Name
 A2I_Var_*

Used to limit the number of generated Data Dictionary entries. Only those ASAP2 variables will be added to the Data Dictionary (available for measurements), which match one of the patterns. A pattern may either be the exact name of a measurement variable, or include a `*` character as a wildcard.

ECU.<n>.ASAP2_Filter = ddict | import

Defines, how the ECU.<n>.ASAP2_Keys should be applied.

If set to `import` (default), then the ASAP2 keys will be applied as import filter for the ASAP2 file. This will not only reduce the number of generated Data Dictionary entries, but will already limit the number of measurement quantities, that are displayed in the XCP Configuration Dialog of the CarMaker GUI.

If set to `ddict`, then all ASAP2 measurement quantities will be imported from the ASAP2 file, but only those are added to the Data Dictionary, which match the specified ASAP2 keys. The XCP Configuration Dialog then displays all ASAP2 variables.

Optional. Default `import`.

14.4.4 CCP / XCP (De-)Activation via Onboard Diagnostics

Some ECUs do not allow access to CCP / XCP functions by default, but require a short handshake of diagnostic messages, before activating the CCP / XCP slave.

ECU.<n>.nInitFrame = NumberOfFrames

Defines the number of diagnostic commands, which are necessary to activate the CCP / XCP slave inside of the ECU. The contents of the data frames have to be defined also, and must have the prefix “ECU.<n>.nInitFrame<m>”, with *n* being the index of the ECU in the range of $0 \leq n < n_{ECU}$, and *m* being the index for the diagnostic frame (which must be in the range of $0 \leq m < n_{InitFrame}$).

**ECU.<n>.InitFrame.<m>.Request = Request
ECU.<n>.InitFrame.<m>.ExpReply = Reply**

Contents of the Diagnostics request, that needs to be sent to the ECU for activation of the CCP / XCP functionality (`Request`), and the expected response (`Response`). The data has to be specified as a list of data bytes in hexadecimal notation, separated by white spaces.

Example

```
ECU.0.InitFrame.0.Request = 10 01  
ECU.0.InitFrame.0.ExpReply = 50 01  
ECU.0.InitFrame.0.Request = 31 01 02 03  
ECU.0.InitFrame.0.ExpReply = 71 01 02 03
```

ECU.<n>.nResetFrame = NumberOfFrames

Defines the number of diagnostic commands, which are necessary to deactivate the CCP / XCP slave inside of the ECU. The contents of the data frames have to be defined also, and must have the prefix “ECU.<n>.nResetFrame<m>”, with *n* being the index of the ECU in the range of $0 \leq n < n_{ECU}$, and *m* being the index for the diagnostic frame (which must be in the range of $0 \leq m < n_{ResetFrame}$).

ECU.<n>.ResetFrame.<m>.Request =	Request
ECU.<n>.ResetFrame.<m>.ExpReply =	Reply

Contents of the Diagnostics request, that needs to be sent to the ECU for activation of the CCP functionality (**Request**), and the expected response (**Reply**). The data has to be specified as a list of data bytes in hexadecimal notation, separated by white spaces.

Example

```
ECU.0.ResetFrame.0.Request = 10 01  
ECU.0.ResetFrame.0.ExpReply = 50 01  
ECU.0.ResetFrame.1.Request = 31 01 02 03  
ECU.0.ResetFrame.1.ExpReply = 71 01 02 03
```

14.4.5 ASAP2 variable Mappings

ECU.<n>.Map.<MappingKey> = ASAP2_Name
--

Maps the ASAP2 variable **ASAP2_Name** to an info file key. The key **ECU.<n>.Map.<MappingKey>** must be exactly the same string, as previously registered with `CM_XCP_A2lVar_Register()`. The value **ASAP2_Name** follows the naming scheme for ASAP2 variables `<Prefix>.<Module>.<QuName>` with the following meanings:

- **Prefix:** CCP or XCP (optional).
- **Module:** Name of the CCP / XCP module, as defined in the ASAP2 file.
- **QuName:** Name of the ASAP2 measurement quantity, as defined in the ASAP2 file.

Example

```
ECU.0.Map.WheelSpeed_FL = XCP.ESP.Wh1Spd_FL  
ECU.0.Map.WheelSpeed_FR = XCP.ESP.Wh1Spd_FR
```

14.5 The CCP and XCP Configuration Dialog

As soon as the CCP or XCP function is integrated into the CarMaker GUI and activated via the file `.CarMaker.tcl`, a new LED button will appear at the right bottom of the main window, just below the Stop button. If you press this button, its color will toggle between dark grey with the label “CCP off” (or “XCP off”) and colored orange with the label “CCP on” (or “XCP on”): Every time, when the state of the button changes to orange (CCP / XCP on), the *CCP Configuration* (or *XCP Configuration*) dialog will appear:

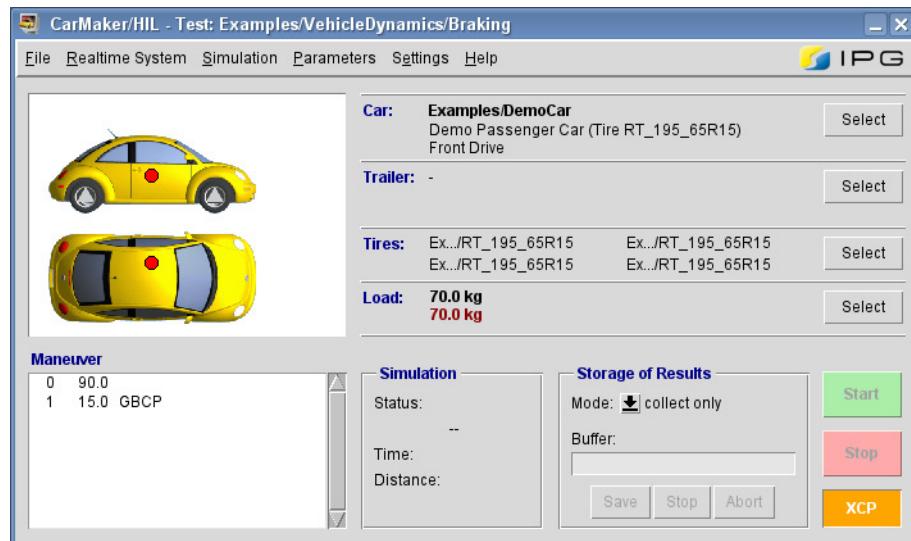


Figure 14.7: CarMaker GUI with XCP

The *CCP Configuration* dialog and the *XCP Configuration* dialog will be opened also when pressing the corresponding entry in the “Realtime System” menu.

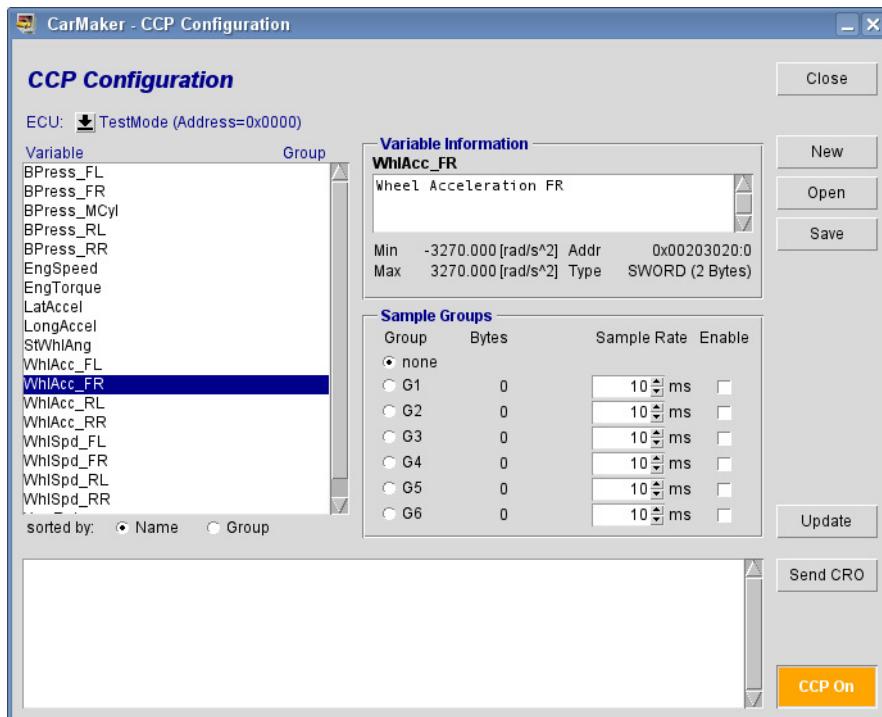


Figure 14.8: CCP Configuration dialog

The *XCP Configuration* dialog looks similar. However, there are some more features:

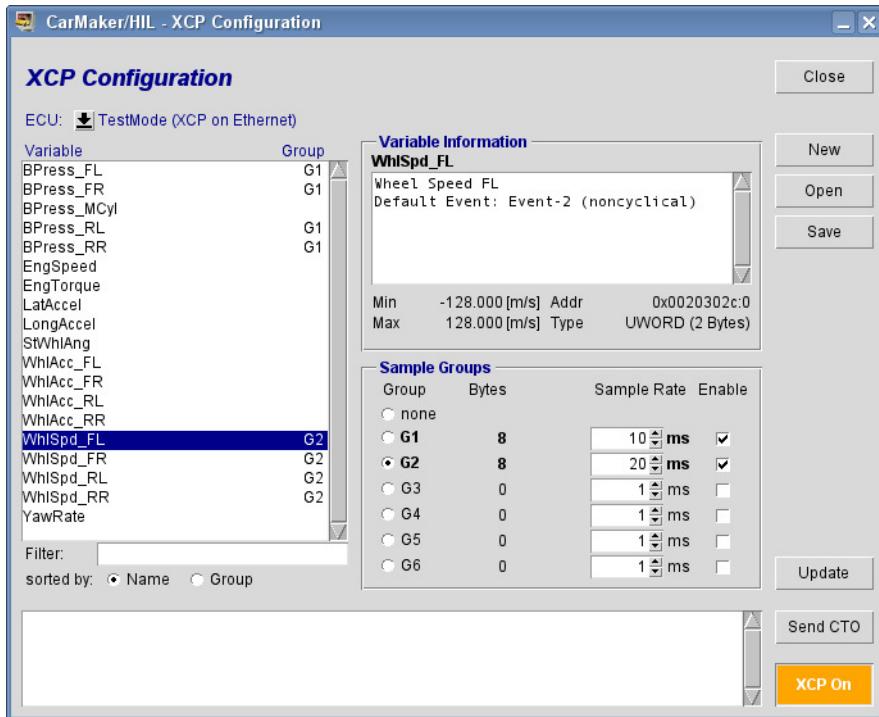


Figure 14.9: XCP Configuration dialog

The following description will again focus on XCP. Since the *CCP Configuration* dialog looks almost the same, we limit ourselves on one explanation.

14.5.1 Structure of the XCP Configuration Dialog

The XCP Configuration dialog is composed of seven sections:

ECU Selection

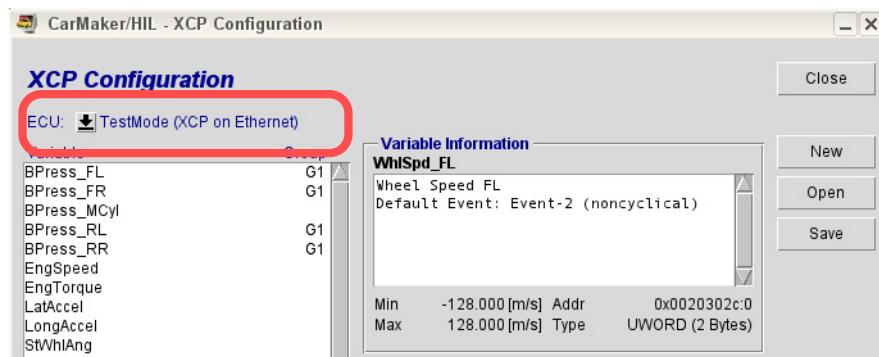


Figure 14.10: ECU Selection section

Using this menu, the user selects the ECU to which he wants to acquire data from. It is only possible to use the XCP functionality with one ECU at a time.

List of ECU variables

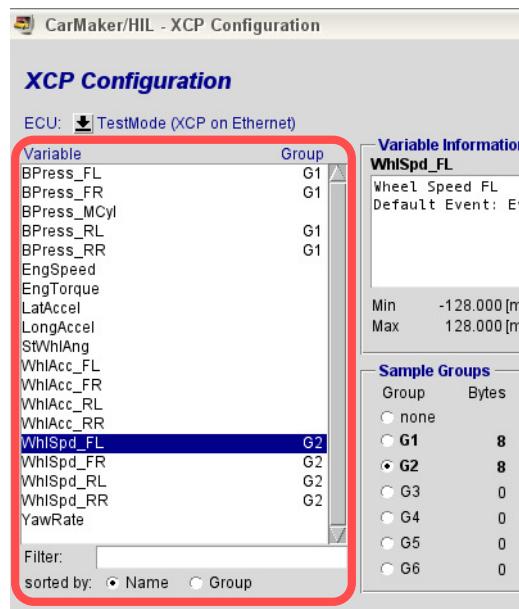


Figure 14.11: ECU Variables section

In this window, all variables which are available for data measurements, are listed. The amount of quantities also depends on the set of name patterns, if specified with the key `ECU.<n>.ASAP2_Keys` in the `XCPParameters` file. These variables are read by the XCP module when the CarMaker executable is launched. The names of the measurement variables correspond to those defined in the ASAP2 file. The user is offered the possibility to select variables and to assign them to a so called sample group for measurement.

- Selecting a variable in the list is done by a left-click with the mouse on the corresponding name: the variable is then highlighted in blue. If only one variable is selected, information about the variable is displayed in the *Variable Information* window.
- You can select multiple list entries at a time, using the mouse together with the Shift or Ctrl button – just like with common file managers.
- A variable can be assigned to a sample group by clicking the corresponding radio button in the *Sample Groups* window.
- The variable list can either be sorted by *Name* or by *Group*. Sorting by group means, that those variables are listed first, that are selected for a measurement.
- In the *Filter* entry, you can enter a search pattern for variable names. The variable list is automatically reduced to those, whose name matches the filter pattern.



The filter function is available for XCP, only. In the CCP Configuration dialog, this entry does not exist.

Log window

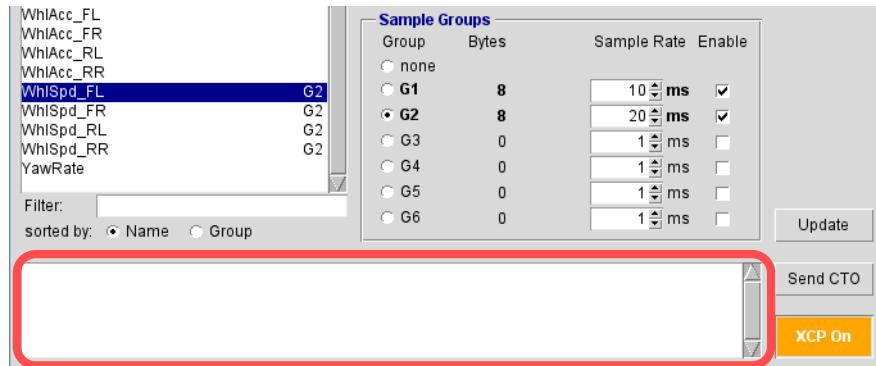


Figure 14.12: Log window

All XCP error messages sent by the ECU are displayed in this window.

Configuration Buttons

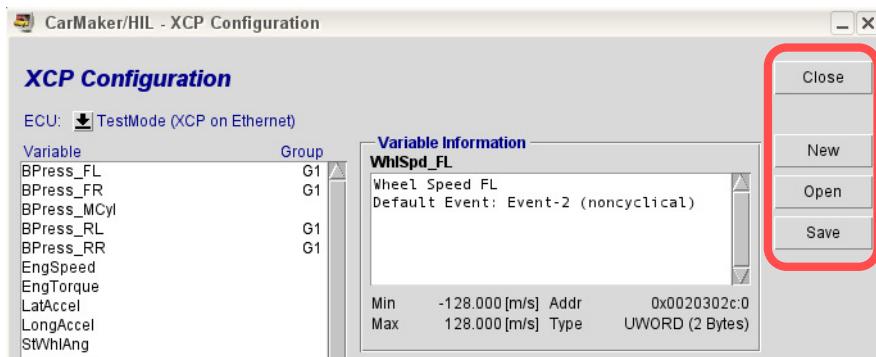


Figure 14.13: Configuration Buttons

In order to ease the operation with XCP measurements, a configuration can be saved and restored afterwards. The configuration comprises the list of selected variables, their assignment to sample groups and the configured sample rates. The following four configuration buttons are available:

- *Close*: Closes the XCP Configuration dialog.
- *New*: Resets the current settings to defaults and starts with a new and empty configuration
- *Open*: Loads an existing configuration.
- *Save*: Saves the current configuration.

When pressing the Save button, a file browser dialog appears, asking for a path and file name. The configuration data will then be written in the infofile format.

To restore the configuration to a previously saved one, press the Open button, browse to the directory with your saved configurations and choose a file. After loading the configuration, the name of the info file and its location are displayed in the title bar of the XCP Configuration dialog. The selected variables, the defined groups and the given sample times are updated with the settings from the loaded file. On the next variable update, the configuration will be automatically checked for collisions with the currently loaded ASAP2 file. If there are any unknown variables, a warning message is displayed in the Log window.

Variable Information

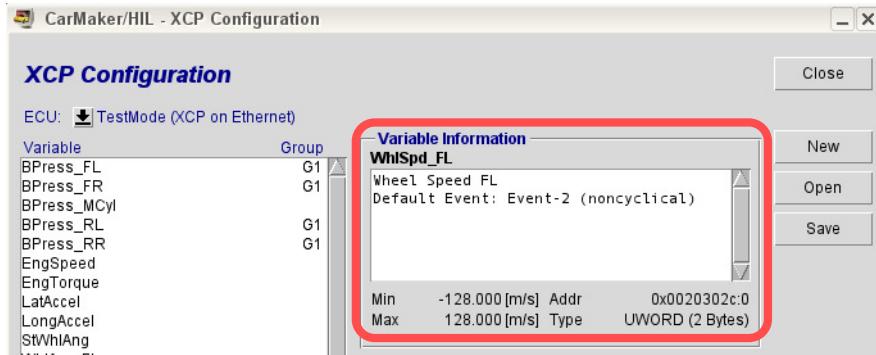


Figure 14.14: Variable Information section

This area displays some information about selected variables:

- The complete variable name (*Name* and *LongIdentifier*).
- A short description of the variable, as specified in the ASAP2 file.
- The minimal and maximal possible values for the variable (*Min* and *Max*).
- The address of the variable in the ECU memory (*Addr*) the address extension.
- The size of the variable in bytes and the data type (*Type*).

Sample Groups

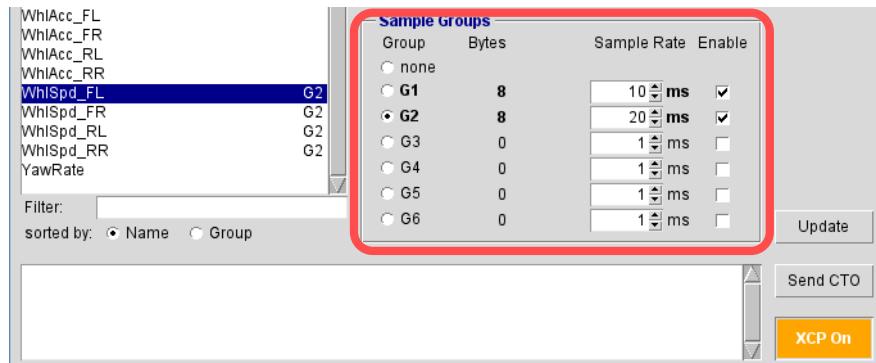


Figure 14.15: Sample Groups section

In this area, you configure the data measurement:

- If one or more quantities are selected in the variable list, they are assigned to a sample group by clicking one of the following radio buttons: *none* or *G1* to *G6*.
When selecting a group, the element size of all assigned variables will be totaled and displayed in the *Bytes* column.
- The update frequency for a sample group is tuned in the *Sample Rate* column.
- Finally, in order to enable the transmission, the check button in the *Enable* column has to be checked.

The operating concept of the *XCP Configuration* dialog tries to hide the internal configuration in DAQ lists, ODT lists and ODT entries from the user. Since the protocol details would only confuse the user, we concentrate on the basics, which is receiving the physical value of a selected measurement variable at a given sample rate.

When programming the ECU for data acquisition, for each measurement quantity, the best DAQ configuration will be determined. First, all the members of one group will be tried to be assigned to one and the same ODT and DAQ list. However, due to the limits in size and number of available ODT lists, ODT entries and DAQ lists and due to the requirements of a single quantity and its event configuration, a sample group cannot be meant to represent a single DAQ list.

Communication Control

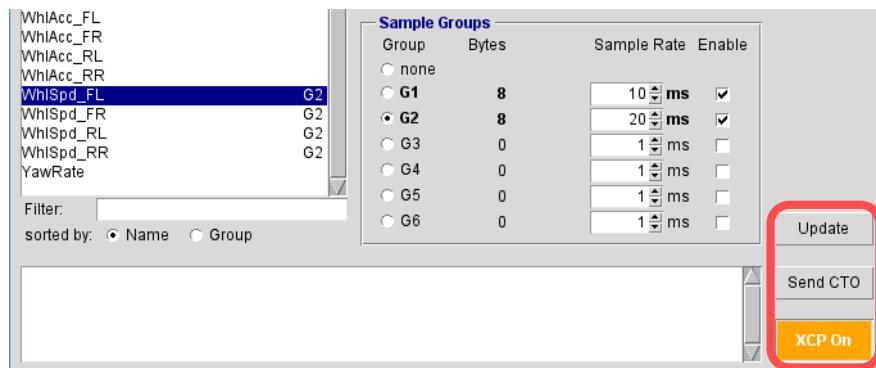


Figure 14.16: Communication Control section

With the LED button in the lower right corner of the dialog, the user can choose, if the data acquisition should be started the next time a simulation is started. Each time, when the LED button is pressed, it will change its color from dark grey with the text *XCP Off* to orange with the text *XCP On*:

- To enable automatic data acquisition with each CarMaker TestRun, the LED button has to be in its orange state.
- Automatic data acquisition is disabled, if the LED button is in its dark grey state.
- During a data acquisition, the LED button will be colored dark green.

The LED button is a second instance of the LED button in the lower right corner of the main GUI, and has the same function.

The *Update* button issues an update request to the realtime system. The *XCP Configuration* dialog will then synchronize the configuration with the running CarMaker process. Normally, the update is done automatically, whenever there are changes detected.

With the *Send CTO* button, arbitrary XCP requests can be sent to the ECU. The result will be displayed in the *Log window*. When pressing the button, the following popup will appear:

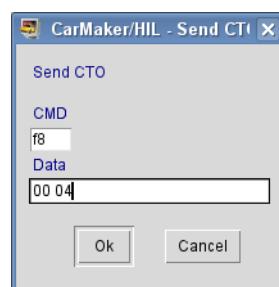


Figure 14.17: Send CTO popup

In the *CMD* entry, enter the code of the XCP command in hexadecimal notation. The *Data* field is optional, depending on the selected command. The data bytes need to be listed in hexadecimal notation, grouped in bytes. [Figure 14.17](#) above shows the *GET_SEED* request for DAQ access.

Context Menu

The communication with the ECU is handled automatically and does not need any additional controls. However, there is still a context menu available, giving access to some useful features. The context menu is displayed whenever you right-click into the XCP Configuration dialog:

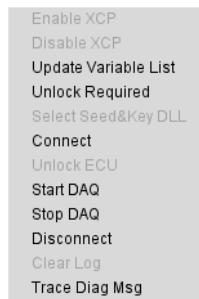


Figure 14.18: Context Menu

Some functions might be disabled. The availability depends on the individual configuration:

- *Enable XCP / Disable XCP*: Allows to enable and disable the XCP functionality in the slave ECU. This feature is only available, if there is a diagnostics handshake configured in the XCPParameters file (refer to [section 14.4.4 'CCP / XCP \(De-\)Activation via Onboard Diagnostics'](#)).
- *Update Variable List*: Requests the name of the ASAP2 file from the realtime system and refreshes the variable list.
- *Update DAQ Configuration*: Requests the DAQ configuration from the ECU. This comprises the number of available static and dynamic DAQ lists, ODT lists, ODT entries, limitations in size and many more DAQ specific parameters of the ECU.
- *Unlock Required / Select Seed&Key DLL / Unlock*: If the ECU needs to be unlocked first before granting access to different functions, you can select the *Unlock Required* button. After that, you can select a *Seed&Key DLL* and request to Unlock the ECU.

This feature is not available for CCP.

- *Start DAQ / Stop DAQ*: Starts and stops a data acquisition manually.
- *Connect / Disconnect*: Connect to or disconnect from ECU.
- *Clear Log*: Clears the contents of the Log window. This feature is only available, if the mouse pointer is located inside the Log window.
- *Trace Diag Msg*: Allows to watch the diagnostic messages, which are sent to and received from the ECU.



14.6 ScriptControl Interface

The *CCP Configuration* dialog and the *XCP Configuration* dialog offer an interactive control for the configuration of DAQ measurements. However, for automated tests and customization purpose, there is the same and even more functionality available as ScriptControl commands.

This section lists the most common ScriptControl commands for CCP and XCP. The description uses the following notation for the description of arguments:

- `[arg]`: Optional parameter, argument can be omitted
- `<arg>`: Mandatory argument

14.6.1 CCP Commands

User Configuration

CCP::NewCfg

Syntax

`CCP::NewCfg`

Description

Creates a new user configuration:

- clears the list of ASAP2 variables.
- resets the group definition and the sample times.
- reads the list of ASAP2 variables from the realtime system.

CCP::OpenCfg

Syntax

`CCP::OpenCfg <filename>`

Description

Loads a previously saved user configuration from file *filename*.

Return value

- On success: 0
- On error: <0 (error message is displayed in *Log window*)

CCP::SaveCfg

Syntax

`CCP::SaveCfg [<filename>]`

Description

Saves the current user configuration into the file *filename*. If an empty string, or no file name is specified, a file browser will be opened and wait for the user to select either an existing file, or input the name for a new configuration file.

Management of ASAP2 Variable List

CCP::ClearVariables

Syntax

CCP::ClearVariables

Description

Clears the list of ASAP2 variables. The list of ASAP2 variable has to be requested again from the realtime system.

CCP::UpdateVariables

Syntax

CCP::UpdateVariables

Description

Compares the list of loaded variables with the variables available on the realtime system. If some of the variables are not available in the current ASAP2 data base, a warning message with the name of these variables is displayed in the ScriptControl window.

Return value

- On error, Tcl list with names of unavailable variables.
- On success, empty list ("").

CCP::RequConfig

Syntax

CCP::RequConfig

Description

Requests the ECU configuration from the realtime system. The execution of the script will not be interrupted, i.e. this function does not wait for the result.

CCP::RequASAP2VarInfo

Syntax

CCP::RequASAP2VarInfo

Description

Requests the list of ASAP2 variables from the realtime system. The execution of the script will not be interrupted, i.e. this function does not wait for the result.

CCP::GetVariables

Syntax

```
CCP::GetVariables
```

Description

Get the list of ASAP2 variables loaded on the RT Unit. The function returns a Tcl list with names of all available variables.

CCP::GetActiveVariables

Syntax

```
CCP::GetActiveVariables
```

Description

Get the list of active variables, i.e. selected for measurement. The function returns all variables assigned to an active group.

Return value

- Tcl list with names of selected variables in all active sample groups.
- If no variables selected, no sample group activated: empty Tcl list ("").

CCP::GetGroupVariables

Syntax

```
CCP::GetGroupVariables <group>
```

Description

Get the list of variables, assigned to the given sample group (`group` = group number = 1..6).

Return value

- Tcl list of variable names, assigned to given group.
- If no variables assigned to the group: empty Tcl list ("").

CCP::GetSampleRate

Syntax

```
CCP::GetSampleRate <group>
```

Description

Get the sample time of the given sample group (`group` = group number = 1..6).

Return value

- ≥ 0 : sample time in milliseconds.
- < 0 : invalid parameter (unknown group).

CCP::GetVariable

Syntax

```
CCP::GetVariable <varname>
```

Description

Get detailed information about variable (`varname` = variable name).

Return value

- Tcl list with variable information. Each list element defines a property of the variable:
 {Name Address AddrExt Size DataType Unit Min Max Group}
 - Name: Name of ASAP2 variable.
 - Address: Address in ECU memory.
 - Size: Size of data element in bytes.
 - DataType: Data type of variable. Possible values are: UBYTE, SBYTE, UWORD, SWORD, ULONG, SLONG, FLOAT32_IEEE, FLOAT64_IEEE, BIT.
 - Unit: Unit of physical value.
 - Min: Minimum possible value.
 - Max: Maximum possible value.
 - Group: Number os assigned sample group (or 0, if not assigend to any group)
- If the specified variable does not exist, an empty Tcl list ("") is returned.

CCP::StartStopDAQ_AutoSet

Syntax

```
CCP::StartStopDAQ_AutoSet <on_off>
```

Description

Activate / Deactivate the automatic start of data acquisition with each TestRun. The parameter `on_off` can have one of the following values:

- 0: Deactivate automatic data acquisition.
- 1: Activate automatic data acquisition.

CCP::ResetStateOnRT

Syntax

```
CCP::ResetStateOnRT
```

Description

Resets the CCP communication on the realtime system. Following operations are performed:

- Any active data acquisition is stopped.
- If connected, the CCP session is closed.
- Switches the CCP master on the realtime system into its initial state.

The execution of the script will not be interrupted, i.e. this function does not wait for the result.

Log Messages

CCP::Log

Syntax

```
CCP::Log <class> <msg>
```

Description

Displays a log message in the *Log window* and ScriptControl window. The parameter class gives the type of the message msg. Accepted types are:

- Msg: Standard text.
- Warning: Displayed as bold yellow text in *Log window*. In the ScriptControl window, the same text will be displayed with the standard font format, with the prefix “Warning”.
- Error: Displayed as bold red text in *Log window*. In the ScriptControl window, the same text will be displayed with the standard font format, with the prefix “Error”.

CCP::ClearLog

Syntax

```
CCP::ClearLog
```

Description

Clears the content of the *Log window*.

CCP Activation / Deactivation

CCP::EnableCCP

Syntax

```
CCP::EnableCCP
```

Description

Activates the CCP slave on the selected ECU by using diagnostic commands (KWP 2000, UDS, etc.).

CCP::DisableCCP

Syntax

```
CCP::DisableCCP
```

Description

Deactivates the CCP slave on the selected ECU by using diagnostic commands (KWP 2000, UDS, etc.).

CCP Protocol Functions

CCP::CONNECT

Syntax

```
CCP::CONNECT
```

Description

Open a CCP connection to the selected ECU, using the *CONNECT* command. The execution of the script will not be interrupted, i.e. this function does not wait for the result.

CCP::DISCONNECT

Syntax

```
CCP::DISCONNECT
```

Description

Terminates an active CCP session to the selected ECU. The execution of the script will not be interrupted, i.e. this function does not wait for the result.

CCP::TxCRO

Syntax

```
CCP::TxCRO <cmd> [data]
```

Description

Sends an arbitrary CRO request to the ECU. The parameter `cmd` is the code of the CCP command in hexadecimal notation. Data bytes are optional, depend on the selected command and need to be specified as a Tcl list of bytes in hexadecimal notation.

The execution of the script will not be interrupted, i.e. this function does not wait for the result.

CCP::SendHex <cmd> (data) (timeout)

Syntax

```
CCP::SendHex <cmd> [data] [timeout]
```

Description

Sends an arbitrary CRO request to the ECU and waits for the result. The parameter `cmd` is the code of the CCP command in hexadecimal notation. Data bytes are optional, depend on the selected command and need to be specified as a Tcl list of bytes in hexadecimal notation. An additional `timeout` can be specified in milliseconds, to handle the case that the ECU does not respond (default timeout is 5000 = 5 seconds).

Return value

- If the command was transmitted successful, and the ECU sent a response: A Tcl list consisting of the sent command code `cmd`, the status of the result (`ff` for positive response, `fe` for negative response) and the received data bytes.
- On error (time out): Empty Tcl list (""). An error message is displayed in the *Log window*.

CCP::CRO_Err2Msg

Syntax

```
CCP::CRO_Err2Msg <ccperrno>
```

Description

Converts the given CCP error code `ccperrno` to a human readable text, as defined in the CCP Specification.

DAQ Management

CCP::StartDAQ

Syntax

```
CCP::StartDAQ
```

Description

Starts Data Acquisition.

Return value

- 0: Data Acquisition successfully started.
- < 0: On error, error message displayed in *Log window*.

CCP::StopDAQ

Syntax

```
CCP::StopDAQ
```

Description

Stops active Data Acquisition.

Return value

- 0: Data Acquisition successfully stopped.
- < 0: On error, error message displayed in *Log window*.

14.6.2 XCP Commands

User Configuration

XCP::NewCfg

Syntax

```
XCP::NewCfg
```

Description

Creates a new user configuration:

- clears the list of ASAP2 variables.
- resets the group definition and the sample times.
- reads the list of ASAP2 variables from the realtime system.

XCP::OpenCfg

Syntax

```
XCP::OpenCfg <filename>
```

Description

Loads a previously saved user configuration from file *filename*.

Return value

- On success: 0
- On error: <0 (error message is displayed in *Log window*)

XCP::SaveCfg

Syntax

```
XCP::SaveCfg [<filename>]
```

Description

Saves the current user configuration into the file *filename*. If an empty string, or no file name is specified, a file browser will be opened and wait for the user to select either an existing file, or input the name for a new configuration file.

Management of ASAP2 Variable List

XCP::ClearVariables

Syntax

```
XCP::ClearVariables
```

Description

Clears the list of ASAP2 variables. The list of ASAP2 variable has to be requested again from the realtime system.

XCP::UpdateVariables

Syntax

```
XCP::UpdateVariables
```

Description

Compares the list of loaded variables with the variables available on the realtime system. If some of the variables are not available in the current ASAP2 data base, a warning message with the name of these variables is displayed in the ScriptControl window.

XCP::RequConfig

Syntax

```
XCP::RequConfig
```

Description

Requests the ECU configuration from the realtime system. The execution of the script will not be interrupted, i.e. this function does not wait for the result.

XCP::RequASAP2VarInfo

Syntax

```
XCP::RequASAP2VarInfo
```

Description

Requests the list of ASAP2 variables from the realtime system. The execution of the script will not be interrupted, i.e. this function does not wait for the result.

XCP::GetVariables

Syntax

```
XCP::GetVariables
```

Description

Get the list of ASAP2 variables loaded on the RT Unit. The function returns a Tcl list with names of all available variables.

XCP::GetActiveVariables

Syntax

```
XCP::GetActiveVariables
```

Description

Get the list of active variables, i.e. selected for measurement. The function returns all variables assigned to an active group.

Return value

- Tcl list with names of selected variables in all active sample groups.
- If no variables selected, no sample group activated: empty Tcl list ("").

XCP::GetGroupVariables

Syntax

```
XCP::GetGroupVariables <group>
```

Description

Get the list of variables, assigned to the given sample group (`group` = group number = 1..6).

Return value

- Tcl list of variable names, assigned to given group.
- If no variables assigned to the group: empty Tcl list ("").

XCP::GetSampleRate

Syntax

```
XCP::GetSampleRate <group>
```

Description

Get the sample time of the given sample group (`group` = group number = 1..6).

Return value

- ≥ 0 : sample time in milliseconds.
- < 0 : invalid parameter (unknown group).

XCP::GetVariable

Syntax

```
XCP::GetVariable <varname>
```

Description

Get detailed information about variable (`varname` = variable name).

Return value

- Tcl list with variable information. Each list element defines a property of the variable:
{Name Address AddrExt Size DataType Unit Min Max Group}
 - Name: Name of ASAP2 variable.
 - Address: Address in ECU memory.
 - Size: Size of data element in bytes.
 - DataType: Data type of variable. Possible values are: UBYTE, SBYTE, UWORD, SWORD, ULONG, SLONG, FLOAT32_IEEE, FLOAT64_IEEE, BIT.
 - Unit: Unit of physical value.
 - Min: Minimum possible value.
 - Max: Maximum possible value.
 - Group: Number os assigned sample group (or 0, if not assigend to any group)

- If the specified variable does not exist, an empty Tcl list ("") is returned.

XCP::StartStopDAQ_AutoSet

Syntax

```
XCP::StartStopDAQ_AutoSet <on_off>
```

Description

Activate / Deactivate the automatic start of data acquisition with each TestRun. The parameter `on_off` can have one of the following values:

- 0: Deactivate automatic data acquisition.
- 1: Activate automatic data acquisition.

XCP::SaveDAQ_ControlModeSet

Syntax

```
XCP::SaveDAQ_ControlModeSet [ctrl]
```

Description

Changes the method, how DAQ measurement data is written to disk. The parameter is optional and `ctrl` can have one of the following values:

- `auto`: All DAQ samples are written to disk immediately, starting with the start of DAQ measurement.
- `manual`: DAQ samples will be written to the prepared MDF file only, if the DAQ measurement is already started, and if recording has been triggered by the command [XCP::SaveDAQ_Start](#). DAQ samples are written to disk, until either the DAQ measurement is stopped, or if recording is paused by the command [XCP::SaveDAQ_Pause](#).

The return value of the command is the new control mode. If called without arguments, the command returns the current control mode.

XCP::SaveDAQ_Start

Syntax

```
XCP::SaveDAQ_Start
```

Description

Triggers (enables) the recording of DAQ samples into the prepared MDF file. If called again, after being paused (s. [XCP::SaveDAQ_Pause](#)), recording will continue by appending new DAQ samples to the same MDF file.

XCP::SaveDAQ_Pause

Syntax

```
XCP::SaveDAQ_Pause
```

Description

Pauses the recording of DAQ samples into the prepared MDF file. The Running DAQ measurement will not be affected, the MDF file is still kept opened. Recording can be continued again with the command [XCP::SaveDAQ_Start](#), as long as the DAQ measurement is still running.

XCP::SaveDAQ_Stop

Syntax

```
XCP::SaveDAQ_Stop
```

Description

Stops the recording of DAQ samples into the prepared MDF file. The Running DAQ measurement will not be affected, the MDF file is closed. After the MDF file has been closed, the XCP Storage module is reinitialized immediately and prepared for storage again.

While reinitializing, a new MDF file will be created and prepared for storage. The file name of the new MDF file is derived from the previous file name – e.g. as configured with key *ECU.<n>.SaveFile* ([section 14.4.2 'XCPParameters'](#)) – by appending _<FileNo> to the base name of the original file name. The counter *FileNo* starts with 2 (2nd MDF file) and is incremented with each reinitialization of the storage module.

If Recording is active, or started immediately with [XCP::SaveDAQ_Start](#) again, no data will be written to file, while the reinitialization process is still ongoing.

XCP::ResetStateOnRT

Syntax

```
XCP::ResetStateOnRT
```

Description

Resets the XCP communication on the realtime system. Following operations are performed:

- Any active data acquisition is stopped.
- If connected, the XCP session is closed.
- Switches the XCP master on the realtime system into its initial state.

The execution of the script will not be interrupted, i.e. this function does not wait for the result.

Log Messages

XCP::Log

Syntax

```
XCP::Log <class> <msg>
```

Description

Displays a log message in the *Log window* and ScriptControl window. The parameter class gives the type of the message msg. Accepted types are:

- Msg: Standard text.
- Warning: Displayed as bold yellow text in *Log window*. In the ScriptControl window, the same text will be displayed with the standard font format, with the prefix “Warning”.
- Error: Displayed as bold red text in *Log window*. In the ScriptControl window, the same text will be displayed with the standard font format, with the prefix “Error”.

XCP::ClearLog

Syntax

```
XCP::ClearLog
```

Description

Clears the content of the *Log window*.

XCP Activation / Deactivation

XCP::EnableXCP

Syntax

```
XCP::EnableXCP
```

Description

Activates the XCP slave on the selected ECU by using diagnostic commands (KWP 2000, UDS, etc.).

XCP::DisableXCP

Syntax

```
XCP::DisableXCP
```

Description

Deactivates the XCP slave on the selected ECU by using diagnostic commands (KWP 2000, UDS, etc.).

XCP Protocol Functions

XCP::CONNECT

Syntax

XCP::CONNECT

Description

Open a XCP connection to the selected ECU, using the *CONNECT* command. The execution of the script will not be interrupted, i.e. this function does not wait for the result.

XCP::DISCONNECT

Syntax

XCP::DISCONNECT

Description

Terminates an active XCP session to the selected ECU. The execution of the script will not be interrupted, i.e. this function does not wait for the result.

XCP::GET_STATUS

Syntax

XCP::GET_STATUS

Description

Requests the current session status from the XCP slave ECU. This function sends the *GET_STATUS* request (Command 0xfd) and waits for the result.

Return value

- On Success: Response from ECU (s. XCP::SendHex).
- On error: Empty Tcl list ("").

XCP::TxCTO

Syntax

XCP::TxCTO <cmd> [data]

Description

Sends an arbitrary CTO request to the ECU. The parameter `cmd` is the code of the XCP command in hexadecimal notation. Data bytes are optional, depend on the selected command and need to be specified as a Tcl list of bytes in hexadecimal notation.

The execution of the script will not be interrupted, i.e. this function does not wait for the result.

XCP::SendHex

Syntax

```
XCP::SendHex <cmd> [data] [timeout]
```

Description

Sends an arbitrary CTO request to the ECU and waits for the result. The parameter `cmd` is the code of the XCP command in hexadecimal notation. Data bytes are optional, depend on the selected command and need to be specified as a Tcl list of bytes in hexadecimal notation. An additional `timeout` can be specified in milliseconds, to handle the case that the ECU does not respond (default timeout is 5000 = 5 seconds).

Return value

- If the command was transmitted successful, and the ECU sent a response: A Tcl list consisting of the sent command code `cmd`, the status of the result (`ff` for positive response, `fe` for negative response) and the received data bytes.
- On error (time out): Empty Tcl list (""). An error message is displayed in the *Log window*.

XCP::XCP_Err2Msg

Syntax

```
XCP::XCP_Err2Msg <xcpperno>
```

Description

Converts the given XCP error code `xcpperno` to a human readable text, as defined in the XCP Specification Part 2 – Protocol Layer Specification.

DAQ Management

XCP::StartDAQ

Syntax

```
XCP::StartDAQ
```

Description

Starts Data Acquisition.

Return value

- 0: Data Acquisition successfully started.
- < 0: On error, error message displayed in *Log window*.

XCP::StopDAQ

Syntax

```
XCP::StopDAQ
```

Description

Stops an active Data Acquisition.

Return value

- 0: Data Acquisition successfully stopped.
- < 0: On error, error message displayed in *Log window*.

Unlock Procedure

XCP::SelSK_DLL

Syntax

```
XCP::SelSK_DLL
```

Description

Opens a file browser dialog to browse for and select a Seed&Key Shared Library (either Windows DLL or Linux Shared Object). The Shared Library must meet the requirements for external Seed&Key functions, as defined in XCP Specification Part 4 – Interface Specification.

XCP::UnlockECU

Syntax

```
XCP::UnlockECU [Resource]
```

Description

Unlocks the ECU for access to a given resource. This function handles the complete unlock procedure, i.e. requests a seed for the specified `Resource`, computes the key with the selected Seed&Key Shared Library and sends the key back to the ECU. The progress of the unlock procedure is displayed in the Log window and in the ScriptControl window.

If `Resource` is omitted, the default resource DAQ will be used.

For a list of available resource keywords, refer to XCP::SK_GetAvailPriv.

Return value

- On Success: 0.

- On Error: -1.

XCP::SK_GetAvailPriv

Syntax

```
XCP::SK_GetAvailPriv [FName]
```

Description

Determines the available privileges, provided by the given Seed&Key Shared Library. If `FName` is not specified, the selected Seed&Key Shared Library will be used, as defined with `XCP::SelSK_DLL`. This command calls the `XCP_GetAvailablePrivileges()` function.

The available privileges are returned as a Tcl list of keywords. Possible members are:

cal_pag	Access to Calibration / Paging commands
daq	Access to DAQ lists commands (direction DAQ)
stim	Access to DAQ lists commands (direction STIM)
pgm	Access to Programming commands

Return value

- On Success: Tcl list with available privileges.
- On Error: Empty Tcl list ("").

XCP::SK_CompKeyFromSeed

Syntax

```
XCP::SK_CompKeyFromSeed <Seed> [Resource] [FName]
```

Description

Computes the key out of a Seed, which was delivered by the ECU as a reply to the `GET_SEED` command. If `Resource` is not specified, the resource DAQ will be used. If `FName` is omitted, the selected Seed&Key Shared Library will be used, as defined with `XCP::SelSK_DLL`. This command calls the `XCP_ComputeKeyFromSeed()` function.

For a list of possible resource keywords, refer to `XCP::SK_GetAvailPriv`.

Return value

- On Success: Computed key as Tcl list of bytes in hexadecimal notation.
- On Error: Empty Tcl list ("").

XCP::Unlock_RequSeed

Syntax

```
XCP::Unlock_RequSeed [Resource]
```

Description

Requests a seed from the XCP slave ECU for an unlock of the specified resource by sending the *GET_SEED* command. If `Resource` is not specified, the default resource DAQ is used.

For a list of possible resource keywords, refer to `XCP::SK_GetAvailPriv`.

Return value

- On Success: The seed, sent by the ECU, as Tcl list of bytes in hexadecimal notation.
- On Error: The string “ERROR”.

XCP::Unlock_SendKey

Syntax

```
XCP::Unlock_SendKey <Key>
```

Description

Send the computed `Key` to the ECU for an unlock of a requested resource.

Return value

- On Success: 0.
- On Error: -1.

Calibration Commands

XCP::Cal::UpdateParameters

Syntax

```
XCP::Cal::UpdateParameters
```

Description

This function uses the ASAP2 file of the currently selected ECU and scans it for available parameters, that are available for online calibration.



The ASAP2 file will be scanned for *CHARACTERISTIC* definitions. Only those characteristics, that are identified as independent and available for calibration, are imported. All parameters, that are marked as virtual (*VIRTUAL_CHARACTERISTIC*), or dependent on other characteristics (*DEPENDENT_CHARACTERISTIC*), will be ignored.

XCP::Cal::ParList::GetParList

Syntax

```
XCP::Cal::ParList::GetParList
```

Description

Returns a Tcl list of parameter names, that are available for online calibration. The names can be used for calibration commands, like XCP::Cal::ReadParameter.

XCP::Cal::ReadParameter

Syntax

```
XCP::Cal::ReadParameter <name>
```

Description

Reads the parameter, specified with `name` and returns its physical value.

Return value

- On Success: Physical value of parameter.
- On Error: Generates Tcl Error.

XCP::Cal::WriteParameter

Syntax

```
XCP::Cal::WriteParameter <name> <value>
```

Description

Writes the parameter, specified with `name` and sets its physical `value`.

Return value

- On Success: 0.
- On Error: Generates Tcl Error.

XCP::Cal::ReadParameterHex

Syntax

```
XCP::Cal::ReadParameterHex <name>
```

Description

Reads the parameter, specified with `name` and returns its raw value in hexadecimal notation.

Return value

- On Success: Raw value of parameter.
- On Error: Generates Tcl Error.

XCP::Cal::WriteParameterHex

Syntax

```
XCP::Cal::WriteParameterHex <name> <value>
```

Description

Writes the parameter, specified with `name` and sets its raw `value` (specified in hexadecimal notation).

Return value

- On Success: 0.
- On Error: Generates Tcl Error.

14.7 References

The implementation of CCP and XCP is based on the following specifications.

CCP Specification

- H. Kleinknecht: "CAN Calibration Protocol" ASAM e.V., Version 2.1, 1999, ASAM MCD-1 CCP (<http://www.asam.net>)

XCP Specification

- ASAM (Eds.): "XCP Version 1.0 - Part 1 - Overview" ASAM e.V., 2003, ASAM MCD-1 XCP V1.0.0 (<http://www.asam.net>)
- ASAM (Eds.): "XCP Version 1.0 - Part 2 - Protocol Layer Specification" ASAM e.V., 2003, ASAM MCD-1 XCP V1.0.0 (<http://www.asam.net>)
- ASAM (Eds.): "XCP Version 1.0 - Part 3 - XCP on CAN - Transport Layer Specification" ASAM e.V., 2003, ASAM MCD-1 XCP V1.0.0 (<http://www.asam.net>)
- ASAM (Eds.): "XCP Version 1.0 - Part 3 - XCP on Ethernet - Transport Layer Specification" ASAM e.V., 2003, ASAM MCD-1 XCP V1.0.0 (<http://www.asam.net>)
- ASAM (Eds.): "XCP Version 1.0 - Part 3 - XCP on FlexRay - Transport Layer Specification" ASAM e.V., 2005, ASAM MCD-1 XCP V1.0.0 (<http://www.asam.net>)
- ASAM (Eds.): "XCP Version 1.0 - Part 4 - Interface Specification" ASAM e.V., 2003, ASAM MCD-1 XCP V1.0.0 (<http://www.asam.net>)
- ASAM (Eds.): "XCP Version 1.0 - Part 5 - Example Communication Systems" ASAM e.V., 2003, ASAM MCD-1 XCP V1.0.0 (<http://www.asam.net>)

ASAP2 Specifications

- Application Systems Standardization Working Group: "ASAP2: Standardized Description Data" ASAM e.V., Version 1.5.1, 2003, ASAM MCD-2 MC V1.5.1 (<http://www.asam.net>)
- ASAM MCD-2MC Workgroup: "Measurement and Calibration Data Specification" ASAM e.V., Version 1.6, 2009, ASAM MCD-2 MC V1.6.0 (<http://www.asam.net>)

MDF Specification

- ASAM MDF: "Measurement Data Format" ASAM e.V., Version 4.1.0, 2012, ASAM MDF V4.1.0 (<http://www.asam.net>)

Chapter 15

EtherCAT

Superior performance, high flexibility and reliability of a communication system are some essential features of a well developed communication system today. These characteristics are pushed to a high level when engineers talk about EtherCAT (Ethernet for Control Automation Technology). Developed by Beckhoff, EtherCAT was introduced in 2003 and is constantly maintained through the EtherCAT Technology Group (ETG). Finally, the International Electrotechnical Commission (IEC) standardized the technology in 2007.

The quality philosophy of EtherCAT is quite easy to understand. The continuous enhancement of the technology has to stay interoperable, means that there is one EtherCAT version, not many of them. Thereby every device on the market is able to communicate with each other. Furthermore, EtherCAT is based on a communication system which is used in nearly every computer network worldwide, the IEEE 802.3 Ethernet standard. The EtherCAT protocol is basically embedded in the data frame of the Ethernet protocol, already existing hardware can be used. This is why the implementation of EtherCAT stays cost-effective.

IPG increases the application range of Xpack4 with the support of EtherCAT. Automation components can be directly integrated into the CarMaker environment. Typical automotive test systems apply these standardized components for implementing basic operations such as measuring or digital I/O. This integration of standard automation components leads to a quickly built-up, cost-effective and reliable test system.

The first step is to use the Xpack4 system as an EtherCAT Slave. But why? In most application cases, there are already running EtherCAT networks waiting for an automotive integration platform such as CarMaker/HIL to interact with all other slave devices connected to the EtherCAT bus. The master plays a less important role in the communication of process data. He usually runs on a host pc with soft realtime conditions, controlling and monitoring the entire EtherCAT network. The master is mainly responsible for sending/receiving the EtherCAT frame and thus providing a data container for the slaves. Transferring data between the slaves is handled by the master by telling the slaves where to extract data out or put data in the container.

Due to the extensive possibilities of the Xpack4 platform within an automotive test system, EtherCAT is suited best for exchanging a huge amount of process data in a high efficient and convenient way.

15.1 Overview

15.1.1 Basic Information

An EtherCAT network is designed as a master-slave topology. The protocol is optimized for a high amount of process data due to the maximum possible EtherCAT frame data of 1486 bytes. In contrast to the send/receive mechanism in an Ethernet network, an EtherCAT slave device reads the input data addressed to them while the telegram passes through and inserts output data similarly ("processing on the fly"). The frames are only delayed by a few bit times in each slave node and typically the entire network can be served with just one EtherCAT frame sent by the master. This short and predictable delay is achieved by the use of dedicated ASICs in the slaves called EtherCAT Slave Controller (ESC). These chips are separated from any user application and work independently, therefore realtime requirements of an automation network can be kept. According to this, EtherCAT is a representative of the *Realtime Ethernet* family. A basic network configuration is shown in [Figure 15.1](#).



Figure 15.1: EtherCAT network example with Xpack4 as EtherCAT Slave

The network example above is built up as a line topology, but EtherCAT is more flexible and can have many different topologies - at the same time! Due to the use of the 100BASE-TX Fast-Ethernet standard, one cable has both Tx and Rx directions. Every slave has at least two RJ45 connectors. The last device in a line automatically loops back the passing EtherCAT frame (shown at the right side of [Figure 15.1](#)). If a slave have more than 2 connectors, additional branches are possible to extend the bus topology in different ways. But one thing stays the same all time: the actual transfer within the cable behaves like a ring with the master as the start- and endpoint of transferring the EtherCAT frame.

During network startup, the master configures the network using configuration data of all slaves, either provided as offline data with XML-based EtherCAT Slave Information files (ESI) or as online data via service data communication between master and each separate slave. After all slave information has been processed, the master sends initial configuration data via service data objects (SDO) to the slaves containing information about which telegrams of the EtherCAT frame belong to them. Besides the SDO telegrams of an EtherCAT frame, process data is transferred via PDO telegrams (process data object). A PDO itself is organized in a collection of entries, which are registered in an object ditionary within the slave device. They mainly represent the user relevant part for exchanging data between the connected bus devices (TxPDO: slave sends telegram, RxPDO: slave receives telegram).

Each slave has its own EtherCAT State Machine (ESM). The transition between the states is fully controlled by the master and allows a reliable operation of the communication system. The four main states of the ESM are:

- Init (INIT): Master accesses the slave registers and configures the SDO mailbox
- Pre-Operational (PREOP): Master configures process data communication via SDO.
- Safe-Operational (SAFEOP): Slaves can receive data, output is in safe state (output data will not be evaluated by the master).
- Operational (OP): Slave outputs get valid and will be evaluated by the master.

Figure 15.2 shows an EtherCAT network example. The upper left part represents the Control station of the EtherCAT network: Beckhoff TwinCAT software running on a host computer for configuring, controlling and monitoring the communication network and the dedicated CX9020 CPU module for providing realtime capabilities. The network configuration is based on the collection of all ESI files of the connected slave devices. An EtherCAT Configuration Tool, which is often integrated in an EtherCAT Master Tool Suite like Beckhoff's TwinCAT, parses the slave information and generates the final network information file called ENI (EtherCAT Network Information). This file is used by the master driver for further handling of the network.

In contrast to having a fixed ESI file like simple EtherCAT devices, the ESI file of the Xpack4 system has to be created dynamically because of changing process data. For this purpose, the ECATParameters infofile is read in during startup containing information about all Tx- and RxPDOs, which are relevant to the CarMaker/HIL application. The CIFXParameters infofile contains hardware specific information about all installed EtherCAT modules (cifX cards) and should be configured first. Data is exchanged in the IO module of CarMaker, the Data Dictionary is connected to it for mapping PDO entries and CarMaker model quantities and connecting them to the APO clients like CarMaker GUI or IPG Control. DVA access is possible, too.

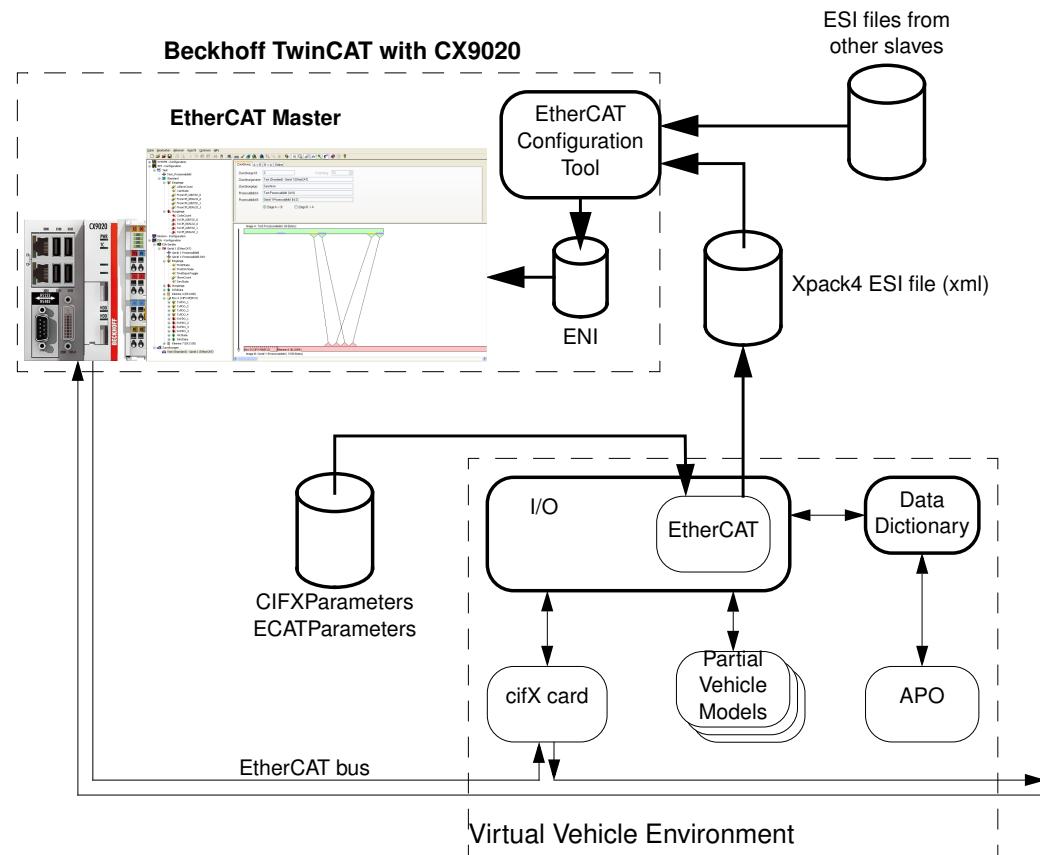


Figure 15.2: EtherCAT network example with Beckhoff master and Xpack4 slave

15.1.2 Features

- Supported hardware module:
 - Hilscher cifX 80-RE Compact PCI card with netX100 chip, EtherCAT Slave Firmware v4.2.0 and cifX device driver v1.0.3
 - Direct Memory Access (DMA enabled)
 - Interrupt controlled (IRQ enabled)

- Dynamic generation of the EtherCAT Slave Information file (ESI) during startup of Car-Maker/HIL application, supported EtherCAT device description version: v1.2
- Automatic update of Slave Information Interface (SII) and Object Dictionary (OD) during startup of CarMaker/HIL application, no need to configure the content with the master.
- PDOs are configured with a CarMaker infofile (s. [section 15.4.2 'ECATParameters'](#)), hence there is no need to recompile the CarMaker/HIL application if the configuration changes. Hardware specific parameters are also configured with an infofile, see [section 15.4.1 'CIFXParameters'](#) for more details.
- Use of modular device profile MDP5001 (standardized modeling of structures within an EtherCAT slave device). Mainly the object dictionary is structured with the intention to provide easy access to the device by the master. The EtherCAT Slave integration in Xpack4 is IPG specific, therefore no standardized device profile (e.g. CiA 402 for motion control) can be applied. As a result, only the common model of a modular device profile is used and mandatory objects are created. See ETG5001 standardization for more details.
- A cycle update rate can be applied to each PDO to reduce PCI bus and CPU load.
- The infofile configuration allows mapping between PDO entries and CarMaker model quantities with optional calculation parameters factor, offset, minimum and maximum.
- Supported data types for PDO entries are:
 - char, unsigned char (8 bit)
 - short, unsigned short (16 bit)
 - int, unsigned int (32 bit)
 - long long, unsigned long long (64 bit)
 - float (32bit single precision)
 - double (64 bit double precision)
- CAN over EtherCAT (CoE): Acyclic SDO communication is handled by a second thread to provide realtime capability. Following mechanisms are supported:
 - SDO, SDO information, PDO upload at startup
- APO access to EtherCAT Data Dictionary quantities (PDO entries).
- Multiple card support

15.1.3 Limitations

- Maximum number of cyclic input/output process data: 512 bytes in sum
- Due to the MDP5001 device profile, the number of PDOs and their entries are limited as follows:
 - Maximum number of TxPDOs: 256
 - Maximum number of RxPDOs: 256
 - Maximum number of entries per PDO: 16
- Logical ReadWrite (LRW) is not supported (PDOs for CarMaker/HIL have to be configured either as Tx- or RxPDO, but not both at the same time). This limitation only affect the send/receive mechanism of the master.
- Only data types of the CarMaker Data Dictionary are supported for PDO entries (s. [section 15.1.2 'Features'](#) for supported data types)

15.2 EtherCAT Implementation in CarMaker/HIL

15.2.1 Hardware Integration of EtherCAT in Xpack4

The Hilscher cifX 80-RE module is a Compact PCI card, which is distributed by MEN with the module name F752. The card is uniquely identified by its two serial numbers:

- MEN serial: 6 digit number, printed on the board beneath a data matrix code. This number can be also found on your delivery note and is only important for further support.
- Hilscher serial: 5 digit number, printed on the board beneath the barcode. This number is permanently integrated in the module and is necessary for the hardware configuration and identification.

The driver of the cifX card is splitted in an userspace library and a kernel module. The library is called `libcifx` and is provided as a static library in the CarMaker/HIL installation directory. This library requires `libpciaccess`, which is delivered in version 0.13.1 as shared object with the network file system `xeno_rt`. In summary, the NFS is extended by following additional files:

- Kernel module: `.../xeno_rt/lib/modules/$(KERNEL_VER)/uio_netx.ko`
 - `KERNEL_VER` is the kernel version of the Xpack4 system. Type in `uname -r` to get it (e.g. `3.14.34-ipipe32`).
- `libpciaccess: .../xeno_rt/lib/libpciaccess.so`
- Bootloader: `.../xeno_rt/lib/firmware/cifx/BSL/NETX100-BSL.bin`
- Firmware: `.../xeno_rt/lib/firmware/cifx/ECS V4.X/cifxececs.nxf`

The card needs a special directory structure to find the required files for the intended start-up behaviour. This `base directory` is automatically & temporarily created during the initialization phase of the CarMaker/HIL application and is located at `.../xeno_rt/tmp/cifx`. Bootloader and firmware are copied to this location. The structure of the `base directory` looks like follows:

- Bootloader `.../$(BASE_DIR)/NETX100-BSL.bin`
- Log file `.../$(BASE_DIR)/cifx0.log`
- Config file `.../$(BASE_DIR)/deviceonconfig/1280100/$(SERIAL)/device.conf`
- Firmware `.../$(BASE_DIR)/deviceonconfig/1280100/$(SERIAL)/channel10/cifxececs.nxf`

The basic netX operating system of the card runs the bootloader, determining which firmware has to be loaded. The cifX card is able to load different firmware files. In our case, `cifxececs.nxf` extends the netX OS with the EtherCAT Slave functionality. A log file `cifx<nC>.log` is created by the driver for each card, where `<nC>` is the card number. The config file implies an alias name of the card and enabled features like DMA and interrupt control. The base directory needs the serial numbers of the used cards, so that it is possible to individually set the firmware and configuration for each card. Just run `mioutil` to get the Hilscher serial number.

All the information are stored in parameters and must be configured in an infofile (s. [section 15.4.1 'CIFXParameters'](#) for the cifX hardware configuration).

15.2.2 Integration of EtherCAT in a CarMaker Project

The project template for CarMaker/HIL is already prepared for EtherCAT network communication with the cifX card. However, EtherCAT is not active by default. It needs to be enabled in the Makefile, followed by a rebuild of the realtime application.

Makefile

On compilation, the preprocessor macro `WITH_ECAT` is used to activate the EtherCAT communication. In the Makefile of the project template, there is already a line which adds the macro `WITH_ECAT` to the Makefile variable `DEF_CFLAGS`, but it is commented out:

```
### EtherCAT
#DEF_CFLAGS += -DWITH_ECAT
```

By activating the line, different code sections in the modules `IO.c` and `User.c` are enabled, which handle the initialization of the cifX card and manage the EtherCAT communication.

15.2.3 Initialization of the EtherCAT Framework

Basic Initialization

Basic initialization is done in the module `IO.c` in three steps:

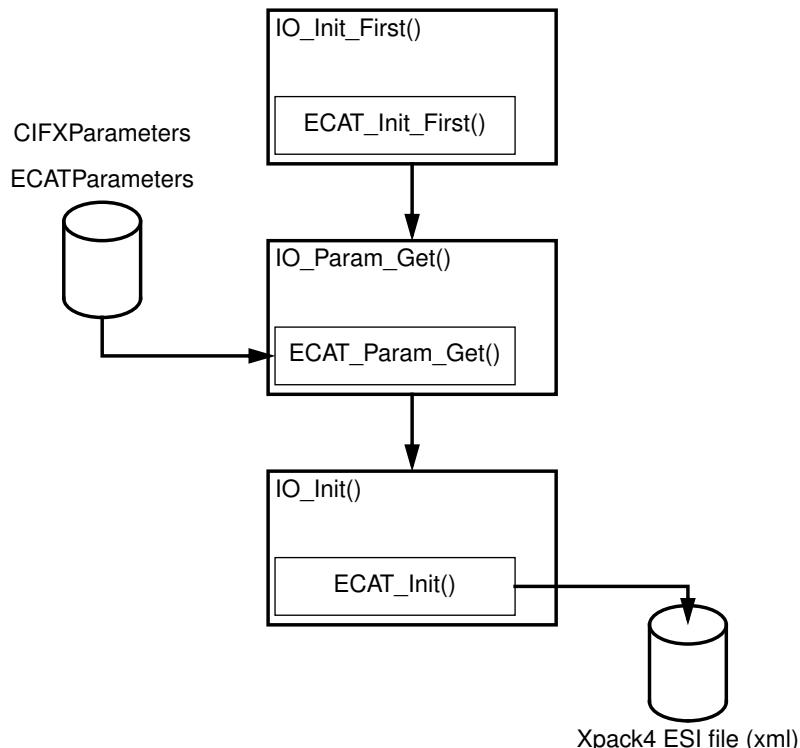


Figure 15.3: Basic initialization

In `IO_Init_First()`, the function `ECAT_Init_First()` is called to initialize application internally resources and to create a second thread, which handles the SDO communication of the Xpack4 EtherCAT slave. According to this, the creation of a semaphore allows to trigger the SDO thread each time `ECAT_In()` is called.

The configuration is performed in `IO_Param_Get()`: `ECAT_Param_Get()` reads the cifX card configuration (s. [section 15.4.1 'CIFXParameters'](#)) and identifies the EtherCAT interfaces which are used by the application. The function then reads the PDO configuration for the Xpack4 slave (s. [section 15.4.2 'ECATParameters'](#)).

In `IO_Init()`, the function `ECAT_Init()` is called in order to configure the EtherCAT interface which should be used and to prepare for startup (currently only one module is supported). First, the cifX driver is initialized and CarMaker gets connected to the card. The PDO information, obtained from the `ECATParameters` file, is used to update the slave information interface (SII). This has to be done every start-up, because the SII memory on the cifX card

is only an emulated EEPROM with predefined content. Based on the SII, objects will be created in a standardized object dictionary, so information can be easily read from the master via SDO.

Data Dictionary Quantities

During the initialization of the EtherCAT interface, the function [ECAT_Init\(\)](#) declares quantities for the EtherCAT PDO entries. The purpose of this is to provide access to all EtherCAT signals as defined in the [ECATParameters](#) file.

All generated Data Dictionary quantities follow the naming scheme below:

<Prefix>.<PDOName>.<EntryName>

Prefix	Prefix for identification of the communication network (default: ECAT)
PDOName	Name of the PDO, that contains several entries.
EntryName	Name of the entry.

Completion of Initialization

The EtherCAT initialization is completed in `IO_Init_Finalize()`:

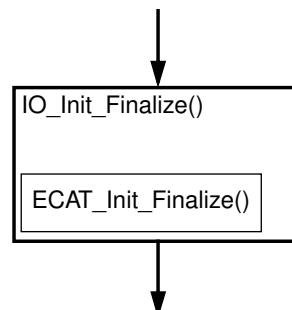


Figure 15.4: Completion of initialization

The function [ECAT_Init_Finalize\(\)](#) maps CarMaker model quantities and EtherCAT PDO entries as defined in the [ECATParameters](#) file. Finally, the bus state is set to on and the communication begins.

15.2.4 Online Communication

Input Processing

EtherCAT telegrams are received and processed in `IO_In()` at the beginning of each simulation cycle:

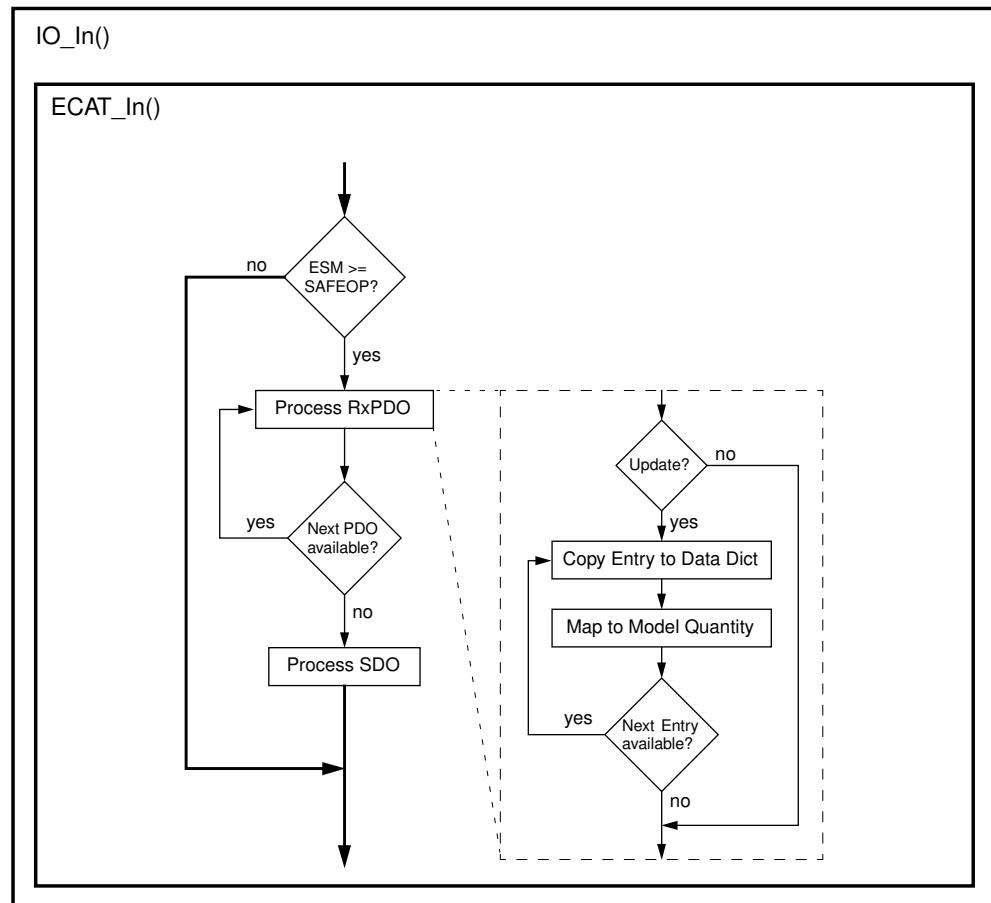


Figure 15.5: Input Processing

First, the function `ECAT_In()` checks if the ESM allows to read input data from the DPM of the cifX card. As soon as the state reaches `SAFEOP`, the function fetches the list of RxPDOs. The current RxPDO will be updated when the CarMaker cycle number is a multiple of the PDO update rate as defined in `ECATParameters`, otherwise the next PDO will be processed. The values of the PDO entries will be copied to the created Data Dictionary quantities one by one. If a mapping is applied as defined in `ECATParameters`, an internal function overwrites the value of the specified model quantity. Here the data type conversion is done automatically and the user does not have to worry about matching data types. After processing the RxPDOs, the semaphore counter of the dedicated SDO thread gets incremented, so that the thread is allowed to handle SDO communication each CarMaker cycle. The SDO processing is mainly responsible for responding to ESM state change requests. The SDO communication has to be handled in a second thread because of blocking function calls which can not guarantee hard realtime behaviour.

Output Processing

For output of simulation results to the EtherCAT bus, the functions `User_Out()` and `IO_Out()` are involved:

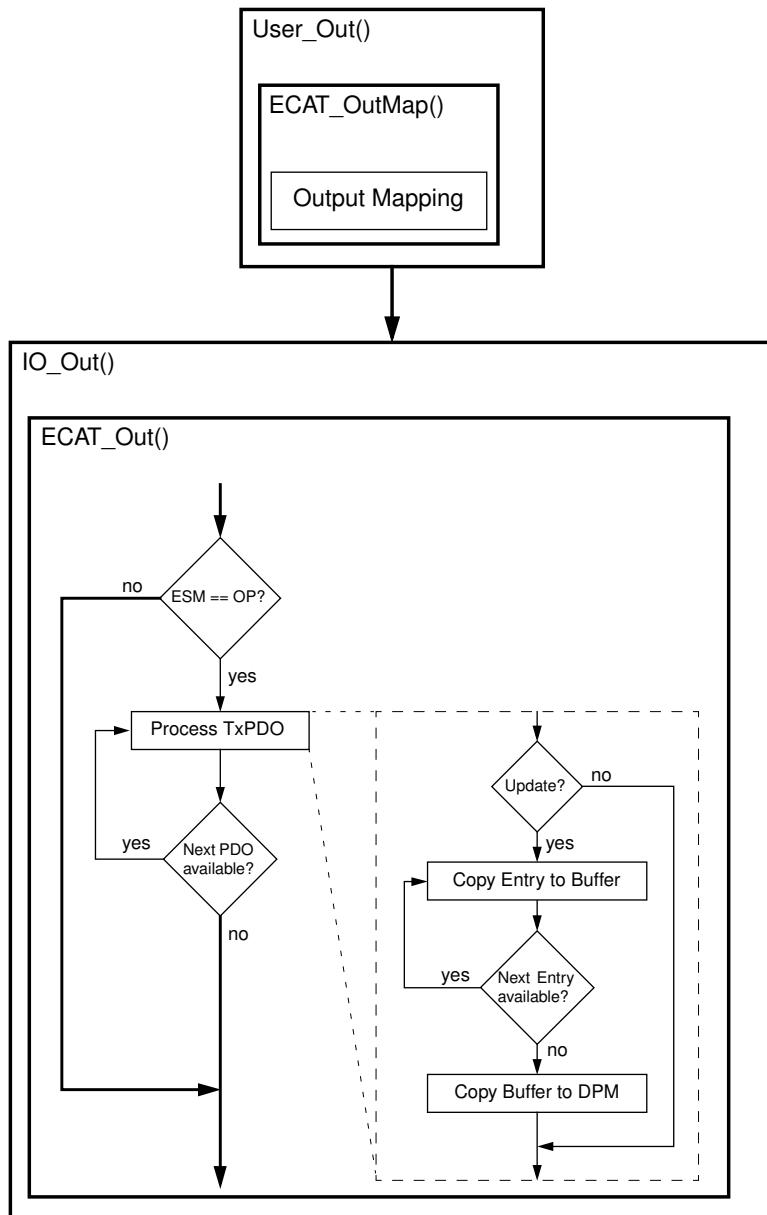


Figure 15.6: Output Processing

In `User_Out()` the function `ECAT_OutMap()` is called, which performs the mapping between recently updated model quantities and the EtherCAT PDO entry variables.

At the end of the simulation cycle – in `IO_Out()` – `ECAT_Out()` is called. First, the function checks if the ESM allows to write output data to the DPM of the cifX card. As soon as the state reaches OP, the function fetches the list of TxPDOs. The current TxPDO will be processed when the CarMaker cycle number is a multiple of the PDO update rate as defined in `ECATParameters`, otherwise the next PDO will be processed if available. The values of the PDO entries will be copied to a buffer one by one. Finally, the buffer will be copied to the DPM of the cifX card.

15.3 Functional Description

15.3.1 EtherCAT Function Interface

Configuration / Initialization

ECAT_Init_First ()

```
int ECAT_Init_First (void);
```

Description

First (early) initialization of the EtherCAT communication library. Internal application resources are initialized and a second thread called `ecat_sdo` is created and started. According to this, a semaphore is created to trigger the SDO thread each time `ECAT_In ()` is called.

This function has to be called prior to all other `ECAT_*`() functions.

Return Value

- 0, if the first initialization succeeds.
- Otherwise: <0. Additional information about the reason is written into the session log.

ECAT_Param_Get ()

```
int ECAT_Param_Get (struct tInfos *inf, const char *pfx);
```

Description

Reads the EtherCAT configuration from the given infofile handle `inf`. If a prefix is passed, too (`pfx != NULL`), then “`pfx.`” will be prepended to all keys which are read from the given infofile handle `inf`.

First, `ECAT_Param_Get ()` tries to read the key `CIFXParameters`. If `inf` is not NULL and the key exists, then its value is interpreted as the name of another infofile with the FileIdent CarMaker-CIFXParameters. All configuration parameters belonging to the card initialization will then be read from this file. If `inf` is NULL, then a file with name `CIFXParameters` will be searched. The format of the `CIFXParameters` file is described in [section 15.4.1 'CIFXParameters'](#). The cifX card configuartion failes if `inf` is not NULL and if the key `CIFXParameters` does not exist.

Second, `ECAT_Param_Get ()` tries to read the key `ECATParameters`. If `inf` is not NULL and the key exists, then its value is interpreted as the name of another infofile with the FileIdent CarMaker-ECATParameters. All PDO configuration will then be read from this file. If `inf` is NULL, then a file with name `ECATParameters` will be searched. The format of the `ECATParameters` file is described in [section 15.4.2 'ECATParameters'](#). The PDO configuartion failes if `inf` is not NULL and if the key `ECATParameters` does not exist.

Return Value

- 0, if all cifX card parameters and PDO configuration have been successfully read.
- Otherwise: <0. Additional information about the reason is written into the session log.

ECAT_Init ()

```
int ECAT_Init (void);
```

Description

Initialize the cifX driver and opens a cifX card. The PDO information obtained from the [ECATParameters](#) file is used to update the slave information interface (SII). Based on the SII content, objects will be created in a standardized object dictionary, so information can be easily read from the master via SDO. Finally, the function generates Data Dictionary quantities for each PDO entry.

Return Value

- 0, if the cifX driver and the cifX card have been successfully initialized, SII and object dictionary are updated and Data Dictionary quantities are generated for each PDO entry.
- Otherwise: <0. Additional information about the reason is written into the session log.

Startup / Stop of EtherCAT Communication

ECAT_Init_Finalize ()

```
int ECAT_Init_Finalize (void);
```

Description

The function [ECAT_Init_Finalize \(\)](#) maps CarMaker model quantities and EtherCAT PDO entries as defined in the [ECATParameters](#) file. Finally, the bus state is set to on and the communication begins.

Return Value

- 0, if all CarMaker model quantities are found in the Data Dictionary and the communication could be successfully started on the cifX card.
- Otherwise: <0. Additional information about the reason is written into the session log.

ECAT_Cleanup ()

```
int ECAT_Cleanup (void);
```

Description

Stops the EtherCAT communication of the cifX card, terminates the SDO thread, deinitializes the driver and frees all internally used resources.

Return Value

- 0, if terminating the CarMaker/HIL application and deinitializing the hardware were successful.
- Otherwise: <0. Additional information about the reason is written into the session log.

Cyclic Calculations (realtime)

ECAT_In ()

```
int ECAT_In (unsigned CycleNo);
```

Description

This function goes through the list of RxPDOs and copies them into a buffer individually, if following conditions are satisfied:

- The EtherCAT State Machine (ESM) is set to `SAFEOP` or `OP`.
- The cycle number `CycleNo` is a multiple of the update rate of the PDO (set in the [ECAT-Parameters](#) file for each PDO).

After that, the entries are extracted from the PDO and copied to the generated Data Dictionary quantities. If a mapping of an entry exists (defined in [ECATParameters](#)), the value of the entry will then be copied to a CarMaker model quantity with the given calculation parameters (factor, offset, min, max).

Parameters

- `CycleNo` is the current simulation cycle of the realtime application.

Return Value

- 0, if all PDOs were successfully received and processed.
- Otherwise: <0.

ECAT_OutMap ()

```
int ECAT_OutMap (unsigned CycleNo);
```

Description

After the calculation step of the model but before I/O output, this function needs to be called in each simulation step to update all PDO entry variables with new values from mapped model variables, if a mapping is defined for this entry in [ECATParameters](#). Call this function at the end of `User_Calc()` before calling `IO_Out()`.

Parameters

`CycleNo` is the current simulation cycle of the realtime application.

Return Value

- 0, if the frame was sent successfully.
- Otherwise: <0.

ECAT_Out ()

```
int ECAT_Out (unsigned CycleNo);
```

Description

This function goes through the list of TxPDOs and copies them into the DPM of the cifX card individually, if following conditions are satisfied:

- The EtherCAT State Machine (ESM) is set to `OP`.
- The cycle number `CycleNo` is a multiple of the update rate of the PDO (set in the [ECAT-Parameters](#) file for each PDO).

Parameters

- `CycleNo` is the current simulation cycle of the realtime application.

Return Value

- 0, if all PDOs were successfully processed and sent.
- Otherwise: <0.

Manual Access to PDO Entries

ECAT_GetPdo ()

```
tECAT_Pdo* ECAT_GetPdo (char *chName, char *pdoName);
```

Description

This function returns a pointer to a PDO structure of type `tECAT_Pdo` (s. [section 15.5.1 'EtherCAT_Slave.h'](#) for the definition of the structure). The function goes through the list of all Tx- and RxPDOs of one channel specified by `chName` and returns a pointer to the PDO with the given `pdoName`. This pointer is used to get access to the PDO entries (s. [ECAT_GetPdoEntry \(\)](#)).

Parameters

- `chName` contains the name of the channel, to which the PDO is assigned. The assignment is done in [ECATParameters](#).
- `pdoName` contains the name of the PDO as a string, whose pointer should be returned by this function.

Return Value

- A pointer to the PDO with the given name.
- Otherwise: NULL. Additional information about the reason is written into the session log.

ECAT_GetPdoEntry ()

```
tECAT_PdoEntry* ECAT_GetPdo (tECAT_Pdo *pdo, char *entryName);
```

Description

This function returns a pointer to a PDO entry structure of type `tECAT_PdoEntry` (s. [section 15.5.1 'EtherCAT_Slave.h'](#) for the definition of the structure). The function goes through the list of all entries within the specified `pdo` and returns a pointer to the PDO entry with the given `entryName`. This function is used to get access to the PDO entries, for example reading/writing from/to the entry variables.

Parameters

- `pdo` points to a PDO structure of type `tECAT_Pdo`. This pointer is obtained by [ECAT_GetPdo \(\)](#), which must be called before this function.
- `entryName` contains the name of the PDO entry as a string, whose pointer should be returned by this function.

Return Value

- A pointer to the PDO entry with the given name.
- Otherwise: NULL. Additional information about the reason is written into the session log.

15.4 Configuration Files

15.4.1 CIFXParameters

The **CIFXParameters** infofile provides the hardware configuration for the installed cifX cards, which should be integrated in CarMaker/HIL. The listed parameters below have a common naming, starting with `CIFX` as the base key, followed by `nC` representing the card number of the desired hardware. It is recommended to only configure the serial number. If more than one card are used, the alias name must also be different for a uniquely identification of each card.

Header Information

FileIdent = CarMaker-CIFXParameters Ver

Identifies the infofile as cifX parameters file of version `Ver`. Currently supported value for `Ver` is 1.

CifX Card Parameters

CIFX.BaseDir = <BaseDir>

Path to firmware and interface configurations within the network file system. This absolute path must be the directory, where the bootloader `NETX100-BSL.bin` is stored in. The base directory will be created with all necessary subfolders if not available. The default base directory is `/tmp/cifx`, so it will be only created temporarily.

CIFX.LogLvl = <LogLvl>

This parameter specifies a bit mask for enabling several debugging levels, whose outputs are logged to `.../BaseDir/cifx<nC>.log` (where `<nC>` is the card number). If logging is disabled or fails, outputs are printed to console output. The `LogLvl` can also be any bit mask combination of the values in the table below, the default level is `0x0E` (all messages except of debugging are logged).

0x00	Tracing disabled (logging printed to console output)
0x01	Debug messages will be logged
0x02	Information messages will be logged
0x04	Warning messages will be logged
0x08	Error messages will be logged
0xFF	All messages will be logged

CIFX.<nC>.BSL = <BSL>

Name of the bootloader which should be used during the startup phase of the cifX driver.
The default bootloader is NETX100-BSL.bin.

CIFX.<nC>.Serial = <Serial>

This is the Hilscher serial number (not MEN) and is printed on the hardware module as a 5 digit number beneath the barcode. Just run mioutil to get the Hilscher serial. [Listing 15.1](#) shows the output of mioutil, providing some basic information if one or more cifX cards are installed in CarMaker/HIL:

Listing 15.1: List of found cifX cards

```
1: [rtl-hil] 1) mioutil
2:
3: =====
4: * X E           Hilscher RT Ethernet interfaces (www.hilscher.com)
5: * N O
6: * M A I         2 RT-interfaces for XENOMAI found
7: =====
8:
9: * n   Mxxx   Serial   Hardware Rev.   Device
10:  0    M752   12345    3             CIFX 80-RE
11:  1    M752   67890    3             CIFX 80-RE
12:
```

CIFX.<nC>.FW = <FW>

Name of the firmware which should be loaded during the startup phase of the cifX driver.
The default firmware is cifxecl.nxf.

CIFX.<nC>.Alias = <Alias>

Alias name of the communication channel. Since the driver currently supports only one channel per card, this alias could be interpreted as a name identifier for the cifX card. If there is only one cifX card in use, the default alias name `XpackECAT` can be applied to it. The PDO configuration of [ECATParameters](#) references to this alias name for assigning the configuration to the right hardware.

CIFX.<nC>.DMA = <DMA>

Enable (1) or disable (0) the Direct Memory Access mechanism of the cifX card. This parameter is enabled by default.

CIFX.<nC>.IRQ = <IRQ>

Enable (1) or disable (0) the Interrupt control mechanism of the cifX card. This parameter is enabled by default.

15.4.2 ECATParameters

The [ECATParameters](#) infofile provides the PDO configuration which should be assigned to available cifX cards. For a better readability, only the RxPdo key parameters are described with this chapter, but refer to TxPdo as well (just replace Rx with Tx).

The PDO configurations have following subkeys in common to create PDOs, PDO Entries, mapping tables and description strings:

<nSet>	Number of the parameter set defined in ECATParameters . You are allowed to define more parameter sets than needed. The assignment of a parameter set to a specific card is done with <code>ECAT.<nSet>.Channel</code> .
<nRxPdo>	Number of RxPdo (starts counting with 0 for each parameter set and PDO type).

Header Information

FileIdent = CarMaker-ECATParameters Ver

Identifies the infofile as EtherCAT PDO configuration file of version `Ver`. Currently supported value for `Ver` is 1.

Configuration Parameters

ECAT.<nSet>.Channel =<Channel>

Channel name, to which the PDO configuration should be assigned. At this time, cifX cards only support one channel, so the string also represents a user defined unique identifier for the hardware (like the serial number). The available channel names can be obtained under the keys `CIFX.<nC>.Alias` of the CIFXParameters infofile. If `ECAT.<nSet>.Channel` or `CIFX.<nC>.Alias` are missing or the alias names of various cards are identically, the incorrectly configured cards get their serial as the alias name automatically by putting a warn message to the session log.

ECAT.<nSet>.ESI =<ESI-Filename>

Name of the xml slave description file, which will be created in `.../PROJECTDIR/Data/Config` during the startup phase of a CarMaker/HIL application. The EtherCAT Master needs this file for offline configuration. See [section 15.4.3 'EtherCAT Slave Information'](#) for some basic information of the file content.

ECAT.<nSet>.RxPdo.<nRxPdo> =<PdoUpdateRate> <PdoName>

Creates a PDO within the parameters set `<nSet>` with the number `<nRxPdo>`. `<PdoUpdateRate>` is a positive integer (greater than 0) and defines the rate of receiving new PDO data depending on the CarMaker cycle time. A PDO update rate of 2 means for example, that PDO data is copied from the card's DPM to the internal memory every second cycle. The second parameter is interpreted as the PDO name (no whitespaces allowed).

ECAT.<nSet>.RxPdo.<nRxPdo>.Entries:

<code><EntryName></code>	<code><Unit></code>	<code><DataType></code>
<code><EntryName></code>	<code><Unit></code>	<code><DataType></code>

The PDO Entries of a specific PDO are described in this entry table. All entry names, represented by `<EntryName>`, have to be different within an entry table. With `<Unit>` the user can add a string to the scalar quantity of the entry. If the entry should not have a unit, use “-” as a placeholder. `<DataType>` determines the size of the entry within a PDO and affects the frame length of an EtherCAT telegram.

The following table shows the supported data types:

UNSIGNED8	unsigned integer, 8 bits
INTEGER8	signed integer, 8 bits
UNSIGNED16	unsigned integer, 16 bits
INTEGER16	signed integer, 16 bits
UNSIGNED32	unsigned integer, 32 bits
INTEGER32	signed integer, 32 bits
UNSIGNED64	unsigned integer, 64 bits
INTEGER64	signed integer, 64 bits
REAL32	single precision floating point, 32 bit
REAL64	double precision floating point, 64 bit

ECAT.<nSet>.RxPdo.<nRxPdo>.Map:

<code><EntryName></code>	<code><MapVar></code>	<code><[Factor [Offset [Min [Max]]]]></code>
<code><EntryName></code>	<code><MapVar></code>	<code><[Factor [Offset [Min [Max]]]]></code>

This table allows to map a PDO entry with a CarMaker model quantity. Only the desired entries have to be specified here. `<MapVar>` must be a quantity which has been added to the Data Dictionary before `ECAT_Init_Finalize()` is called. `<Factor>`, `<Offset>`, `<Min>` and `<Max>` are optional calculation parameters and are interpreted as follows:

- RxPdo:
 - $\text{MapVar} = \text{EntryVar} * \text{Factor} + \text{Offset}$
- TxPdo:
 - $\text{EntryVar} = \text{MapVar} * \text{Factor} + \text{Offset}$

`Min` and `Max` are applied after the calculation.

ECAT.<nSet>.RxPdo.<nRxPdo>.<EntryName>.Desc = Description

Description strings can be applied to a PDO Entry if desired. This string is also visible in the EtherCAT Slave Information file as a PDO entry element called <Comment>.

15.4.3 EtherCAT Slave Information

After the PDO configuration is done with the `ECATParameters` file for each cifX card, a xml file is created during the startup phase of the CarMaker/HIL application in `ECAT_Init()`. The filename can be changed in `ECATParameters`. The file is necessary, if the EtherCAT Master configures the communication network in offline mode. All EtherCAT Slave Information (ESI) files are then processed by the master (or a dedicated network configuration tool). Finally, an EtherCAT Network Information (ENI) file is generated containing the slave information of the entire network. These files following a standardized structure, which is defined by the EtherCAT Technology Group. The standard ETG2000 delivers the ESI structure whereas ETG2100 specifies the ENI (s. <https://www.ethercat.org> for more information). Listing 15.2 shows the structure of an ESI file with the mandatory elements up to the third subelement layer. `<Vendor>` delivers information about the device vendor with name and vendor ID (IPG: 0x507). `<Groups>` provides some common information, which can be assigned to all `<Devices>`. The specific slave information is listed in element `<Device>`. There you can find the PDO configuration of `ECATParameters` as well as used data types, hardware information and configuration of the EtherCAT Slave Controller (ESC).

Listing 15.2: Basic structure of the ESI file in xml format

```
1:  <?xml version="1.0" encoding="utf-8"?>
2:  <EtherCATInfo>
3:      <Vendor>
4:          </Vendor>
5:      <Descriptions>
6:          <Groups>
7:              <Group>
8:                  </Group>
9:          </Groups>
10:         <Devices>
11:             <Device>
12:                 </Device>
13:             </Devices>
14:         </Descriptions>
15:     </EtherCATInfo>
```

15.5 Header Files

15.5.1 EtherCAT_Slave.h

All user accessible functions are listed below in [Listing 15.3](#). These functions form the EtherCAT Slave API and must be called within the CarMaker/HIL application (s. [section 15.3 'Functional Description'](#)).

Listing 15.3: EtherCAT API functions defined in EtherCAT_Slave.h

```
1: tECAT_Pdo*      ECAT_GetPdo          (char* chName, char* pdoName);
2: tECAT_PdoEntry* ECAT_GetPdoEntry    (tECAT_Pdo* pdo, char* entryName);
3: int             ECAT_Init_First     (void);
4: int             ECAT_Param_Get      (struct tInfos* inf, const char* pfx);
5: int             ECAT_Init          (void);
6: int             ECAT_Init_Finalize  (void);
7: int             ECAT_In            (unsigned CycleNo);
8: int             ECAT_OutMap         (unsigned CycleNo);
9: int             ECAT_Out           (unsigned CycleNo);
10: int            ECAT_Cleanup        (void);
```

The EtherCAT PDOs are summarized in the global structure `tECAT` (s. [Listing 15.5](#)). This structure contains two singly-linked lists for Tx- and RxPDOs, pointing to structures of type `tECAT_Pdo`. `NumRxPdo` and `NumTxPdo` represent the total numbers of PDOs defined in [ECAT-Parameters](#). The `tECAT_Pdo` structure stores the user configured PDO name, the update rate and other useful information. The element `entryList` points to a further list of type `tECAT_PdoEntry`, which contains all the PDO entry data for each PDO.

Finally, here is an example for reading from/writing to PDO entries in user code:

- Channel name: “Xpack4Slave0”
- PDO: “Status_Motor”
- PDO Entries: “RPM_MotorFL” (double), “State_MotorFL” (unsigned char)

Listing 15.4: Example for accessing PDO entries in user code

```
1: tECAT_Pdo*      pdo      = ECAT_GetPdo("Xpack4Slave0", "Status_Motor");
2: tECAT_PdoEntry* entryRpmFL = ECAT_GetPdoEntry(pdo, "RPM_MotorFL");
3: tECAT_PdoEntry* entryStaFL = ECAT_GetPdoEntry(pdo, "State_MotorFL");
4:
5: if (entryStaFL->ucVar == Motor_OK)
6:     entryRpmFL->dVar = UpdateRPM(MotorFL);
```

Listing 15.5: EtherCAT structures defined in EtherCAT_Slave.h

```
1:  typedef struct EcatMap
2:  {
3:      char*      quantName;
4:      tDDictEntry* ddictEntry;
5:      unsigned char Mask;
6:      double      Factor;
7:      double      Offset;
8:      double      Min;
9:      double      Max;
10: } tEcatMap;
11:
12: typedef struct ECAT_PdoEntry* nextECAT_PdoEntry;
13: typedef struct ECAT_PdoEntry
14: {
15:     unsigned BitLen;
16:     unsigned BitOffset; /* in pdo */
17:     char*      name;
18:     char*      unit;
19:     tDDType    DDTType;
20:     tEcatMap*  map;
21:     char*      comment;
22:     union
23:     {
24:         double      dVar;
25:         float       fVar;
26:         long long   llVar;
27:         unsigned long long ullVar;
28:         int         iVar;
29:         unsigned    uiVar;
30:         short       sVar;
31:         unsigned short usVar;
32:         char        cVar;
33:         unsigned char ucVar;
34:     };
35:     nextECAT_PdoEntry next;
36: } tECAT_PdoEntry;
37:
38: typedef struct ECAT_Pdo* nextECAT_Pdo;
39: typedef struct ECAT_Pdo
40: {
41:     unsigned      ByteLen; /* round up for byte-alignment */
42:     unsigned      ByteOffset; /* in DPM Area */
43:     unsigned      UpdateRate; /* number of main cycles */
44:     char*      name;
45:     unsigned char NumEntries;
46:     tECAT_PdoEntry* entryList;
47:     nextECAT_Pdo  next;
48: } tECAT_Pdo;
49:
50: typedef struct ECAT
51: {
52:     unsigned      NumRxPdo;
53:     tECAT_Pdo*   rxpdoList;
54:     unsigned      NumTxPdo;
55:     tECAT_Pdo*   txpdoList;
56: } tECAT;
```

Chapter 16

CANiogen – CANdb Import Tool

Connecting an ECU to CarMaker/HIL normally requires the application engineer to code all CAN messages that have to be transferred between the ECU and the simulation program. This part is realized in the module `IO.c` through the functions `IO_In()` and `IO_Out()`. In `IO_In()`, CAN signals are extracted from received CAN messages and its values are assigned to associated input quantities. In `IO_Out()`, respectively, values of output quantities are coded into CAN messages, which are transmitted to the ECU.

However, coding the handling of CAN messages by hand takes a lot of time and is error-prone. If you have either a vector CAN database (CANdb) or a FIBEX data base (XML), where the ECU is modeled within its surrounding CAN network, you can automate this step with CANiogen.

16.1 Overview

16.1.1 Basic information

CANiogen is a command line tool, which is intended to be called automatically during the compilation process of CarMaker/HIL. This guarantees that changes to the CAN data base (vector CANdb or FIBEX) and/or the simulation program will affect the behavior of CarMaker/HIL.

As input, CANiogen accepts a vector CANdb file, or a FIBEX XML data base (version 2.0.1 or 3.1). This file models a CAN network with all its network nodes (ECU) by defining CAN messages and CAN signals, which are transferred between the ECUs. Depending on the test stand, there are ECUs which are included as hardware components. Others might be realized in software, which means that their behavior is simulated by CarMaker/HIL.

The user now has the choice, to either call CANiogen with a list of ECUs which should be served by CarMaker/HIL. CANiogen will then assume, that these ECUs build the real part of the CAN network, and will import all information from the CAN database which is necessary to simulate the rest of the CAN network. Or, the user gives a list of simulated ECUs that tells CANiogen, which ECUs build the simulated part of the CAN network. CANiogen will then assume all other ECUs to be realized in hardware. In substitution for all ECUs located in the simulated part of the CAN network, CarMaker/HIL will handle all CAN messages (both, reception and transmission), which these ECUs exchange with real network nodes. The user can also decide to receive or send additional single messages, as well as he can choose to receive additional single signals, which are not automatically included by CANiogen. All these information are entirely extracted from the given CAN database.

By default, CANiogen generates C-code as output, which is recommended for most applications. This C-code is capable of handling all CAN traffic, as configured with command line arguments. For use with J1939 based rest bus simulation in CarMaker/HIL, CANiogen offers a second J1939 mode. In this mode, CANiogen generates a description of the J1939 bus in infofile format. However, a special CAN database with J1939 descriptions is required (vector CANdb file with J1939 attributes).

C-Generator Mode

In this mode, the output generated by CANiogen are isolated C-modules, which can be easily integrated in CarMaker/HIL. The user only has to add about 5 lines of source code in the module *IO.c*, to activate the generated code. Additional lines need to be adapted for proper assignment between the generated I/O quantities and their associated model variables. Compared to the conventional method – coding everything by hand – the amount of code the user is responsible for, stays much more manageable.

The C-modules generated by CANiogen (typically called *IO_CAN*), will be represented through the files *IO_CAN.c*, *IO_CAN.h*, *IO_CAN_User.c*, *IO_CAN_User.h* and *IO_CAN_VarList.txt*. The header file *IO_CAN.h* defines a set of macros and fast inline functions to encode/decode signals of CAN messages and declares several public functions, that have to be called in *IO.c*. Each imported CAN signal, will be presented by a generated I/O variable. These variables will be added to the Data Dictionary, too. To simplify the use of generated I/O variables in CarMaker/HIL, the file *IO_CAN_VarList.txt* lists all generated I/O variables (CAN signals, CAN message timings, length of CAN messages), together with the data type and full variable name. The C-module file *IO_CAN.c* contains the generated functions.

If not yet existing, CANiogen will also create a C-module called *IO_CAN_User*. This framework – represented by the files *IO_CAN_User.c* and *IO_CAN_User.h* – should be used to customize and refine the interface between model and CAN I/O. These files will never be overwritten by CANiogen.

The following Figure 16.1 shows, how the imported CAN database will fit into the CarMaker/HIL environment.

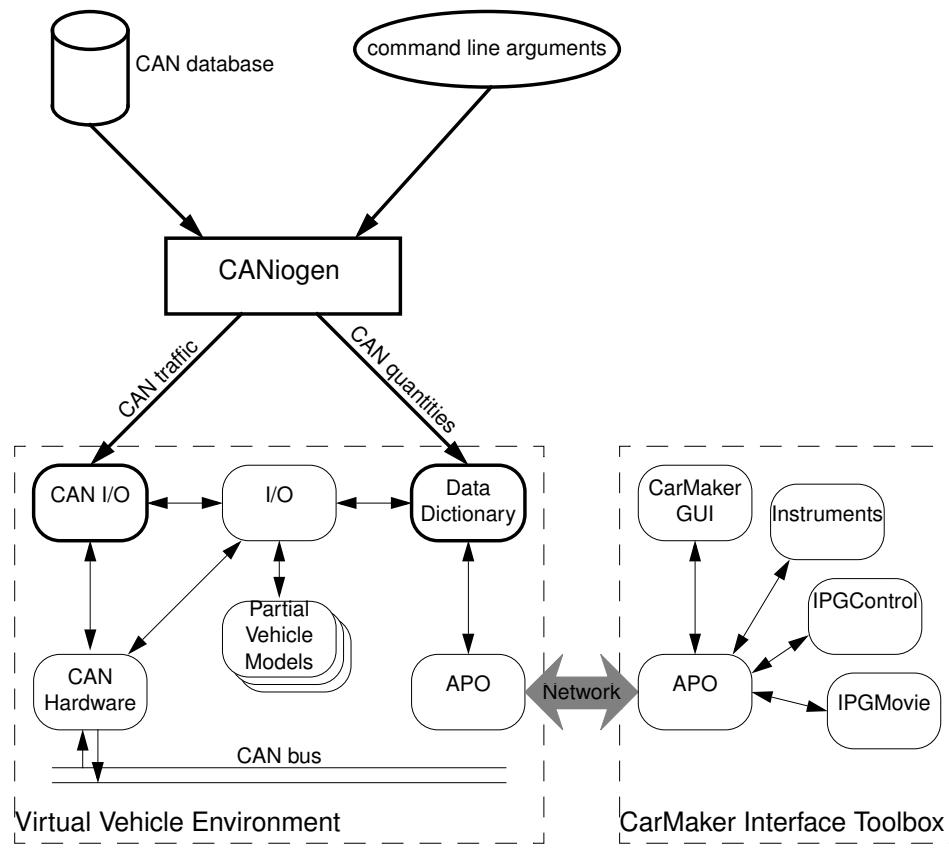


Figure 16.1: CANdb Import

J1939 Mode

When used for J1939 applications, CANiogen generates a description of the bus traffic in infofile format. The output is then used by the J1939 rest bus simulation in CarMaker/HIL. The main difference to the C-generator mode is, that changes to the CAN database do not require the CarMaker/HIL realtime application to be re-compiled.

The generated output file is called by default `J1939Parameters` and will be stored in the directory `Data/Config` within the CarMaker project. It carries the *FileIdent* "CarMaker-J1939 1" and lists the ECU configuration, PDU configuration and signal descriptions. With these definitions, the CarMaker J1939 rest bus simulation is able to handle the CAN traffic and generates automatically Data Dictionary entries for all J1939 signals.

The connection between CarMaker model variables and J1939 signals is to be done with the help of a `J1939Mappings` file, which is in infofile format, too. To ease the integration step, CANiogen also creates an example mapping file in the same directory as the `J1939Parameters` file, called `J1939Mappings.default`. This file is meant to be a template for customizations by the user. Useful informations can be extracted for use in a customized `J1939Mappings` file. It is not recommended to change the default mapping file, since it will be overwritten without prior notice.

The following Figure 16.1 shows, how the imported CAN database and the generated info-files will fit into the CarMaker/HIL environment.

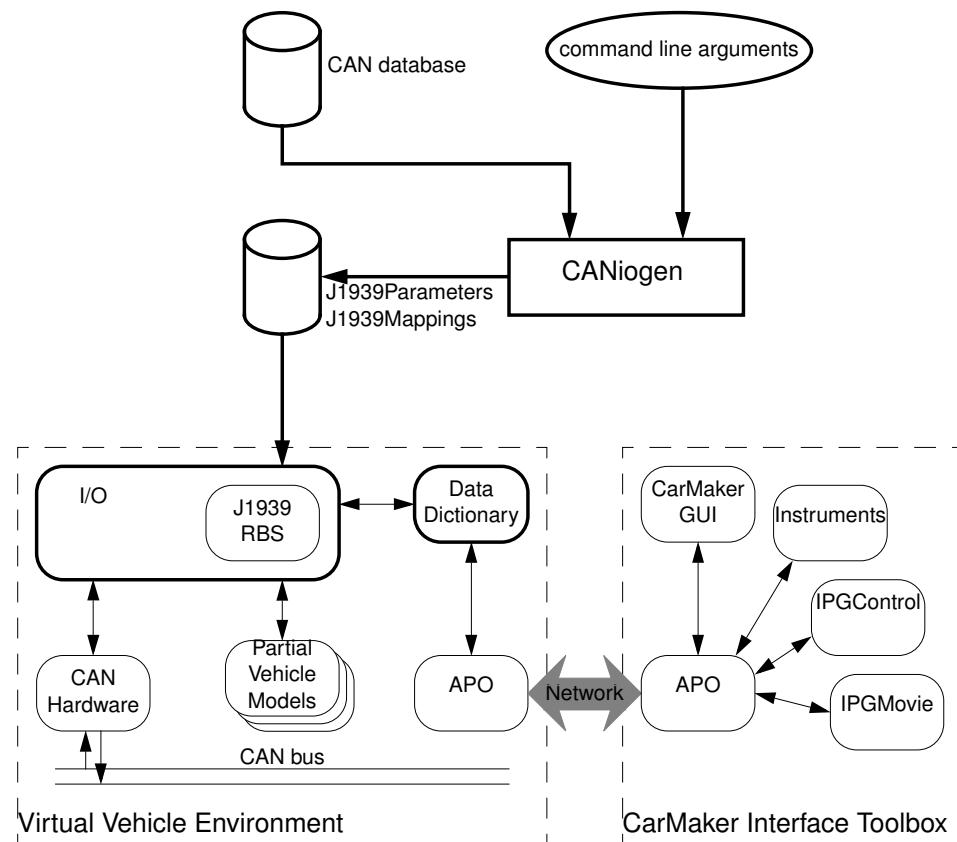


Figure 16.2: CANdb Import for J1939

16.1.2 Features

CANIogen was developed with the help of CANdb++ version 2.5 from Vector Informatik GmbH. Compatibility to CAN databases that were created with other versions of CANdb++, cannot be guaranteed. However, it is not likely to meet a compatibility problem that makes the import of a CAN database impossible.

A good example for this is the enhancement of the CAN protocol, called CAN FD (CAN with flexible data rate). This quite new technology causes some changes in the Vector CAN database. CANiogen is able to distinguish between a standard CAN database and a database with CAN FD messages (mixed databases are also possible) automatically.

A Vector CAN database defines ECUs (network nodes) and CAN messages, which are transferred inside a CAN network. Each CAN message has a unique message identifier and may contain up to 8 data bytes. The sender of a message codes CAN signals into the data bytes, to supply information to other ECUs. The CAN signals may vary in size, i.e. the data width need not to be a multiple of full bytes. Even worse, a CAN signal may be placed at any bit position within the data bytes and can be stored in Motorola or Intel byte order. As a result, it is difficult to decode and encode the CAN signals by hand when programming the CAN traffic for a CarMaker/HIL application, and bugs are usual. CANiogen helps you, by automating the following steps:

- manages transmission and reception of CAN messages for a given ECU, as specified in a CAN database.
- generates CAN I/O variables for any signal of all imported CAN messages and ECUs.

- generates Data Dictionary entries for all generated I/O variables (Data Dictionary entries can be omitted generally or for single I/O variables).
- imports additional CAN messages and signals of special interest.
- retrieves information about how and how often to send a CAN message (cyclic, spontaneous, ...) from the CAN database.
- fully supports extended CAN messages with 29-bit-identifiers.
- is able to deal with multiplexed CAN signals.
- can deal with J1939.
- fully supports Vector CAN FD databases and mixed databases with CAN & CAN FD messages.

Additionally, CANiogen can also deal with XML data bases according to the FIBEX standards version 2.0.1 and 3.1. The FIBEX standard (MCD-2 NET) is maintained and released by the ASAM group (s. <http://www.asam.net>). When using a FIBEX data base as input, CANiogen supports also data bases with multiple busses (clusters) – even of different type (CAN, FlexRay, MOST ...). However, CANiogen will only import CAN clusters, and will not combine multiple CAN clusters during import.

16.2 Using CANiogen

16.2.1 The CANiogen Command Line

When called without arguments or with the option `-h`, CANiogen prints a usage and exits. The usage lists the current version of CANiogen and gives an overview of accepted command line arguments. The following options are supported:

- `-h`:
Prints a usage and exits.
- `-H`:
Prints some more options with special features and exits.
- `-version`:
Prints version information and exits.
- `-outfmt format`:
Selects the output format. The format of the output files decides, whether CANiogen operates in the standard C-generator mode, or in the J1939 mode. Possible values are:
 - `cgen`: C-generator mode. This is the default.
 - `j1939`: J1939 mode.
- `-outdir dir`:
Defines the directory `dir` as destination directory for all generated output files. If the directory does not exist, it will be created first (see [section 16.3.1 'Output files'](#)).

Import and RBS options

- `-fibex`:
Import from FIBEX file. The last parameter (file name of CAN data base) must be a valid FIBEX data base.
- `-prefs prefs.tcl`:
Tells CANiogen to source the (Tcl-)file `prefs.tcl`. This file can be used e.g. to define a unit map table, which helps to rename units of CAN signals in CAN database to Car-Maker typical units when declaring Data Dictionary entries (see [section 16.2.6 'Customizing CANiogen'](#)).
- `-srvECU ECU1,ECU2,...`:
Gives a comma separated list of ECUs which are part of the test stand and should be served by the CarMaker/HIL application (see Serving ECUs (-srvECU option) in [section 16.2.2 'Importing Electronic Control Units \(ECU\)'](#)).
- `-simECU ECU1,ECU2,...`:
Gives a comma separated list of ECUs which are part of the rest bus simulation and whose CAN traffic should be simulated by the CarMaker/HIL application (see Simulating ECUs (-simECU option) in [section 16.2.2 'Importing Electronic Control Units \(ECU\)'](#)).
- `-excECU ECU1,ECU2,...`:
Gives a comma separated list of ECUs which should be ignored by CANiogen (see Excluding ECUs (-excECU option) in [section 16.2.2 'Importing Electronic Control Units \(ECU\)'](#)).
- `-rcvMsg Msg1,Msg2,...`:
Gives a list of CAN messages which should be additionally received (see [section 16.2.3 'Receiving CAN messages and signals of special interest'](#)).

- `-rcvSig Sig1,Sig2,...:`
Gives a list of CAN signals which should be additionally received (see [section 16.2.3 'Receiving CAN messages and signals of special interest'](#)).
- `-sndMsg Msg1,Msg2,...:`
Gives a list of CAN messages which should be additionally sent (see [section 16.2.4 'Sending arbitrary CAN messages'](#)).
- `-ignore Inf1,Inf2,...:`
Specifies which kind of information in the CAN database should be ignored by CANiogen (see [section 16.2.7 'Optimizing the output of CANiogen'](#)).

C-Code Generator options

- `-gateway:`
Enables the Gateway mode. In this mode, several real ECUs – specified with the option `-srvECU` – can be served at once, while each ECU is connected to the realtime system on a separate CAN bus. The generated C-code allows the CarMaker/HIL application to act as a gateway to the real ECUs of a CAN network, while all CAN traffic to these ECUs is totally controlled by CarMaker/HIL (see Acting as Gateway to real ECUs (-gateway option) in [section 16.2.2 'Importing Electronic Control Units \(ECU\)'](#)).
- `-suppress Sig1,Sig2,...:`
Gives a list of CAN signals / variables which should not be listed in the Data Dictionary (see [section 16.2.5 'Suppressing of I/O variables in the Data Dictionary'](#)).
- `-genAll:`
Tells CANiogen to generate as much C-code as possible (see [section 16.2.7 'Optimizing the output of CANiogen'](#)).
- `-noRngChk / -noRngChkRx / -noRngChkTx:`
Disables automatic range checking for signals when reading/writing the value from/to the data bytes of a CAN message (see [section 16.2.8 'Disabling range checking of Signal values'](#)).
- `-ddMsgId:`
Tells CANiogen to use the message id with data dictionary entries, instead of the message name. (see [section 16.2.10 'Naming of generated Data Dictionary entries'](#)).
- `-ioModule module_name:`
Prefix for generated C-files, C-functions and I/O-variables. Default value is "IO_CAN" (see [section 16.2.9 'Naming of generated files, I/O variables and functions'](#)).
- `-ioPrefix prefix:`
Prefix for generated I/O-variables, if required to be different from the prefix used for the C-files and C-functions (see [section 16.2.9 'Naming of generated files, I/O variables and functions'](#)).
- `-ddPrefix dd_prefix:`
Prefix for generated data dictionary entries, if required to be different from the prefix used for I/O-variables (see [section 16.2.10 'Naming of generated Data Dictionary entries'](#)).

J1939 options

- `-outfile filename:`
Tells CANiogen to write the generated infofile output to the file named `filename`. If the file already exist, it will be overwritten. The default file name is `J1939Paramters` (see [section 16.2.9 'Naming of generated files, I/O variables and functions'](#)).

FIBEX options

- `-canch ch_name:`

Specifies the name of the CAN channel, used for import. Only ECUs, which are connected to the CAN channel of name `ch_name`, will be scanned for import.

Extended features

Additionally to the options above, CANiogen accepts the following options:

- `-user login:`

Gives the login of the user who calls CANiogen. This might be necessary, if CANiogen fails to determine the login name automatically.

- `-host hostname:`

Gives the host name of the computer, CANiogen is running on. This might be necessary, if CANiogen fails to determine the host name automatically.

- `-quiet:`

Activates quiet mode, which tells CANiogen not to print any information, except warnings and errors.

- `-verbose:`

Disables quiet mode. CANiogen will print any information, also warnings and errors.

- `-debug level:`

Activates debugging output. Be careful with the use of this option. The higher the level, the more output is produced on the console. Be careful, it might be a multiple of the amount of generated source code.

- `-msgCTA att:`

Name of CAN message attribute, which holds the cycle time for sending a CAN message. The default attribute is GenMsgCycleTime (see [section 16.2.11 'Extended features'](#)).

- `-msgDTA att:`

Name of CAN message attribute, which holds the delay time when sending a CAN message. If specified, this attribute will be used to distribute the transmission of CAN messages in between their cycle times. A suitable attribute may be GenMsgDelayTime (see [section 16.2.11 'Extended features'](#)). But, in most case you will work best with the automatic distribution, which is the default.

- `-msgSTA att:`

Name of CAN message attribute, which holds the send type of a CAN message. The default attribute is GenMsgSendType (see [section 16.2.11 'Extended features'](#)).

- `-msgSCAv value:`

Value for CAN message attribute holding the send type of a CAN message, which marks a CAN message to be transmitted cyclically. The default value is cyclic (see [section 16.2.11 'Extended features'](#)).

- `-sigIVA att:`

Name of CAN signal attribute, which holds the default value of a CAN signal. The default attribute is GenSigInactiveValue (see [section 16.2.11 'Extended features'](#)).

- `-sigInvT text:`

Text to identify the member of a value table, which defines a raw value for a signal, indicating, that the signal value is invalid. The default text is invalid (see [section 16.2.11 'Extended features'](#)).

- `-sigErrT text:`

Text to identify the member of a value table, which defines a raw value for a signal indicating, that the signal is in error state. The default text is `error` (see section 16.2.11 'Extended features').

- `-sigUDefT` text:

Text to identify the member of a value table, which defines a raw value for a signal, indicating, that the signal is undefined. The default text is `undefined` (see section 16.2.11 'Extended features').

- `-sigUAvlT` text:

Text to identify the member of a value table, which defines a raw value for a signal, indicating, that the signal is unavailable. The default text is `unavailable` (see section 16.2.11 'Extended features').

- `-mkrules` objlist:

Generate Makefile rules for given list of `IO_CAN` object files and exit. If more than one object file is specified, `objlist` must be a list of objects, separated by commas or space characters and encapsulated in double quotes. The output is printed to standard out and can be included in Makefiles (see section 16.5.3 'Modifications to the Makefile').

- `-m410`:

When a standard CAN database (without CAN FD) is used, CANiogen takes the hardware functions of the M51 module for receiving and transmitting CAN messages, even if a M410 module is installed in CarMaker/HIL. This option will force CANiogen to set the hardware functions to the M410 module.

- `-eclipse`:

Enables CarMaker's eclipse project support. It's automatically taking care if you use the eclipse template providing by IPG.

How to use CANiogen

To import a vector CAN database, CANiogen requires the file name of a CANdb file (normally provided with the extension `.dbc`) and at least one of the options `-simECU`, `-srvECU`, `-rcvMsg`, `-sndMsg` or `-rcvSig`. If none of these options are given, CANiogen cannot import any data due to lack of information and exits with an error message:

```
> CANiogen CANdbdbc
CANiogen 1.2.2 -- (c) 2007 IPG GmbH, Karlsruhe, e-mail: info@ipg.de
I/O generator for vector CAN databases
```

ERROR: please specify at least:

- one ECU to handle receipt and transmission of CAN messages and/or
- one message to receive from CAN bus and/or
- one signal to receive from CAN bus and/or
- one message to send on CAN bus

The options `-simECU` and `-srvECU` cannot be used simultaneously.

A common CarMaker/HIL test stand realizes a complete virtual environment, whereas selective components of a virtual vehicle are realized in real hardware, such as an electronic control unit (ECU). The ECU itself – let us assume its name is *ESP* – is part of a CAN network, described in the CAN database (see Figure 16.3 for an example). It is the job of CarMaker/HIL, to simulate the behavior of a subset of this CAN network in such a manner, that the ECU believes to be located in a real vehicle. This subset will be defined through the options `-srvECU`, `-simECU`, `-rcvMsg`, `-sndMsg` and `-rcvSig`.

In the CAN network of Figure 16.3, the behavior of the ECUs *Engine* and *StWhlSensor* should be simulated by CarMaker/HIL, in order to supply the ECU *ESP* with the CAN messages *Engine_1*, *Engine_2* and *StWhlAng*. On the other hand, CarMaker/HIL should receive and evaluate the CAN message *ESP*, which is produced by the ECU *ESP*.

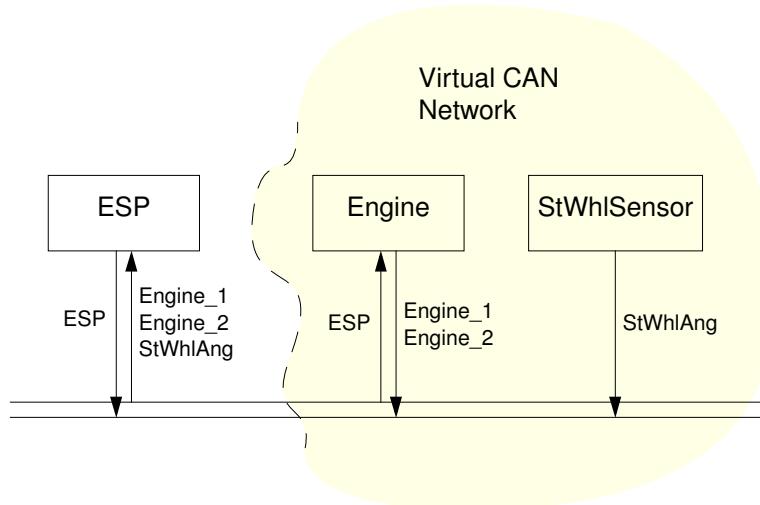


Figure 16.3: CAN network example

16.2.2 Importing Electronic Control Units (ECU)

CANIogen offers two ways to import ECUs as a whole from a CAN database: with the option `-simECU` and with the option `-srvECU`. These two options have oppositional effects, so they cannot be used simultaneously.

Simulating ECUs (`-simECU` option)

With the option `-simECU` you give a list of all ECUs in the CAN database, which send CAN messages to the real ECUs of the test stand, or receive CAN messages from them. CANiogen accepts a comma separated list of ECUs, as named in the CAN database. Unknown ECU names will result in an error message.

For this set of ECUs, CANiogen generates data structures and C code to manage all the CAN messages and signals, which are transferred to and from other ECUs in the same CAN network. Typically, the import list of ECUs includes all these ECUs of the CAN database, which are simulated by CarMaker/HIL, except the ones which are part of the test stand hardware.

With a CAN database, describing the CAN network of Figure 16.3, the command line might look something like:

```
> CANiogen -simECU Engine,StWhlSensor CANdbdbc
```

In this case, the virtual CAN network spans the whole network, except the ECU *ESP*, as *ESP* receives CAN messages from all the other ECUs. However, in other cases the virtual CAN network can be much smaller.

Serving ECUs (`-srvECU` option)

This option causes right the opposite to the option `-simECU`. Instead of specifying a list of ECUs to be simulated by CarMaker/HIL, you give a list of ECUs which are part of the real CAN network. The remaining part of the CAN network is to be considered as virtual.

CANIogen then generates C-code to supply these ECUs with CAN messages and CAN signals from the virtual part of the CAN network. On the other hand, the information which the real ECUs provide to the rest of the CAN network will be read back, too. Typically, the import list of ECUs names all the ECUs, which are part of the test stand. The rest of the CAN network will be taken to be simulated by CarMaker/HIL.

With a CAN database, describing the CAN network of [Figure 16.3](#), the command line might look something like:

```
> CANiogen -srvECU ESP CANdbdbc
```

In this case, the virtual CAN network spans the whole network, except the ECU *ESP*, as *ESP* receives CAN messages from all the other ECUs. If the CAN network tends to be quite big, and if there are only a few ECUs part of the test stand, you should prefer this option.

Excluding ECUs (-excECU option)

Of course, a CAN database might include even much more information than needed for the CAN network simulation of a specific test stand. There are often ECUs defined which can be ignored completely.

Its the option `-excECU`, that allows you to give a list of ECUs which should be ignored completely when generating the data structures and C code for CarMaker/HIL.

Acting as Gateway to real ECUs (-gateway option)

Some applications require CarMaker/HIL to control also the CAN traffic between ECUs that are part of the real CAN network. In this case, CarMaker/HIL has to act as a gateway to those ECUs which are the real part of the CAN network. Each of these ECUs has to be connected to the realtime system on its own CAN bus:

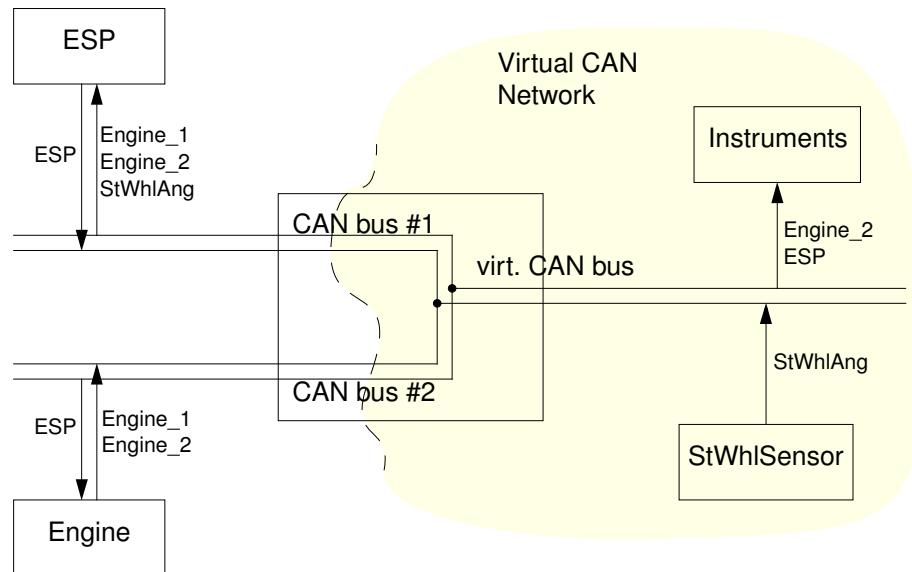


Figure 16.4: Rest bus simulation with CarMaker/HIL as gateway to real ECUs

With a constellation as shown in [Figure 16.4](#), CarMaker/HIL will receive the CAN message *ESP* from ECU *ESP* on CAN bus 1 and the CAN messages *Engine_1* and *Engine_2* from ECU *Engine* on CAN bus 2. In order to pretend the two ECUs *ESP* and *Engine* to be connected to each other directly, CarMaker/HIL will also forward the CAN message *ESP* to ECU *Engine* on CAN bus 2, as well as CAN messages *Engine_1* and *Engine_2* to ECU *ESP* on CAN bus 1.

Additionally, CarMaker/HIL will still send the CAN message *StWhlAng* to ECU *ESP*.

However, before distributing a CAN message to other real ECUs, the message and its data bytes can be manipulated, e.g. by changing values of bits or signals, by simulating bit errors, by delaying the transmission, by changing the message timings, or even by discarding the complete message.

To enable this function, use the `-gateway` switch together with the `-srvECU` option. In order to tell CANiogen to act as a gateway for the ECUs *ESP* and *Engine* as shown in Figure 16.4, the command line looks as follows:

```
> CANiogen -gateway -srvECU ESP,Engine CANdbdbc
```

The generated code will differ from the code generated without the `-gateway` option. Refer to section 16.3 'CANiogen's output (C-Code generator Mode)' and section 16.5 'Integration into CarMaker/HIL (C-Code Mode)' for details.

16.2.3 Receiving CAN messages and signals of special interest

In some cases you may be interested in additional information from ECUs, that are enclosed in the real part of the CAN network but do not interact with parts of the virtual CAN network. Normally, these ECUs would be ignored by CANiogen. The additional information, which you want to receive from the real part of the CAN network, can be specified through the options `-rcvMsg` and `-rcvSig`. When using these options, CANiogen also expects comma separated lists of CAN message names (or ids - in decimal or hexadecimal notation), or of CAN signal names, respectively.

Let us assume the steering wheel sensor *StWhlSensor* in Figure 16.3 to be the only simulated part of the CAN network. The other ECUs (*ESP* and *Engine*) will supply each other with all needed information. Only the CAN message *StWhlAng* has to be generated and transmitted by CarMaker/HIL. With the resulting command,

```
> CANiogen -simECU StWhlSensor CANdbdbc
```

only the data structures and C code to handle the transmission of CAN message *StWhlAng*, will be generated by CANiogen. No further CAN messages or signals will be handled by default. Nevertheless, you may be interested in receiving information like the wheel speeds and yaw rate from the ECU *ESP*, as well as the engine speed from the ECU *Engine*. Let us assume, that these information are provided by the CAN message *ESP*, and the CAN signal *E2_Data1* in message *Engine_2*. The command line would now look like:

```
> CANiogen -simECU StWhlSensor -rcvMsg ESP -rcvSig E2_Data1 CANdbdbc
```

16.2.4 Sending arbitrary CAN messages

Suppose, you have an ECU (*ECU1*) connected to your test stand, running a preliminary firmware version with limited operability. Another ECU (*ECU2*), which is also connected to the test stand, expects to receive a special CAN message (0x180) from *ECU1*. But, *ECU1* which should send this message, does not implement this function in its early version. So we have to simulate this CAN message by software, in order to complete the virtual environment. This is done with the option `-sndMsg`:

```
> CANiogen -srvECU ECU1,ECU2 -sndMsg 0x180 CANdbdbc
```

The option `-sndMsg` tells CANiogen to generate all necessary I/O variables, timing data structures and C-code to send CAN messages, even if they are not part of the virtual CAN network.

16.2.5 Suppressing of I/O variables in the Data Dictionary

Depending on the size of the CAN network and the number of simulated ECUs, there may be generated a huge amount of I/O variables. By default, all these I/O variables will be listed in the Data Dictionary, which may blow it up unnecessarily. As many of them will hardly interest the user, you are able to suppress the listing of any generated I/O variable in the Data Dictionary with the option `-suppress`.

With the option `-suppress`, CANiogen accepts a comma separated list of patterns. A pattern can be just the name of a CAN signal, the name of a CAN message or a combination of signal name and message name. Even the wildcard `*` is allowed. CANiogen accepts the following kinds of patterns:

- `-suppress S_Data1:`
The listing of all CAN signals, named `S_Data1` in any CAN message will be suppressed.
- `-suppress S_Data*:`
The listing of all CAN signals that match the pattern `S_Data*` in any CAN message will be suppressed.
- `-suppress S_Data1@StWhlAng:`
The listing of the CAN signal named `S_Data1` in the CAN message `StWhlAng` will be suppressed.
- `-suppress S_Data*@StWhlAng:`
The listing of all CAN signals that match the pattern `S_Data*` in the CAN message `StWhlAng` will be suppressed.
- `-suppress S_Data*@StWhl*:`
The listing of all CAN signals that match the pattern `S_Data*` in all CAN messages which match the pattern `StWhl*` will be suppressed.
- `-suppress *@StWhl* or -suppress @StWhl*:`
The listing of all CAN signals, timing variables and data length counters in all CAN messages that match the pattern `StWhl*` will be suppressed.

If we look again at [Figure 16.3](#), supposing that we want to suppress the listing of the signals `E_Data1` in the message `ESP` and all signals that begin with `E1_Data`, then the command line will look like:

```
> CANiogen -srvECU ESP -suppress E_Data1@ESP,E1_Data* CANdb.dbc
```



If you call CANiogen inside of a Unix-like command shell, you will have to quote the wildcards like `*`, for instance by prepending a backslash `\`. Otherwise, the command shell may perform a command line substitution (as performed on a command like “`copy *.dbc /tmp`”).



Another – more generally, but less flexible – way to suppress the listing of generated variables in the Data Dictionary does not need any command line option, but can be done on C-level: to control whether all signal quantities and all timing quantities should be listed in the Data Dictionary, CANiogen generates two global variables `IO_CAN.DeclareQuants` and `IO_CANTimings.DeclareQuants`. By default, these variables are initialized with 1 (declare quantities). However, if you wish e.g. not to list any timing variables in the Data Dictionary, just place the line

```
IO_CANTimings.DeclareQuants = 0;
```

in the function `IO_CAN_User_Init_First()`. The advantage of this method is: you can easily enable or disable the listing without the need to generate the code again.

16.2.6 Customizing CANiogen

Right after initialization, CANiogen searches for a file named `.CANiogen.tcl` in the current directory. If the file exists, it will be loaded and processed by the built-in Tcl interpreter. The contents of the file should be plain Tcl code.

Another way to load a preferences file is to use the command line option `-prefs`:

```
CANiogen -prefs CANiogenPrefs.tcl -srvECU ESP CANdbdbc
```

This option tells CANiogen to source the Tcl file `CANiogenPrefs.tcl`. The format of this file is the same as with `.CANiogen.tcl`.

Mapping units of CAN signal to CarMaker units (SI units)

When working with CAN databases and CarMaker, it might happen that IPGControl does not offer to convert and display a CAN signal in another unit, like `km/h` in `m/s`, or similar. This happens, if the spelling of the unit in the CAN database does not exactly match one of the common SI units, which are well known in CarMaker.

However, you do not need to give up. CANiogen provides a feature to map the name of a unit of a CAN signal to a common name, which will be recognized by CarMaker.

Supposed to be, the CAN database uses the unit “miles per hour” for CAN signals expressing velocities. Of course, this is not a SI unit and if the Data Dictionary entry is created for such a CAN signal, IPGControl will not be able to display the quantity in `km/h` nor group it with other velocity quantities. So, it would be nice to declare the quantities with the unit name `mph` instead.

To enable the mapping of units, create a file named `.CANiogen.tcl` in the current directory with the following contents:

Listing 16.1: Preferences file for CANiogen

```
1: # Preferences file for CANiogen
2:
3: #
4: # Unit Mapping Table
5: #
6: array set Unit_Map {
7:   "miles per hour"      mph
8:   "kilometers per hour" km/h
9:   meters                m
10:  kilometers            km
11:  meters/sec            m/s
12: }
```

As you can see in [Listing 16.1](#), there is defined a Tcl array `Unit_Map` with the `array set` command, listing pairs of replacement directives. They tell CANiogen to replace the unit of any signal that matches a key on the left side with the corresponding string on the right side.



Make sure that any key or replacement does not contain unquoted white spaces. Otherwise, the results may be unpredictable.

The unit mapping mechanism is just a textual replacement. There is no conversion of signal values.

16.2.7 Optimizing the output of CANiogen

CANiogen will always try to minimize the amount of generated C-code. However, in some cases you might be interested in additional macros, I/O variables or CAN messages, or you want to ignore obsolete information of the CAN database. The following options allow you to control this:

- `-ignore unknown:`

This option tells CANiogen to ignore CAN messages and CAN signals to and from the ECU `Vector_XXX`. This is the default for messages and signals, that do not have a sender or receiver

- `-ignore nCyclic:`

Use this option to exclude non cyclic CAN messages from the `IO_CAN_Send()` function. CANiogen generates the data structures for all Tx messages of an ECU, cyclic and non cyclic CAN messages. The `-ignore nCyclic` option tells CANiogen, that only these messages, which are recognized as cyclic, shall be included in the `IO_CAN_Send()` function. Otherwise, even non cyclic Tx messages will be listed, but you can still control the transmission with the timing parameters.

- `-genAll:`

To decode and encode signals in a CAN message, CANiogen generates `*_SetV()` and `*_GetV()` macros for each signal. Of course, many of these macros are unnecessary, because not every signal is received as well as transmitted. CANiogen optimizes the output by generating only those macros, that are really necessary. If you use the option `-genAll`, all of the `*_SetV()` and `*_GetV()` macros are generated, whether needed or not. This may drastically increase the amount of generated code.

16.2.8 Disabling range checking of Signal values

According to the attributes of a signal, which are defined in the CAN database file, CANiogen generates the `*_SetV()` and `*_GetV()` macros to limit the value of the signal to fit into the range of its minimum and maximum. This limiting is performed always when encoding the value of a signal into a CAN message, as well as when decoding it out of a CAN message. As a consequence, the value of a signal can never be out of range.

However, if you are testing the functionality and reliability of an ECU, you might be interested in disabling the range checking. This is done with the options `-noRngChk`, `-noRngChkRx` and `-noRngChkTx`.

CANiogen will then not include calls to the macros `IO_CAN_FixMin()` and `IO_CAN_FixMax()` in the `*_SetV()` and `*_GetV()` macros. However, the macros `IO_CAN_Range()`, `IO_CAN_FixMin()` and `IO_CAN_FixMax()` are generated anyway and can be used by hand.

16.2.9 Naming of generated files, I/O variables and functions

C-Code Generator Mode

In C-Code generator mode, CANiogen normally names the generated files `IO_CAN*` and uses `IO_CAN` as prefix for all generated variables or functions. The output directory is always the present working directory from which CANiogen is called. CANiogen is intended to be called in the source directory of your CarMaker/HIL project.

If you want to use more than one CAN database simultaneously, you can initiate CANiogen to use different prefixes in order to avoid conflicts during the compilation and linking process. This allows you to use different CAN databases for different I/O configurations.

The prefix can be changed with the option `-ioModule`:

```
> CANiogen -srvECU ESP -ioModule IO_ESP CANdbdbc
```

This prompts CANiogen to name the output files `IO_ESP.*` and to use the prefix `IO_ESP` for all generated I/O variables and functions.

Alternatively (or additionally), you can use the option `-ioPrefix`. CANiogen will then use another prefix for generated I/O variables, which may differ from the file name. This prefix will only affect the name of I/O variables, but not the name of data types and functions. For instance, if you call CANiogen with `-ioPrefix IO_ESP` but do not use the option `-ioModule`, the output files will still be named `IO_CAN*`, as well as the data types and functions will still have the prefix `IO_CAN` (e.g. `tIO_CANVec`, `IO_CAN_Send()`), but all generated variables will have the prefix `IO_ESP` (e.g. `tIO_CANVec IO_ESP`).



Be careful when using more than one CAN database. Generated files will be overwritten without a prompt. It is recommended to use the option `-ioModule` with different module names for each imported CAN database. The option `-ioPrefix` then is obsolete.

J1939 Mode

In J1939 mode, the generated infofile is named by default `J1939Parameters` and will be saved in the directory `Data/Config`. The name and the directory of the output file can be customized with the `-outfile` and the `-outfile` option. Additionally, a second default mapping file will be generated in the same directory. As long as the name of the output file contains the sub string `Parameters`, the name of the mapping file will be generated by replacing the sub string `Parameters` with `Mappings` and adding the suffix `.default` (e.g. `J1939Mappings.default`). Otherwise, the mappings file will be the name of the parameters file, plus the suffix `.Map_defaults` (e.g. `J1939_ESP.Map_defaults`).



Be careful when using more than one CAN database. Generated files will be overwritten without a prompt. For each imported CAN database, the option `-outfile` has to be used to assign distinct names to the ouput files.

16.2.10 Naming of generated Data Dictionary entries

For all generated data dictionary entries, CANiogen uses the following naming scheme:

`IO_CAN.CAN_Msg_Name.Signal_Name`

However, this might result in very long names, making it uncomfortable for the user when attempting to find a variable in IPGControl or other CarMaker tools.

CANiogen offers two command line options, `-ddMsgId` and `-ddPrefix`, to control the naming scheme.

The option `-ddMsgId` tells CANiogen to use the message id in hex, instead of the message name. For instance, the following command line

> CANiogen -ddMsgId CANdbdbc

will result in data dictionary entries like

`IO_CAN.101h.Signal_Name`

for the CAN message with identifier 0x101 (257 decimal).

In case the prefix for I/O variables is too long, the command line option `-ddPrefix` prefix will be a good help:

> CANiogen -ddPrefix CAN CANdbdbc

will tell CANiogen to use the prefix *CAN*, instead of *IO_CAN*, which is the default. All data dictionary entries will be named matching the following scheme:

```
CAN.CAN_Msg_Name.Signal_Name
```

Thus, using CANiogen with both options – `-ddMsgId` and `-ddPrefix` – as follows,

```
> CANiogen -ddMsgId -ddPrefix CAN CANdbdbc
```

will produce the shortest data dictionary entries at all:

```
CAN.101h.Signal_Name
```

Changing format string for message id

When using the option `-ddMsgId`, you might be interested in a customized format of the message id in the name of a data dictionary entry. By default, CANiogen uses the format string "%03xh" and the tcl command `format`, which produces a naming like shown above.

However, you can replace the format string with the help of the preferences file `.CANiogen.tcl` – or with the command line option `-prefs CANiogenPrefs.tcl` – by changing the variable `Imp(ddMsgIdFmt)`:

```
set Imp(ddMsgIdFmt) "%d"
```

The command above will result in a decimal coding of the message id. The data dictionary entry of the example above will be named:

```
CAN.257.Signal_Name
```

16.2.11 Extended features

Message and Signal attributes

Some information which CANiogen needs to transmit CAN messages and to initialize variables are encapsulated in attributes of messages and signals. As a default, CANiogen evaluates the following attributes:

- `GenMsgCycleTime` (**message attribute**):

This attribute is checked to determine the cycle time for the transmission of a CAN message. The value is expected to be an integer, which gives the cycle time in milliseconds. If this attribute is not found in the CAN database, the send period of a CAN message will be set to -1. You will then have to set the timing parameters of the CAN message by hand in order to activate the transmission (see [section 16.3.4 'The C module IO_CAN.c'](#) and [section 16.5 'Integration into CarMaker/HIL \(C-Code Mode\)'](#)).

If the cycle time is given by another attribute, you can tell CANiogen to use this attribute instead, with the option `-msgCTA`:

```
> CANiogen -srvECU ESP -msgCTA MsgCycleTimeAtt CANdbdbc
```

- `GenMsgDelayTime` (**message attribute**):

This attribute is used to distribute the transmission of different CAN messages within their cycle time. This helps to reduce the load of the CPU and the CAN bus. The transmission of a CAN message is triggered by the following condition:

```
if (GenMsgCycleTime > 0 && CycleNo%GenMsgCycleTime == GenMsgDelayTime)
then send CAN message
else do nothing
```

This means, a CAN message will only be sent if `GenMsgDelayTime` is in the valid range of $0 \leq \text{GenMsgDelayTime} < \text{GenMsgCycleTime}$. You might have to adjust the timing parameter of a message by hand, if `GenMsgDelayTime` does not have a valid value (see [section 16.3.4 'The C module IO_CAN.c'](#) and [section 16.5 'Integration into CarMaker/HIL \(C-Code Mode\)'](#)).

If the distribution is to be described better by another attribute, you can tell CANiogen to use this attribute instead, with the option `-msgDTA`:

```
> CANiogen -srvECU ESP -msgDTA MsgDelayTimeAtt CANdbdbc
```

- `GenMsgSendType` (**message attribute**):

This attribute is used to recognize, if a CAN message is to be transmitted periodically or not. As a default, CANiogen compares the attribute `AttMsgSndType` with the string “cyclic”. The comparison is done case insensitive, which means that if the attribute `AttMsgSndType` does not include the string “cyclic” or any possible spelling of it with capital letters, the CAN message will not be recognized as a cyclic message. But, as long as the option `-ignore nCyclic` is not used the CAN message will be imported anyway. Only the timing parameters will be initialized with invalid values (-1) to prevent the automatic transmission. You can still control the transmission of these messages by adjusting the timing parameters (see [section 16.3.4 'The C module IO_CAN.c'](#) and [section 16.5 'Integration into CarMaker/HIL \(C-Code Mode\)'](#)).

If the send type is to be described better by another attribute, or if cyclic messages are indicated by another value than “cyclic”, you can adjust this with the options `-msgSTA` and `-msgSCAv`:

```
> CANiogen -srvECU ESP -msgSTA MsgSendTypeAtt -msgSCAv AttValCyclic CANdbdbc
```

- `GenSigInactiveValue` (**signal attribute**):

This attribute is used to initialize CAN signals with suitable values, especially those signals, which are not provided with simulation results. If this attribute is not defined for a signal, its initial value will be set to 0, which might be invalid. To set the initial value by hand, please refer to [section 16.3.4 'The C module IO_CAN.c'](#) and [section 16.5 'Integration into CarMaker/HIL \(C-Code Mode\)'](#).

If the initial value of CAN signals is to be described better by another attribute, you can adjust this with the option `-sigIVA`:

```
> CANiogen -srvECU ESP -sigIVA SigStartValAtt CANdbdbc
```

Invalid, error, undefined and unavailable values of Signals

It is quite common to define special raw values of CAN signals, which represent a special state, rather than a physical value, e.g. to mark a signal as invalid, erroneous, undefined or unavailable. This can be defined by a separate state signal, or by the signal itself.

If the CANdb defines a value table for a CAN signal (e.g. signal has enumeration type), CANiogen supports and generates invalid-, error-, undefined- and unavailable-states for the signal.

CANiogen just needs to know which text is used to describe the different states. By default, CANiogen searches for the strings “invalid”, “error”, “undefined” and “unavailable” in value tables to find the raw values of a signal, which qualifies it as invalid, erroneous, undefined or unavailable. However, you can customize this using the options `-sigInvT`, `-sigErrT`, `-sigUDefT` and `-sigUAvlT`:

```
> CANiogen -srvECU ESP -sigInvT "signal not valid" -sigErrT "signal has error" \
-sigUDefT "signal is undefined" -sigUAvlT "signal not available" CANdbdbc
```

16.3 CANiogen's output (C-Code generator Mode)

16.3.1 Output files

CANiogen generates the C modules `IO_CAN.c` and `IO_CAN_User.c`, the corresponding header files `IO_CAN.h` and `IO_CAN_User.h`, as well as a complete list with all generated I/O variables in the file `IO_CAN_VarList.txt` (how to change the naming of the files is described in [section 16.2.9 'Naming of generated files, I/O variables and functions'](#)). All the files are generated in the current working directory, from which CANiogen is executed (unless the output directory is overridden with the option `-outdir`).

The output files `IO_CAN.c` and `IO_CAN.h` will be overwritten without a prompt. You should not modify them.

Customization can be done with the files `IO_CAN_User.c` and `IO_CAN_User.h`. These files are created by CANiogen as a framework for the user, to connect the CarMaker model environment with the virtual CAN environment. Once created, they will never be overwritten.

16.3.2 The header file IO_CAN.h

The header file `IO_CAN.h` lists the data structures of the generated I/O variables and defines several macros and functions for data access.

I/O variables

The I/O variables are arranged hierarchically within data structures, which reflect the relationship in the CAN database. Within the data structure `IO_CAN` at top level, an ECU is presented as a C data structure, which itself contains CAN messages, transmitted by the ECU. A CAN message is also defined as a C data structure, which lists its CAN signals according to their data type:

Listing 16.2: I/O variables generated by CANiogen

```

1:  /* Input Vector for CAN communication */
2:  typedef struct {
3:      int DeclareQuants;
4:      int DeclareQuantsRaw;
5:      int DeclareQuantsState;
6:
7:      /* ECU Engine */
8:      struct tIO_CAN_Engine {
9:          /* CAN Msg 0x201 (Engine_1) */
10:         int Engine_1_DLC;
11:         struct tIO_CAN_Engine_Engine_1 {
12:             unsigned short     E1_Data1;
13:             unsigned short     E1_Data1_Raw;
14:             char               E1_Data1_State;
15:             unsigned short     E1_Data2;
16:             unsigned short     E1_Data2_Raw;
17:             char               E1_Data2_State;
18:             float              E1_Data3;
19:             unsigned char     E1_Data3_Raw;
20:             char               E1_Data3_State;
21:             unsigned char     E1_Info;
22:             unsigned char     E1_Info_Raw;
23:             char               E1_Info_State;
24:             unsigned char     E1_MPlex;
25:             unsigned char     E1_MPlex_Raw;
26:             char               E1_MPlex_State;
27:         } Engine_1;
28:     }

```

Listing 16.2: I/O variables generated by CANiogen

```

29:     /* CAN Msg 0x202 (Engine_2) */
30:     int Engine_2_DLC;
31:     struct tIO_CAN_Engine_Engine_2 {
32:         unsigned int      E2_Data1;
33:         unsigned int      E2_Data1_Raw;
34:         char              E2_Data1_State;
35:         unsigned char    E2_Data2;
36:         unsigned char    E2_Data2_Raw;
37:         char              E2_Data2_State;
38:     } Engine_2;
39: } Engine;
40:
41: /* ECU ESP */
42: struct tIO_CAN_ESP {
43:     /* CAN Msg 0x101 (ESP) */
44:     int ESP_DLC;
45:     struct tIO_CAN_ESP_ESP {
46:         unsigned short   E_Data1;
47:         unsigned short   E_Data1_Raw;
48:         char              E_Data1_State;
49:         unsigned int      E_Data2;
50:         int               E_Data2_Raw;
51:         char              E_Data2_State;
52:         float             E_Data3;
53:         short             E_Data3_Raw;
54:         char              E_Data3_State;
55:     } ESP;
56: } ESP;
57:
58: /* ECU StWhlSensor */
59: struct tIO_CAN_StWhlSensor {
60:     /* CAN Msg 0x001 (StWhlAng) */
61:     int StWhlAng_DLC;
62:     struct tIO_CAN_StWhlSensor_StWhlAng {
63:         unsigned char    S_Counter;
64:         unsigned char    S_Counter_Raw;
65:         char              S_Counter_State;
66:         float             S_Data1;
67:         short             S_Data1_Raw;
68:         char              S_Data1_State;
69:         unsigned char    S_Data1Inv;
70:         unsigned char    S_Data1Inv_Raw;
71:         char              S_Data1Inv_State;
72:         unsigned short   S_Data2;
73:         short             S_Data2_Raw;
74:         char              S_Data2_State;
75:     } StWhlAng;
76: } StWhlSensor;
77: } tIO_CANVec;

```

When used with the `-gateway` option, CANiogen includes also some information of the CAN interfaces for the real ECUs, which are connected on separate CAN busses to the realtime system. The data structures of the gatewayed ECUs include an additional variable `CAN_If` of type `tIO_CANChCfg`, which has the members `Slot` and `Channel` to associate it with a CAN interface. For each CAN message, there will be also generated an additional variable `MsgName_RxCnt`. This variable is required to recognize, if a received CAN message needs to be forwarded to other ECUs:

Listing 16.3: I/O variables generated by CANiogen in Gateway mode

```
1:  /* Interface configuration for ECUs */
2:  typedef struct {
3:      int Slot;
4:      int Channel;
5:  } tIO_CANCHCfg;
6:
7:  typedef enum {
8:      IO_CAN_ECU_Engine      = 1,
9:      IO_CAN_ECU_ESP         = 2,
10:     IO_CAN_ECU_StWhlSensor = 3
11: } tIO_CAN_ECU;
12:
13:
14: /* Signal states */
15: enum {
16:     IO_CAN_SState_None = 0,
17:     IO_CAN_SState_Valid,
18:     IO_CAN_SState_Invalid,
19:     IO_CAN_SState_Unavailable,
20:     IO_CAN_SState_Undefined,
21:     IO_CAN_SState_Error,
22:     IO_CAN_SState_nStates
23: };
24:
25: extern int IO_CAN_SigStates[IO_CAN_SState_nStates];
26:
27:
28: /* Input Vector for CAN communication */
29: typedef struct {
30:     int DeclareQuants;
31:     int DeclareQuantsRaw;
32:     int DeclareQuantsState;
33:
34:     /* ECU Engine */
35:     struct tIO_CAN_Engine {
36:         tIO_CANCHCfg CAN_If;
37:
38:         /* CAN Msg 0x201 (Engine_1) */
39:         int Engine_1_DLC;
40:         int Engine_1_RxCnt;
41:         struct tIO_CAN_Engine_Engine_1 {
42:             unsigned short    E1_Data1;
43:             unsigned short    E1_Data1_Raw;
44:             char              E1_Data1_State;
45:             unsigned short    E1_Data2;
46:             unsigned short    E1_Data2_Raw;
47:             char              E1_Data2_State;
48:             float             E1_Data3;
49:             unsigned char    E1_Data3_Raw;
50:             char              E1_Data3_State;
51:             unsigned char    E1_Info;
52:             unsigned char    E1_Info_Raw;
53:             char              E1_Info_State;
54:             unsigned char    E1_MPlex;
55:             unsigned char    E1_MPlex_Raw;
56:             char              E1_MPlex_State;
57:         } Engine_1;
58:     }
```

Listing 16.3: I/O variables generated by CANiogen in Gateway mode

```
59:     /* CAN Msg 0x202 (Engine_2) */
60:     int Engine_2_DLC;
61:     int Engine_2_RxCnt;
62:     struct tIO_CAN_Engine_Engine_2 {
63:         unsigned int      E2_Data1;
64:         unsigned int      E2_Data1_Raw;
65:         char              E2_Data1_State;
66:         unsigned char    E2_Data2;
67:         unsigned char    E2_Data2_Raw;
68:         char              E2_Data2_State;
69:     } Engine_2;
70: } Engine;
71:
72: /* ECU ESP */
73: struct tIO_CAN_ESP {
74:     tIO_CANhCfg CAN_If;
75:
76:     /* CAN Msg 0x101 (ESP) */
77:     int ESP_DLC;
78:     int ESP_RxCnt;
79:     struct tIO_CAN_ESP_ESP {
80:         unsigned short   E_Data1;
81:         unsigned short   E_Data1_Raw;
82:         char              E_Data1_State;
83:         unsigned int     E_Data2;
84:         int               E_Data2_Raw;
85:         char              E_Data2_State;
86:         float             E_Data3;
87:         short             E_Data3_Raw;
88:         char              E_Data3_State;
89:     } ESP;
90: } ESP;
91:
92: /* ECU StWhlSensor */
93: struct tIO_CAN_StWhlSensor {
94:     /* CAN Msg 0x001 (StWhlAng) */
95:     int StWhlAng_DLC;
96:     int StWhlAng_RxCnt;
97:     struct tIO_CAN_StWhlSensor_StWhlAng {
98:         unsigned char    S_Counter;
99:         unsigned char    S_Counter_Raw;
100:        char              S_Counter_State;
101:        float             S_Data1;
102:        short             S_Data1_Raw;
103:        char              S_Data1_State;
104:        unsigned char    S_Data1Inv;
105:        unsigned char    S_Data1Inv_Raw;
106:        char              S_Data1Inv_State;
107:        unsigned short   S_Data2;
108:        short             S_Data2_Raw;
109:        char              S_Data2_State;
110:    } StWhlAng;
111: } StWhlSensor;
112: } tIO_CANVec;
```

Timing parameters

For the CAN messages, which will be transmitted by CarMaker/HIL, a data structure with timing information will be generated:

Listing 16.4: CAN message timings

```
1:  /* Timing parameters for Tx CAN messages */
2:  typedef struct {
3:      int SendPeriod;
4:      int SendDistrib;
5:  } tIO_CANMsgTiming;
6:
7:  typedef struct {
8:      int DeclareQuants;
9:
10:     /* ECU Engine */
11:     struct IO_CANTimings_Engine {
12:         tIO_CANMsgTiming Engine_1;
13:         tIO_CANMsgTiming Engine_2;
14:     } Engine;
15:
16:     /* ECU StWhlSensor */
17:     struct IO_CANTimings_StWhlSensor {
18:         tIO_CANMsgTiming StWhlAng;
19:     } StWhlSensor;
20: } tIO_CANTimings;
21:
22: extern tIO_CANVec    IO_CAN;
23: extern tIO_CANTimings IO_CAN_Timings;
```

In Gateway mode, each received CAN message will be forwarded to the other real ECUs without any further delay within the same simulation cycle. However, some applications might also want to control and manipulate the timings for the gatewayed CAN messages. In order to handle this and to provide the possibility to switch between the two modes, an additional variable `*_GwNoDelay` will be generated for each gatewayed CAN message:

Listing 16.5: CAN message timings in Gateway mode

```

1:  /* Timing parameters for Tx CAN messages */
2:  typedef struct {
3:      int SendPeriod;
4:      int SendDistrib;
5:  } tIO_CANMsgTiming;
6:
7:  typedef struct {
8:      int DeclareQuants;
9:
10:     /* ECU Engine */
11:     struct IO_CANTimings_Engine {
12:         int             Engine_1_GwNoDelay;
13:         tIO_CANMsgTiming Engine_1;
14:         int             Engine_2_GwNoDelay;
15:         tIO_CANMsgTiming Engine_2;
16:     } Engine;
17:
18:     /* ECU ESP */
19:     struct IO_CANTimings_ESP {
20:         int             ESP_GwNoDelay;
21:         tIO_CANMsgTiming ESP;
22:     } ESP;
23:
24:     /* ECU StWhlSensor */
25:     struct IO_CANTimings_StWhlSensor {
26:         int             StWhlAng_GwNoDelay;
27:         tIO_CANMsgTiming StWhlAng;
28:     } StWhlSensor;
29: } tIO_CANTimings;

```

The timing information is fetched out of the CAN database through the message attributes `GenMsgCycleTime` and `GenMsgDelayTime`. If everything works fine, these attributes are defined in the database. So, the value of the attribute `GenMsgCycleTime` will be assigned to the timing variable `SendPeriod`. This attribute defines a cycle time in milliseconds for cyclically transmitted CAN messages.

The parameter `SendDistrib` is used to distribute all (Tx-) CAN messages with the same cycle time within their time slice. Normally, this distribution is done automatically. Optionally, CANiogen can be requested to use a message attribute like `GenMsgDelayTime` to get the initial value for this parameter out of the CAN database.

If CANiogen distributes the (Tx-) CAN messages automatically, it tries to keep the number of transmitted CAN messages per simulation step as low as possible. However, in order to keep reproducibility, the parameter `SendDistrib` is not determined randomly, but based on prime numbers. Among all messages with the same `SendPeriod`, CANiogen starts with the largest prime number between 0 and `SendPeriod`. The next message receives the next lower prime number and so on. After all prime numbers have been used up, CANiogen continues with odd numbers (largest number first) and finally uses up even numbers. The idea of this distribution is to prevent an overflow of the CAN Tx Queue – not because of too much CAN traffic, but because of an inefficient use of resources.

Function prototypes and macro definitions

For integration into CarMaker/HIL and for access to the generated I/O variables, CANiogen generates a set of function prototypes and macros:

Listing 16.6: Function prototypes

```

1:  /* Function definitions */
2:  int IO_CAN_Init_First (void);
3:  int IO_CAN_Init      (void);
4:  int IO_CAN_Recv      (struct CAN_Msg *Msg, const unsigned CycleNo);
5:  void IO_CAN_RecvLoop (int Slot, int Channel, const unsigned CycleNo);
6:  int IO_CAN_Send      (int Slot, int Channel, const unsigned CycleNo);
7:  int IO_CAN_ReadyToSend(int MsgId, const unsigned CycleNo);
8:  int IO_CAN_SendMsg   (int Slot, int Channel, const unsigned CycleNo, int MsgId);
9:
10:
11: extern int (*IO_CAN_CAN_Recv) (int Slot, int Channel, struct CAN_Msg *Msg);
12: extern int (*IO_CAN_CAN_Send) (int Slot, int Channel, struct CAN_Msg *Msg);
13:
14:
15: /*
16:  ** Hook functions for Rx- and Tx messages:
17:  ** - IO_CAN_RxHook()
18:  ** + will be called for received CAN messages:
19:  **     a) before decoding any signals (IO_CAN_PRE_DECODE)
20:  **         -> allows to manipulate the raw data bytes of the received message
21:  **         -> if return value is <0, the whole message will be discarded
22:  **     b) after decoding signals    (IO_CAN_POST_DECODE)
23:  **         -> allows to react on received CAN message
24:  **         -> return value does not have any effect
25:  ** - IO_CAN_TxHook()
26:  ** + will be called when sending a CAN message:
27:  **     a) before encoding any signals (IO_CAN_PRE_ENCODE)
28:  **         -> allows to do some preparations before specific CAN message is sent
29:  **         -> return value does not have any effect
30:  **     b) after encoding all signals (IO_CAN_POST_ENCODE)
31:  **         -> allows to manipulate the raw data bytes of the Tx message
32:  **         -> if return value is <0, the message will not be sent
33:  */
34: #define IO_CAN_PRE_DECODE 0
35: #define IO_CAN_PRE_ENCODE 0
36: #define IO_CAN_POST_DECODE 1
37: #define IO_CAN_POST_ENCODE 1
38:
39: extern int (*IO_CAN_RxHook) (struct CAN_Msg *Msg, const unsigned CycleNo, int Part);
40: extern int (*IO_CAN_TxHook) (struct CAN_Msg *Msg, const unsigned CycleNo, int Part);

```

However, in Gateway mode, the set of generated functions differs from those listed in Listing 16.6. The functions `IO_CAN_RecvLoop()`, `IO_CAN_Send()` and `IO_CAN_SendMsg()` are replaced by the functions `IO_CAN_RecvLoopGw()`, `IO_CAN_SendGw()` and `IO_CAN_SendMsgGw()`, which do not require `Slot` and `Channel` as parameters. In Gateway mode, the association between Tx CAN message and the appropriate CAN channel is handled inside these functions and controlled by the I/O variables `IO_CAN.ECU.CAN_If`.

Listing 16.7: Function prototypes in Gateway mode

```

1:  /* Function definitions */
2:  int IO_CAN_Init_First (void);
3:  int IO_CAN_Init      (void);
4:  int IO_CAN_Recv      (struct CAN_Msg *Msg, const unsigned CycleNo);
5:  void IO_CAN_RecvLoopGw (const unsigned CycleNo);
6:  int IO_CAN_SendGw    (const unsigned CycleNo);
7:  int IO_CAN_ReadyToSend(int MsgId, const unsigned CycleNo);
8:  int IO_CAN_SendMsgGw (const unsigned CycleNo, int MsgId);

```

With the functions defined in [Listing 16.6](#) (or [Listing 16.7](#) in Gateway mode), all necessary steps to handle the imported CAN messages and signals are covered. The user only has to assign the value of model variables to the appropriate I/O variables and vice versa, while calibration is done automatically in the inline functions `IO_CAN_Msg_Signal_GetV()` and `IO_CAN_Msg_Signal_SetV()`. To decode and encode the I/O variables from/into CAN messages, some macros are used internally. They are listed in the header file only for the sake of completeness.

Listing 16.8: Inline functions and macro definitions

```

1:  /* Makros to encode/decode values of signals */
2:  /* converting to signed value */
3:  #define IO_CAN_Cvt2Signed(v, dw, tmp) ( \
4:      ((tmp=v)&(((dw>32) ? 111 : 1)<<(dw-1))) ? \
5:          (tmp|(((dw>32) ? -111 : -1)<<(dw-1))) : tmp \
6:  )
7:
8:  static inline int
9:  IO_CAN_Cvt2SignedI(
10:     int      v,
11:     unsigned dw)
12: {
13:     return (v & (1<<(dw-1))) ? v | (-1<<(dw-1)) : v;
14: }
15:
16: static inline long long
17: IO_CAN_Cvt2SignedLL(
18:     long long v,
19:     unsigned dw)
20: {
21:     return (v & (1LL<<(dw-1))) ? v | (-1LL<<(dw-1)) : v;
22: }
23:
24: /* converting to float/double value */
25: static inline float
26: IO_CAN_Raw2Float(
27:     unsigned x)
28: {
29:     return *(float *)((void *)&x);
30: }
31: static inline unsigned
32: IO_CAN_Float2Raw(
33:     float x)
34: {
35:     return *(unsigned *)((void *)&x);
36: }
37:
38: static inline double
39: IO_CAN_Raw2Double(
40:     unsigned long long x)
41: {
42:     return *(double *)((void *)&x);
43: }
44:
45: static inline unsigned long long
46: IO_CAN_Double2Raw(
47:     double x)
48: {
49:     return *(unsigned long long *)((void *)&x);
50: }
51:
52: /* to fix range of value */
53: #define IO_CAN_Range(v, min, max) ( \
54:     (v>=min) ? ((v<=max) ? v : max) : min \
55: )
56: #define IO_CAN_FixMin(v, min) ((v>=min) ? v : min)
57: #define IO_CAN_FixMax(v, max) ((v<=max) ? v : max)

```

Listing 16.8: Inline functions and macro definitions

```

58: #define IO_CAN_Engine_1_ID 513u      /* MsgId=0x201 */
59: #define IO_CAN_Engine_1_NAME "Engine_1"
60: #define IO_CAN_Engine_1_DLC 8
61:
62: #define IO_CAN_Engine_1_E1_Data1_StartValue 0
63: #define IO_CAN_Engine_1_E1_Data1_Raw2Phys(raw) \
64:   ((raw))
65: #define IO_CAN_Engine_1_E1_Data1_Decode(data) \
66:   (((data)[1] | ((data)[2]<<8)))
67: static inline unsigned short
68: IO_CAN_Engine_1_E1_Data1_GetV(
69:   unsigned char *data,
70:   unsigned short *val)
71: {
72:   IO_CAN.Engine.Engine_1.E1_Data1_Raw = IO_CAN_Engine_1_E1_Data1_Decode(data);
73:   switch (IO_CAN.Engine.Engine_1.E1_Data1_Raw) {
74: #if defined(IO_CAN_Engine_1_E1_Data1_INVALIDVALUE)
75:   case IO_CAN_Engine_1_E1_Data1_INVALIDVALUE:
76:     IO_CAN.Engine.Engine_1.E1_Data1_State = IO_CAN_SigStates[IO_CAN_SState_Invalid];
77:     break;
78: #endif
79:   default:
80:     IO_CAN.Engine.Engine_1.E1_Data1_State = IO_CAN_SigStates[IO_CAN_SState_Valid];
81:   }
82:   if (IO_CAN.Engine.Engine_1.E1_Data1_State
83: == IO_CAN_SigStates[IO_CAN_SState_Valid])
84:     *val = IO_CAN_Engine_1_E1_Data1_Raw2Phys(IO_CAN.Engine.Engine_1.E1_Data1_Raw);
85:   return *val;
86: }
87:
88: #define IO_CAN_Engine_1_E1_Data1_Phys2Raw(val) \
89:   ((val))
90: #define IO_CAN_Engine_1_E1_Data1_Encode(data, raw) { \
91:   (data)[1] = (IO_CAN_BitOpUS_And((raw), 255)); \
92:   (data)[2] = (IO_CAN_BitOpUS_RShift((raw), 8) & 255); \
93: }
94: static inline void
95: IO_CAN_Engine_1_E1_Data1_SetV(
96:   unsigned char *data,
97:   unsigned short val)
98: {
99:   if (IO_CAN.Engine.Engine_1.E1_Data1_State
100: == IO_CAN_SigStates[IO_CAN_SState_Valid])
101:     IO_CAN.Engine.Engine_1.E1_Data1_Raw = IO_CAN_Engine_1_E1_Data1_Phys2Raw(val);
102: #if defined(IO_CAN_Engine_1_E1_Data1_INVALIDVALUE)
103:   else if (IO_CAN.Engine.Engine_1.E1_Data1_State
104: == IO_CAN_SigStates[IO_CAN_SState_Invalid])
105:     IO_CAN.Engine.Engine_1.E1_Data1_Raw = IO_CAN_Engine_1_E1_Data1_INVALIDVALUE;
106: #endif
107: #if defined(IO_CAN_Engine_1_E1_Data1_ERRORVALUE)
108:   else if (IO_CAN.Engine.Engine_1.E1_Data1_State
109: == IO_CAN_SigStates[IO_CAN_SState_Error])
110:     IO_CAN.Engine.Engine_1.E1_Data1_Raw =
111:       IO_CAN_Engine_1_E1_Data1_ERRORVALUE;
112: #endif
113: #if defined(IO_CAN_Engine_1_E1_Data1_UNDEFINEDVALUE)
114:   else if (IO_CAN.Engine.Engine_1.E1_Data1_State
115: == IO_CAN_SigStates[IO_CAN_SState_Undefined])
116:     IO_CAN.Engine.Engine_1.E1_Data1_Raw =
117:       IO_CAN_Engine_1_E1_Data1_UNDEFINEDVALUE;
118: #endif
119: #if defined(IO_CAN_Engine_1_E1_Data1_UNAVAILABLEVALUE)
120:   else if (IO_CAN.Engine.Engine_1.E1_Data1_State
121: == IO_CAN_SigStates[IO_CAN_SState_Unavailable])
122:     IO_CAN.Engine.Engine_1.E1_Data1_Raw =
123:       IO_CAN_Engine_1_E1_Data1_UNAVAILABLEVALUE;
124: #endif

```

Listing 16.8: Inline functions and macro definitions

```

125: #if defined(IO_CAN_SIGSTATE_UNKNOWN_CONVERT)
126:     else
127:         IO_CAN.Engine.Engine_1.E1_Data1_Raw =
128:             IO_CAN_Engine_1_E1_Data1_Phys2Raw(val);
129: #endif
130:     IO_CAN_Engine_1_E1_Data1_Encode(data, IO_CAN.Engine.Engine_1.E1_Data1_Raw);
131: }

```

16.3.3 The list of generated I/O variables in IO_VarList.txt

The intention of this file is to give the user a short and clear overview of all generated I/O variables. This is useful, because the user still has to complete the assignment between I/O variables and model variables in the I/O module file `IO.c`, manually. In order to assist the user in this step as much as possible, the file `IO_VarList.txt` lists all CAN signals by name, together with their C data type and their full variable name. Again, this list is arranged hierarchically by ECUs and CAN messages:

Listing 16.9: List of generated I/O variables

```

1: ECU Engine:
2: #####
3:
4: CAN Msg 0x201 (Engine_1):
5: signal      data type      full variable name
6: -----
7: Engine_1_DLC int          IO_CAN.Engine.Engine_1_DLC
8: E1_Data1    unsigned short IO_CAN.Engine.Engine_1.E1_Data1
9:           unsigned short IO_CAN_Raw.Engine.Engine_1.E1_Data1
10:          char          IO_CAN_State.Engine.Engine_1.E1_Data1
11: E1_Data2    unsigned short IO_CAN.Engine.Engine_1.E1_Data2
12:           unsigned short IO_CAN_Raw.Engine.Engine_1.E1_Data2
13:          char          IO_CAN_State.Engine.Engine_1.E1_Data2
14: E1_Data3    float          IO_CAN.Engine.Engine_1.E1_Data3
15:           unsigned char IO_CAN_Raw.Engine.Engine_1.E1_Data3
16:          char          IO_CAN_State.Engine.Engine_1.E1_Data3
17: E1_Info     unsigned char IO_CAN.Engine.Engine_1.E1_Info
18:           unsigned char IO_CAN_Raw.Engine.Engine_1.E1_Info
19:          char          IO_CAN_State.Engine.Engine_1.E1_Info
20: E1_MPlex   unsigned char IO_CAN.Engine.Engine_1.E1_MPlex
21:           unsigned char IO_CAN_Raw.Engine.Engine_1.E1_MPlex
22:          char          IO_CAN_State.Engine.Engine_1.E1_MPlex
23:
24: CAN Msg 0x202 (Engine_2):
25: signal      data type      full variable name
26: -----
27: Engine_2_DLC int          IO_CAN.Engine.Engine_2_DLC
28: E2_Data1    unsigned int   IO_CAN.Engine.Engine_2.E2_Data1
29:           unsigned int   IO_CAN_Raw.Engine.Engine_2.E2_Data1
30:          char          IO_CAN_State.Engine.Engine_2.E2_Data1
31: E2_Data2    unsigned char IO_CAN.Engine.Engine_2.E2_Data2
32:           unsigned char IO_CAN_Raw.Engine.Engine_2.E2_Data2
33:          char          IO_CAN_State.Engine.Engine_2.E2_Data2
34:
35:

```

Listing 16.9: List of generated I/O variables

```

36: ECU ESP:
37: #####
38:
39: CAN Msg 0x101 (ESP):
40: signal data type full variable name
41: -----
42: ESP_DLC int IO_CAN.ESP.ESP_DLC
43: E_Data1 unsigned short IO_CAN.ESP.ESP.E_Data1
44: unsigned short IO_CAN_Raw.ESP.ESP.E_Data1
45: char IO_CAN_State.ESP.ESP.E_Data1
46: E_Data2 unsigned int IO_CAN.ESP.ESP.E_Data2
47: int IO_CAN_Raw.ESP.ESP.E_Data2
48: char IO_CAN_State.ESP.ESP.E_Data2
49: E_Data3 float IO_CAN.ESP.ESP.E_Data3
50: short IO_CAN_Raw.ESP.ESP.E_Data3
51: char IO_CAN_State.ESP.ESP.E_Data3
52:
53:
54: ECU StWhlSensor:
55: #####
56:
57: CAN Msg 0x001 (StWhlAng):
58: signal data type full variable name
59: -----
60: StWhlAng_DLC int IO_CAN.StWhlSensor.StWhlAng_DLC
61: S_Counter unsigned char IO_CAN.StWhlSensor.StWhlAng.S_Counter
62: unsigned char IO_CAN_Raw.StWhlSensor.StWhlAng.S_Counter
63: char IO_CAN_State.StWhlSensor.StWhlAng.S_Counter
64: S_Data1 float IO_CAN.StWhlSensor.StWhlAng.S_Data1
65: short IO_CAN_Raw.StWhlSensor.StWhlAng.S_Data1
66: char IO_CAN_State.StWhlSensor.StWhlAng.S_Data1
67: S_Data1Inv unsigned char IO_CAN.StWhlSensor.StWhlAng.S_Data1Inv
68: unsigned char IO_CAN_Raw.StWhlSensor.StWhlAng.S_Data1Inv
69: char IO_CAN_State.StWhlSensor.StWhlAng.S_Data1Inv
70: S_Data2 unsigned short IO_CAN.StWhlSensor.StWhlAng.S_Data2
71: short IO_CAN_Raw.StWhlSensor.StWhlAng.S_Data2
72: char IO_CAN_State.StWhlSensor.StWhlAng.S_Data2
73:
74:
75: # Timing variables #####
76:
77: ECU Engine:
78: message data type full variable name
79: -----
80: Engine_1 (0x201) int IO_CAN_Timings.Engine.Engine_1.SendPeriod
81: int IO_CAN_Timings.Engine.Engine_1.SendDistrib
82: Engine_2 (0x202) int IO_CAN_Timings.Engine.Engine_2.SendPeriod
83: int IO_CAN_Timings.Engine.Engine_2.SendDistrib
84:
85: ECU StWhlSensor:
86: message data type full variable name
87: -----
88: StWhlAng (0x001) int IO_CAN_Timings.StWhlSensor.StWhlAng.SendPeriod
89: int IO_CAN_Timings.StWhlSensor.StWhlAng.SendDistrib

```

In Gateway mode, the CAN interface parameters and the *_GwNoDelay timing variables are also listed in the file *IO_VarList.txt*.

Listing 16.10: List of generated I/O variables in Gateway mode

```

1: ECU Engine:
2: #####
3: Slot No: int IO_CAN.Engine.CAN_If.Slot
4: Channel No: int IO_CAN.Engine.CAN_If.Channel
5:
6: CAN Msg 0x201 (Engine_1):
7: signal data type full variable name
8: -----
9: Engine_1_DLC int IO_CAN.Engine.Engine_1_DLC
10: Engine_1_RxCnt int IO_CAN.Engine.Engine_1_RxCnt
11: E1_Data1 unsigned short IO_CAN.Engine.Engine_1.E1_Data1
12: E1_Data1 unsigned short IO_CAN_Raw.Engine.Engine_1.E1_Data1
13: char IO_CAN_State.Engine.Engine_1.E1_Data1
14: E1_Data2 unsigned short IO_CAN.Engine.Engine_1.E1_Data2
15: E1_Data2 unsigned short IO_CAN_Raw.Engine.Engine_1.E1_Data2
16: char IO_CAN_State.Engine.Engine_1.E1_Data2
17: E1_Data3 float IO_CAN.Engine.Engine_1.E1_Data3
18: E1_Data3 unsigned char IO_CAN_Raw.Engine.Engine_1.E1_Data3
19: E1_Data3 char IO_CAN_State.Engine.Engine_1.E1_Data3
20: E1_Info unsigned char IO_CAN.Engine.Engine_1.E1_Info
21: E1_Info unsigned char IO_CAN_Raw.Engine.Engine_1.E1_Info
22: E1_Info char IO_CAN_State.Engine.Engine_1.E1_Info
23: E1_MPlex unsigned char IO_CAN.Engine.Engine_1.E1_MPlex
24: E1_MPlex unsigned char IO_CAN_Raw.Engine.Engine_1.E1_MPlex
25: E1_MPlex char IO_CAN_State.Engine.Engine_1.E1_MPlex
26:
27: CAN Msg 0x202 (Engine_2):
28: signal data type full variable name
29: -----
30: Engine_2_DLC int IO_CAN.Engine.Engine_2_DLC
31: Engine_2_RxCnt int IO_CAN.Engine.Engine_2_RxCnt
32: E2_Data1 unsigned int IO_CAN.Engine.Engine_2.E2_Data1
33: E2_Data1 unsigned int IO_CAN_Raw.Engine.Engine_2.E2_Data1
34: E2_Data1 char IO_CAN_State.Engine.Engine_2.E2_Data1
35: E2_Data2 unsigned char IO_CAN.Engine.Engine_2.E2_Data2
36: E2_Data2 unsigned char IO_CAN_Raw.Engine.Engine_2.E2_Data2
37: E2_Data2 char IO_CAN_State.Engine.Engine_2.E2_Data2
38:
39:
40: ECU ESP:
41: #####
42: Slot No: int IO_CAN.ESP.CAN_If.Slot
43: Channel No: int IO_CAN.ESP.CAN_If.Channel
44:
45: CAN Msg 0x101 (ESP):
46: signal data type full variable name
47: -----
48: ESP_DLC int IO_CAN.ESP.ESP_DLC
49: ESP_RxCnt int IO_CAN.ESP.ESP_RxCnt
50: E_Data1 unsigned short IO_CAN.ESP.ESP_E_Data1
51: E_Data1 unsigned short IO_CAN_Raw.ESP.ESP_E_Data1
52: E_Data1 char IO_CAN_State.ESP.ESP_E_Data1
53: E_Data2 unsigned int IO_CAN.ESP.ESP_E_Data2
54: E_Data2 int IO_CAN_Raw.ESP.ESP_E_Data2
55: E_Data2 char IO_CAN_State.ESP.ESP_E_Data2
56: E_Data3 float IO_CAN.ESP.ESP_E_Data3
57: E_Data3 short IO_CAN_Raw.ESP.ESP_E_Data3
58: E_Data3 char IO_CAN_State.ESP.ESP_E_Data3
59:

```

Listing 16.10: List of generated I/O variables in Gateway mode

```

60: ECU StWhlSensor:
61: #####
62:
63: CAN Msg 0x001 (StWhlAng):
64: signal      data type      full variable name
65: -----
66: StWhlAng_DLC   int          IO_CAN.StWhlSensor.StWhlAng_DLC
67: StWhlAng_RxCnt int          IO_CAN.StWhlSensor.StWhlAng_RxCnt
68: S_Counter     unsigned char IO_CAN.StWhlSensor.StWhlAng_S_Counter
69:           unsigned char IO_CAN.Raw.StWhlSensor.StWhlAng.S_Counter
70:           char        IO_CAN.State.StWhlSensor.StWhlAng.S_Counter
71: S_Data1       float         IO_CAN.StWhlSensor.StWhlAng.S_Data1
72:           short        IO_CAN.Raw.StWhlSensor.StWhlAng.S_Data1
73:           char        IO_CAN.State.StWhlSensor.StWhlAng.S_Data1
74: S_Data1Inv    unsigned char IO_CAN.StWhlSensor.StWhlAng.S_Data1Inv
75:           unsigned char IO_CAN.Raw.StWhlSensor.StWhlAng.S_Data1Inv
76:           char        IO_CAN.State.StWhlSensor.StWhlAng.S_Data1Inv
77: S_Data2       unsigned short IO_CAN.StWhlSensor.StWhlAng.S_Data2
78:           short        IO_CAN.Raw.StWhlSensor.StWhlAng.S_Data2
79:           char        IO_CAN.State.StWhlSensor.StWhlAng.S_Data2
80:
81:
82: # Timing variables #####
83:
84: ECU Engine:
85: message      data type  full variable name
86: -----
87: Engine_1 (0x201) int      IO_CAN_Timings.Engine.Engine_1_GwNoDelay
88:           int      IO_CAN_Timings.Engine.Engine_1.SendPeriod
89:           int      IO_CAN_Timings.Engine.Engine_1.SendDistrib
90: Engine_2 (0x202) int      IO_CAN_Timings.Engine.Engine_2_GwNoDelay
91:           int      IO_CAN_Timings.Engine.Engine_2.SendPeriod
92:           int      IO_CAN_Timings.Engine.Engine_2.SendDistrib
93:
94: ECU ESP:
95: message      data type  full variable name
96: -----
97: ESP (0x101) int      IO_CAN_Timings.ESP.ESP_GwNoDelay
98:           int      IO_CAN_Timings.ESP.ESP.SendPeriod
99:           int      IO_CAN_Timings.ESP.ESP.SendDistrib
100:
101: ECU StWhlSensor:
102: message      data type  full variable name
103: -----
104: StWhlAng (0x001) int      IO_CAN_Timings.StWhlSensor.StWhlAng_GwNoDelay
105:           int      IO_CAN_Timings.StWhlSensor.StWhlAng.SendPeriod
106:           int      IO_CAN_Timings.StWhlSensor.StWhlAng.SendDistrib

```

16.3.4 The C module IO_CAN.c

This module manages the data flow on the virtual CAN bus. On reception of a CAN message, the data bytes are decoded, converted to physical values and stored in the associated I/O variables. For the transmission of CAN messages, the same is done in the reverse order.

However, the connection between the CarMaker model environment and the simulated CAN bus cannot be done without the help of the user. In order to separate this module from customizations by the user, there are a couple of calls to functions in the module *IO_CAN_User*. This module is described in [section 16.3.5 'The header file IO_CAN_User.h'](#) and [section 16.3.6 'The C module IO_CAN_User.c'](#).

Initialization

At the beginning, there is always needed an accurate initialization. This is done by the functions `IO_CAN_Init_First()` and `IO_CAN_Init()`.

Global variables, like the `IO_CAN` data structure with the I/O variables, are initialized in the function `IO_CAN_Init_First()`. This function also calls `IO_CAN_User_Init_First()` from the module `IO_CAN_User`.

After a short plausibility check, if some interface variables are initialized, the function `IO_CAN_Init()` just calls `IO_CAN_User_Init()`, to let the user control which Data Dictionary entries should be defined.

Beside the declaration of Data Dictionary entries, the function `IO_CAN_User_Init()` is also responsible for a correct initialization of the I/O variables of all CAN signals and the timing parameters for transmitted CAN messages.

To handle the reception and transmission of CAN messages, CANiogen provides the function pointers `IO_CAN_CAN_Recv` and `IO_CAN_CAN_Send`:

```
int (*IO_CAN_CAN_Recv) (int Slot, int Channel, struct CAN_Msg *Msg);
int (*IO_CAN_CAN_Send) (int Slot, int Channel, struct CAN_Msg *Msg);
```

Depending on the hardware platform, these function pointers are initialized with `MIO_M51_Send()`/`MIO_M51_Recv()` (LynxOS / XENO) or `DSIO_CAN_Recv()`/`DSIO_CAN_Send()` (dSPACE). If necessary, the user can change it to his own implementation in the function `IO_CAN_Init_User_First()`.

For better control of the CAN traffic and in order to implement special features, like a message counter or a checksum, CANiogen provides two function pointers – `IO_CAN_RxHook` and `IO_CAN_TxHook`:

```
int (*IO_CAN_RxHook) (struct CAN_Msg *Msg, unsigned CycleNo, int Part);
int (*IO_CAN_TxHook) (struct CAN_Msg *Msg, unsigned CycleNo, int Part);
```

If necessary, the user can assign his own implementations to these function pointers. They will be called with every Tx- or Rx message and allow to manipulate the raw data bytes of a received or transmitted CAN message. These function pointers can be activated by the user in the function `IO_CAN_User_Init_First()`. For an example implementation, refer to Example implementations for Rx- and Tx Hook functions.

Reception of CAN messages

Received CAN messages are decoded and evaluated by the functions `IO_CAN_Recv()` and `IO_CAN_RecvLoop()`.

The function `IO_CAN_Recv()` expects a single CAN message as argument and tries to decode it by checking the message identifier. `IO_CAN_Recv()` accepts both, standard CAN messages with 12-bit-identifiers and extended CAN messages with 29-bit-identifiers.

As an alternative to the functions `IO_CAN_Recv()`, CAN messages can also be processed through the function `IO_CAN_RecvLoop()`. In this case, all available CAN messages will be fetched from the CAN controller and processed immediately. This function is useful, if there is no further CAN traffic than those of the imported CAN database. Calls to the function `IO_CAN_RecvLoop()` should not be mixed with the conventional method.



In Gateway mode, the functionality to scan all CAN controllers autonomously is provided by the function `IO_CAN_RecvLoopGw()`. This function does not need a `Slot` and `Channel` number as function parameter. Instead, the function uses the I/O variables `IO_CAN.ECU.CAN_If` to determine for each real ECU which CAN interface to scan for Tx messages.

With every received CAN message – if defined – the user defined callback function `IO_CAN_RxHook()` is called twice. The first call to this function is done before the data bytes of the message are decoded to the I/O signals. Thus, it is possible to directly affect the val-

ues of CAN signals, before they are assigned to I/O and model variables. Even more, the return value of `IO_CAN_RxHook()` decides, if the message will be discarded (return value < 0), or not (return value >= 0).

After decoding all the CAN signals out of the CAN message, `IO_CAN_RxHook()` is called again. Now, the user can directly react to the reception of a specific CAN message, while having access to the newly updated I/O variables. Unlike the first call, the return value is now ignored.

Transmission of CAN messages

The transmission of CAN messages is handled by the function `IO_CAN_Send()`. This function requires the cycle number `CycleNo` of the main simulation as argument. With the help of the cycle number and the timing parameters for each CAN message it is decided, whether a transmission of the message is required or not. A CAN message is transmitted, whenever the following condition is fulfilled:

```
(CycleNo%IO_CAN_Timings.ECU.MsgName.SigName.SendPeriod > 0
&& CycleNo%IO_CAN_Timings.ECU.MsgName.SigName.SendPeriod
== IO_CAN_Timings.ECU.MsgName.SigName.SendDistrib)
```



In Gateway mode, the condition for transmission of a CAN message is also controlled by the `*_GwNoDelay` timing variable:

```
((IO_CAN.ECU.MsgName_RxCnt > 0 && IO_CAN_Timings.ECU.MsgName_GwNoDelay)
|| (CycleNo%IO_CAN_Timings.ECU.MsgName.SigName.SendPeriod > 0
&& (CycleNo%IO_CAN_Timings.ECU.MsgName.SigName.SendPeriod
== IO_CAN_Timings.ECU.MsgName.SigName.SendDistrib)))
```

Knowing this, you can easily change the timings of cyclic CAN messages. It is even possible to suppress the transmission with the help of a few lines of code. You just have to modify the timing parameters of a message before calling the function `IO_CAN_Send()`. Of course, you should not forget to restore the timing parameters after returning from the function `IO_CAN_Send()`:

Listing 16.11: Sending CAN messages manually

```

1:  #      if defined (WITH_IO_CAN)
2:          /* suppress transmission of message StWhlAng */
3:          if (Suppress_StWhlAng == 1) {
4:              /* backup timing parameters */
5:              OldSendPeriod = IO_CAN_Timings.StWhlSensor.StWhlAng.SendPeriod;
6:              OldSendDistrib = IO_CAN_Timings.StWhlSensor.StWhlAng.SendDistrib;
7:              /* suppress transmission */
8:              IO_CAN_Timings.StWhlSensor.StWhlAng.SendPeriod = -1;
9:              IO_CAN_Timings.StWhlSensor.StWhlAng.SendDistrib = -1;
10:         }
11:         IO_CAN_Send (Slot.CAN, 0, CycleNo);
12:         if (Suppress_StWhlAng == 1) {
13:             /* restore timing parameters */
14:             IO_CAN_Timings.StWhlSensor.StWhlAng.SendPeriod = OldSendPeriod;
15:             IO_CAN_Timings.StWhlSensor.StWhlAng.SendDistrib = OldSendDistrib;
16:         }
17:     #      endif

```

CANiogen also provides a function to check the condition, if a CAN message is going to be sent, or not:

```
int IO_CAN_ReadyToSend(int MsgId, unsigned CycleNo);
```

This function takes a message identifier `MsgId` and the cycle number `CycleNo`, and checks, if the condition mentioned above is true for this CAN message, or not.



The function `IO_CAN_ReadyToSend()` evaluates only the cycle number `CycleNo` and the timing variable of the CAN message with the identifier `MsgId`. I.e. there is no difference if the function is called before or after `IO_CAN_Send()`.

For instance, if you want to implement a message counter in a Tx message, you can use the following code construct:

Listing 16.12: Realizing a counter in a CAN message

```

1: #      if defined (WITH_IO_CAN)
2:         /* check if StWhlAng message is going to be sent -> increment counter */
3:         if (IO_CAN_ReadyToSend(IO_CAN_StWhlAng_ID, CycleNo))
4:             IO_CAN.StWhlSensor.StWhlAng.S_Counter++;
5:             IO_CAN_Send (Slot.CAN, 0, CycleNo);
6: #      endif

```



In Gateway mode, the function `IO_CAN_SendGw()` needs to be called instead of `IO_CAN_Send()`. The example of Listing 16.12 looks as follows:

Listing 16.13: Realizing a counter in a CAN message (Gateway mode)

```

1: #      if defined (WITH_IO_CAN)
2:         /* check if StWhlAng message is going to be sent -> increment counter */
3:         if (IO_CAN_ReadyToSend(IO_CAN_StWhlAng_ID, CycleNo))
4:             IO_CAN.StWhlSensor.StWhlAng.S_Counter++;
5:             IO_CAN_SendGw (CycleNo);
6: #      endif

```

However, the examples above are good to control the transmission of cyclic CAN messages. But, what if we want to send CAN messages non cyclic? For this purpose, CANiogen generates the following function:

```
int IO_CAN_SendMsg (int Slot, int Channel, unsigned CycleNo, int MsgId);
```

This function takes the same arguments as `IO_CAN_Send()`, plus the message identifier `MsgId` of the CAN message to send. It forces the selected CAN message to be sent immediately.



In Gateway Mode, the transmission of a CAN message is handled by the function `IO_CAN_SendMsgGw()`. This function does not require a `Slot` and `Channel` number, but passes an unique `ECU_Id` to the user function '`IO_CAN_User_CAN_SendGw()`'. This function identifies the sender of a CAN message by the `ECU_Id` parameter and decides internally with the help of the `IO_CAN.ECU.CAN_If` variables, to determine on which CAN interfaces the message needs to be transmitted:

```
int IO_CAN_SendMsgGw (unsigned CycleNo, int MsgId);
```

Another way to influence the transmission of CAN messages is to use the user defined call-back function `IO_CAN_TxHook()`. This function – if defined – is called twice with every Tx CAN message. First, it is called before the I/O signal variables are encoded into the data bytes of a CAN messages. Here, the user can make changes to the affected I/O variables, e.g. implement a message counter.

After the I/O signals of the message have been encoded to the data bytes, `IO_CAN_TxHook()` is called again. Now, it is possible to directly affect the data bytes of a CAN message before the message is written to the CAN bus. Even more, the return value of `IO_CAN_TxHook()` decides if the message will be sent (return value ≥ 0) or not (return value < 0).

Example implementations for Rx- and Tx Hook functions

With Rx- and Tx hook functions, the user can control the reception and transmission of CAN messages, he is able to modify the contents of CAN messages or to check the validity of signals. But, he can also implement additional functionality like message counters or the calculation of checksums, etc. CANiogen expects the hook functions to accept a pointer to a data structure of type `CAN_Msg`, the unsigned integer `CycleNo` and the integer `Part` as input parameters:

```
int (*IO_CAN_RxHook) (struct CAN_Msg *Msg, unsigned CycleNo, int Part);
int (*IO_CAN_TxHook) (struct CAN_Msg *Msg, unsigned CycleNo, int Part);
```

Each hook function is called twice for each CAN message, whereas `Part` indicates whether it is the first or the second call:

- for Rx messages, the hook function is called with `Part = IO_CAN_PRE_DECODE` before the data bytes are decoded. With the second call, after all I/O variables have been updated, `Part` has the value `IO_CAN_POST_DECODE`.
- for Tx messages, the hook function is called first with `Part = IO_CAN_PRE_ENCODE`, before any I/O variable is encoded into the data bytes of the CAN message. Here, the I/O variables can still be modified, e.g. to implement a message counter. With the second call, `Part` has the value `IO_CAN_POST_ENCODE`, all I/O signal variable have been encoded into the data bytes of the CAN message.

The hook functions should return an integer value. A return value lower than zero (`<0`) means that the CAN message will be discarded. If the CAN message is left unmodified, the hook function should return zero (0), whereas a modified/manipulated CAN message should be notified with a return value greater than zero (`>0`). If the return value is equal or greater than zero (`>=0`) a Tx CAN message will be transmitted and a Rx CAN message will be evaluated by decoding its signals and storing to appropriate I/O variables respectively.

With the module `IO_CAN_User.c`, CANiogen also generates templates for the hook functions. They can be activated in the function `IO_CAN_User_Init_First()` and modified to fit the needs of the test bench.

The following listings are examples, based on the templates created by CANiogen:

Listing 16.14: Example for Rx Hook function

```
1: static int
2: IO_CAN_User_RxHook(
3:     struct CAN_Msg *Msg,
4:     const unsigned CycleNo,
5:     int             Part)
6: {
7:     switch (Msg->MsgId) {
8:         /* ECU StWhlSensor */
9:         case IO_CAN_StWhlAng_ID: { /* CAN Msg 0x001 (StWhlAng) */
10:             switch (Part) {
11:                 case IO_CAN_PRE_DECODE:
12:                     /* if data of StWhlAng message is invalid -> discard CAN message */
13:                     if (IO_CAN_StWhlAng_S_Data1Inv_SetV(Msg->Data))
14:                         return -1;
15:                     break;
16:                 case IO_CAN_POST_DECODE:
17:                     break;
18:                 }
19:             return 0;
20:         }
21:
22:         /* ECU ESP */
23:         case IO_CAN_ESP_ID: { /* CAN Msg 0x101 (ESP) */
24:             switch (Part) {
25:                 case IO_CAN_PRE_DECODE:
26:                     if (Msg->Data[1] == 0xb0) {
27:                         /* modify raw data byte #0 in CAN message */
28:                         Msg->Data[0] = 0xf1;
29:                         return 1;
30:                     }
31:                     break;
32:                 case IO_CAN_POST_DECODE:
33:                     break;
34:                 }
35:             return 0;
36:         }
37:         default:
38:             break;
39:         }
40:
41:     return -1;
42: }
```

Listing 16.15: Example for Tx Hook function

```

1: static int
2: IO_CAN_User_TxHook(
3:     struct CAN_Msg *Msg,
4:     const unsigned CycleNo,
5:     int             Part)
6: {
7:     switch (Msg->MsgId) {
8:         /* ECU StWhlSensor */
9:         case IO_CAN_StWhlAng_ID: { /* CAN Msg 0x001 (StWhlAng) */
10:             switch (Part) {
11:                 case IO_CAN_PRE_ENCODE:
12:                     /* increment message counter */
13:                     if (++IO_CAN_StWhlSensor.StWhlAng.S_Counter >= 32)
14:                         IO_CAN_StWhlSensor.StWhlAng.S_Counter = 0;
15:                     break;
16:                 case IO_CAN_POST_ENCODE:
17:                     break;
18:                 }
19:             return 0;
20:         }
21:         /* ECU Engine */
22:         case IO_CAN_Engine_1_ID: { /* CAN Msg 0x201 (Engine_1) */
23:             switch (Part) {
24:                 case IO_CAN_PRE_ENCODE:
25:                     break;
26:                 case IO_CAN_POST_ENCODE:
27:                     if (Msg->Data[1] == 0xb0) {
28:                         /* modify raw data byte #0 in CAN message */
29:                         Msg->Data[0] = 0xf1;
30:                         return 1;
31:                     } else if (Msg->Data[1] == 0xb1) {
32:                         /* discard CAN message */
33:                         return -1;
34:                     }
35:                     if (IO_CAN_Engine.Engine_1.E1_MPlex == 0) {
36:                         /* manipulate signal E1_Info */
37:                         IO_CAN_Engine_1_E1_Data1_SetV(Msg.Data, -1.3);
38:                         return 1;
39:                     }
40:                     break;
41:                 }
42:             return 0;
43:         default:
44:             break;
45:         }
46:     return -1;
47: }
48: }
```

16.3.5 The header file `IO_CAN_User.h`

This file declares the user interface functions which are used by the main module `IO_CAN.c` and the user module `User.c` of CarMaker.



CANiogen will not modify or overwrite this file, if it already exists. However, if there are major changes on the CAN database or the virtual or physical CAN environment, it might be advisable to rename it right before executing CANiogen. `IO_CAN_User.h` will then be created again. The user can then customize it with the help of the older version.

16.3.6 The C module IO_CAN_User.c

Customization of the virtual CAN environment is to be done in this file. It is generated automatically, if it does not exist, but will never be overwritten or changed by CANiogen.

On creation, this file holds a framework of functions, well prepared for modifications by the user. This helps to connect the CarMaker model with the virtual CAN environment like transferring outputs of the CarMaker simulation to the CAN bus as well as reading back inputs from connected ECUs.

While most modifications are just a simple assignment between model variable and I/O variable, there are some CAN signals which need a special treatment. CANiogen generates a set of functions, which help the user to implement functionality that cannot be described in CAN databases.



CANiogen will not modify or overwrite this file, if it already exists. However, if there are major changes on the CAN database or the virtual or physical CAN environment, it might be advisable to rename it right before executing CANiogen. `IO_CAN_User.c` will then be created again. Then, the user can customize it with the help of the older version.

`IO_CAN_User_Init_First()`

This function is called by `IO_CAN_Init_First()`, after all global data structures and variables have been initialized. Typically, it is used to initialize own variables and to activate the customized hook functions `IO_CAN_User_RxHook()` and `IO_CAN_User_TxHook()`.



In Gateway mode, this function also needs to initialize the CAN interface informations for each real ECU, by initializing the variables `IO_CAN_CAN_SendGw`, `IO_CAN.ESU.CAN_If.Slot` and `IO_CAN.ESU.CAN_If.Channel`:

```
IO_CAN_CAN_SendGw = IO_CAN_User_CAN_SendGw;
IO_CAN.Engine.CAN_If.Slot      = 2;
IO_CAN.Engine.CAN_If.Channel = 0;
IO_CAN.ESP.CAN_If.Slot        = 2;
IO_CAN.ESP.CAN_If.Channel    = 1;
```

The variable `IO_CAN_CAN_SendGw` is a function pointer to the user function '`IO_CAN_User_CAN_SendGw()`'. This function will be called in Gateway mode by the function `IO_CAN_SendMsgGw()` for each Tx CAN message.

Note, that in Gateway mode each real ECU needs to be connected to the realtime system on a separate CAN interface. Assigning one CAN interface to more than one ECU requires special handling in the user function '`IO_CAN_User_CAN_SendGw()`', to prevent wrong CAN message handling.

`IO_CAN_User_Init()`

Newly created, this function creates Data Dictionary entries for all generated CAN I/O and timing variables. Each generated I/O and timing variable will also be initialized with its start value, which is – if specified – extracted from the CAN database.

It is up to the user to change the start values and to add new or delete Data Dictionary entries.

IO_CAN_User_In()

This function will be called by the function `User_In()` in the module `User.c` of CarMaker in every simulation step. It can be used for general assignment of CAN I/O values to model variables, which do not need a special treatment. However, there is no knowledge about if the CAN I/O variables were updated since the last call.

IO_CAN_User_Out()

This function will be called by the function `User_Out()` in the module `User.c` of CarMaker in every simulation step. It can be used for general assignment of model values to CAN I/O variables, which do not need a special treatment. In order to know if a specific CAN messages is going to be sent, the user might need to check it with `IO_CAN_ReadyToSend()`.

IO_CAN_User_RxHook()

Whenever a CAN message is processed by `IO_CAN_Recv()`, this function is called twice: before and after decoding the data bytes.

With the first call, the third argument `Part` will have the value `IO_CAN_PRE_DECODE`. At this point of time, the CAN message is not yet evaluated and can still be manipulated. Hence, any modification to the data bytes of the CAN message will affect the values of related CAN I/O variables. Even more, if this function returns a value `<0`, the message will be discarded. On the other hand, a value of `0` should be returned, if the CAN message was left unmodified, and `>0` if the message was modified.

When called for the second time – with `Part = IO_CAN_POST_DECODE` – all CAN signals have been decoded from the CAN message. Based on the newly updated CAN I/O signal variables, there can be done e.g. some calculations. The return value now will have no effect at all.

IO_CAN_User_TxHook()

During the preparation of a CAN message for its transmission, this function will be called twice, as well: before and after encoding the data bytes.

On the first call, the third argument `Part` will have the value `IO_CAN_PRE_ENCODE`. This is the time for calculations to update the related CAN I/O signal variables. The return value is not relevant.

When called for the second time – with `Part = IO_CAN_POST_ENCODE` – all related CAN I/O signal values have been encoded to the CAN message. It is now still possible to manipulate the raw data bytes of the CAN message. Similar to `IO_CAN_User_RxHook()`, the return value decides, if the CAN message will be sent: `<0` will discard the message. On the other hand, a value of `>0` should be returned, if the message was manipulated, or `0` else.

IO_CAN_User_CAN_SendGw()

This function is only generated in Gateway mode:

```
int IO_CAN_User_CAN_SendGw (tIO_CAN_ECU ECU_Id, struct CAN_Msg *Msg);
```



In order to be able to decide, on which CAN bus a CAN message needs to be sent, this function is called with an unique `ECU_Id` for the sender of the CAN message. Depending on the hardware configuration and the receiver of the CAN message, the user can freely decide how to send the CAN message.

However, CANiogen will generate this function for the default configuration, which means that every gatewayed ECU is connected to its own exclusive CAN bus.

The default implementation of `IO_CAN_User_CAN_SendGw()` is shown below

Listing 16.16: Example for `IO_CAN_User_CAN_SendGw()`

```
1: static int
2: IO_CAN_User_CAN_SendGw(
3:     tIO_CAN_ECU    ECU_Id,
4:     struct CAN_Msg *Msg)
5: {
6:     int res = 0;
7:
8:     switch (ECU_Id) {
9:         case IO_CAN_ECU_Engine:
10:             res |= IO_CAN_CAN_Send(IO_CAN.ESP.CAN_If.Slot, IO_CAN.ESP.CAN_If.Channel,
11:                                     Msg);
12:             break;
13:         case IO_CAN_ECU_ESP:
14:             res |= IO_CAN_CAN_Send(IO_CAN.Engine.CAN_If.Slot,
15:                                     IO_CAN.Engine.CAN_If.Channel, Msg);
16:             break;
17:         case IO_CAN_ECU_StWhlSensor:
18:             res |= IO_CAN_CAN_Send(IO_CAN.Engine.CAN_If.Slot,
19:                                     IO_CAN.Engine.CAN_If.Channel, Msg);
20:             res |= IO_CAN_CAN_Send(IO_CAN.ESP.CAN_If.Slot, IO_CAN.ESP.CAN_If.Channel,
21:                                     Msg);
22:             break;
23:         default:
24:             res = -1;
25:             break;
26:     }
27:
28:     return res;
29: }
```

16.4 CANiogen's output (J1939 Mode)

16.4.1 Output files

In J1939 mode, CANiogen generates the J1939 parameters file *J1939Parameters* and the corresponding mappings file template *J1939Mappings.default* (how to change the naming of the files is described in [section 16.2.9 'Naming of generated files, I/O variables and functions'](#)). The files are generated in the directory *Src*, located in the current CarMaker project, which also includes the *Makefile* by default, unless the output directory is overridden with the option *-outdir*.

The output files *J1939Parameters* and *J1939Mappings.default* will be overwritten without a prompt. You should not modify them.

For customizations, you should create your own mappings file (e.g. *J1939Mappings*). The example mappings file can be used as template, to find out the set of available ECUs, PDUs and signals.

The file *J1939Parameters* may contain some additional settings. Whenever CANiogen is started, it will first import any previously created *J1939Parameters*, if the file already exists. However, the complete ECU, PDU and signal description will be overwritten and replaced by new descriptions from the CAN database, according to the import instructions.

16.4.2 The J1939 Parameters file

The J1939 parameters file *J1939Parameters* lists the available ECUs and defines all received and transmitted PDUs, including the PGN (signal) descriptions. Additionally, a header is included, which lists the current configuration options (command line arguments), and information about the creator. This allows to reconstruct afterwards, how the generation process was done.

Header information

Fileident = CarMaker-J1939 Ver

Identifies the infofile as J1939 parameters file of version *Ver*. Currently supported value for *Ver* is 1.

Description:

Description of file type and contents

Short description about the type of infofile and it's intended use.

Example Description:

This file was automatically generated by CANiogen 2.0.3 for:
CarMaker - Version 3.5
J1939 Rest Bus Simulation

Created = %Y/%m/%d %H:%M:%S user@host

Creation information about when the file was generated, the `user` and the `host` name.
The format of the date and time stamp is as follows:

<code>%Y</code>	Four digit calendar year of the current date
<code>%m</code>	Two digit month of the year with leading zero (01 .. 12)
<code>%d</code>	Two digit day of the month with leading zero (01 .. 31)
<code>%H</code>	Two digit hour of the current time (00 .. 23)
<code>%M</code>	Two digit minute of the current hour (00 .. 59)
<code>%S</code>	Two digit second of the current minute (00 .. 59)

CANdb = Path

Path to the CAN database file, that was used as input.

```
Cmd.srvECU = ECUName_1 ECUName_2 ...
Cmd.simECU = ECUName_1 ECUName_2 ...
Cmd.excECU = ECUName_1 ECUName_2 ...
```

List of ECU names to be served, simulated or excluded, as specified with the option
`-srvecu`, `-simECU`, or `-excECU`.

```
Cmd.TxMsg.<ECUName> = Msg1 Msg2 ...
Cmd.RxMsg.<ECUName> = Msg1 Msg2 ...
```

Names of J1939 messages, that will be sent in the name of `ECUName` (*Cmd.TxMsg*), or that
will be received from `ECUName` (*Cmd.RxMsg*).

ECU description**ECU.<Addr> = Name AAC IG Sys SI Func FI ECUI ManCod Ident**

Defines properties of a J1939 ECU with given address byte `Addr` and named `Name`. The properties of the ECU follow the SAE J1939 specification for *NAME Systems and Functions*:

AAC	Arbitrary Address Capable
IG	Industry Group
Sys	Vehicle System
SI	Vehicle System Instance
Func	Function
FI	Function Instance
ECUI	ECU Instance

ManCod	Manufacturer Code
Ident	Identity Number

PDU description

TxPDU = CAN_Id_1 CAN_Id_2 ...
RxPDU = CAN_Id_1 CAN_Id_2 ...

Lists the CAN message identifiers for all J1939 messages to be sent (*TxPDU*) and all J1939 messages to be received (*RxPDU*).

Please note: J1939 specifies the CAN communication for extended CAN messages with 29 bit identifiers, only.

PDU.<CAN_Id> = Name Sender SA Prio Page PGN PS CycleTime Size

Specifies a PDU with the CAN message identifier `CAN_Id` and named `Name`, with the following properties:

Sender	Name of sender ECU
SA	Address of Sender
Prio	Message priority
Page	Data Page
PGN	Parameter Group Number
PS	PDU specific (Destination address, group extension, or proprietary)
CycleTime	Cycle time in milli seconds
Size	Message size in bytes

The `CAN_Id` of the PDU corresponds to an identifier, which was previously listed either with the Key *TxPDU*, or *RxPDU*.

PDU.<CAN_Id>.SPN: SPN Size Unit Type Resolution Min Max Position Name

Describes the signals within the PDU with CAN identifier `CAN_Id`:

SPN	Suspect Parameter Number
Size	Size of the signal in bits
Unit	Unit of the signal ("-" if no Unit)
Type	Type of signal (signal, measured, status, ascii or binary)
Resolution	Resolution of one data bit
Min	Minimum physical value
Max	Maximum physical value
Position	Bit position of signal in data frame
Name	Name of signal

16.4.3 The J1939 Mappings file

The J1939 mappings file *J1939Mappings.default* lists all Tx- and Rx- signals, grouped by ECU and PDU name. By default, each signal will be mapped to a constant zero value. The user can now extract all signal mappings of interest in his own mappings file and change the mapping to his own needs.

However, some signals might have special functions, as specified in the SAE J1939 specification. If a signal is recognized to be for instance a special counter or checksum signal, CANiogen will create an appropriate mapping.

For a description of the Header information, please refer to the description for The J1939 Parameters file.

There are several different types of mappings possible. Each type of mapping has its own syntax.

Mapping to Data Dictionary variable

TxMap.<ECUName>:

RxMap.<ECUName>:

SPN <PDUName>.<SPNName>	MapVar	[Factor [Offset [Min [Max]]]]
--	---------------	--------------------------------------

This defines a mapping to a Data Dictionary variable. The Data Dictionary variable must exist. Ideally, the J1939 signal and the Data Dictionary variable have the same unit. If not, optional parameters can be specified for conversion or signal conditioning:

ECUName	Name of Sender ECU
SPN	Suspect Parameter Number
PDUName	Name of PDU
SPNName	Name of signal (SPN)
MapVar	Name of Data Dictionary variable
Factor	Factor for conversion, optional
Offset	Offset for conversion, optional
Min	Minimum possible value, optional
Max	Maximum possible value, optional

Mapping to constant value

TxMap.<ECUName>:

SPN <PDUName>.<SPNName>	Const	Value
--	--------------	--------------

This defines a mapping to a constant value. This type of mapping does not make sense for Rx-signals:

ECUName	Name of Sender ECU
SPN	Suspect Parameter Number
PDUName	Name of PDU
SPNName	Name of signal (SPN)

Const	Keyword to define constant		
Value	Constant value for signal		

CRC Mapping

TxMap.<ECUName>:			
RxMap.<ECUName>:			
SPN <PDUName>.<SPNName>	CRC	J1939	
SPN <PDUName>.<SPNName>	CRC	J1939_2	
SPN <PDUName>.<SPNName>	CRC	J1850	[Start [Count]]
SPN <PDUName>.<SPNName>	CRC	<Type>	[Args]

Defines a CRC mapping. Following types of CRCs are supported:

ECUName	Name of Sender ECU
SPN	Suspect Parameter Number
PDUName	Name of PDU
SPNName	Name of signal (SPN)
CRC	Keyword to define CRC signal
J1939	CRC signal, as defined by SAE J1939 specification for the following SPNs: <ul style="list-style-type: none">• 3188: XBR Message Checksum• 3690: Message Checksum• 4975: Crash Checksum
J1939_2	CRC signal, as defined by SAE J1939 specification for the following SPN: <ul style="list-style-type: none">• 4207: Message Checksum
J1850	CRC signal, as defined by SAE J1850
Start	Start byte for CRC calculation in message, optional
Count	Number of data bytes for CRC calculation, optional
Type	User defined keyword for user defined CRC function
Args	Optional / possible arguments for user defined CRC function

Rolling Counter Mapping

TxMap.<ECUName>:				
RxMap.<ECUName>:				
SPN <PDUName>.<SPNName>	RollCnt	Std	[Min [Max [Incr]]]	
SPN <PDUName>.<SPNName>	RollCnt	<Type>	[Args]	

Defines a signal to be a message counter. Following types are supported:

ECUName	Name of Sender ECU
SPN	Suspect Parameter Number
PDUName	Name of PDU
SPNName	Name of signal (SPN)

RollCnt	Keyword to define message counter signal
Std	Standard Rolling Counter: <ul style="list-style-type: none">• starts with minimum value (default 0)• increments with each sent message• wraps back to 0 after maximum reached
Min	Minimum value for Standard Rolling Counter, optional
Max	Maximum value for Standard Rolling Counter, optional
Incr	Step size (increment) for Standard Rolling Counter, optional
Type	User defined keyword for user defined Counter function
Args	Optional / possible arguments for user defined Counter function

16.5 Integration into CarMaker/HIL (C-Code Mode)

This section applies only to CANiogen, if used in the C-code generator mode.



The output of CANiogen are two largely independent C modules `IO_CAN` and `IO_CAN_User`, which are represented through the files `IO_CAN.c`, `IO_CAN.h`, `IO_CAN_User.c` and `IO_CAN_User.h`. While the files `IO_CAN.c` and `IO_CAN.h` should always be left unchanged, the files `IO_CAN_User.c` and `IO_CAN_User.h` can be customized by the user.

The integration into a CarMaker/HIL project, is done in three steps. First, you will have to modify slightly the existing C module files `IO.c` and `User.c` to enable the CAN I/O part. Then you will have to build up the connection between parameters of the simulation models and the generated I/O variables, which is done in `IO_CAN_User.c`. In order to optimize the automatic build process, you may then want to add some new rules to the `Makefile`.

16.5.1 Modifications to IO.c

First, you have to include the header file `IO_CAN.h` in `IO.c`, in order to make known the data structures, functions and macros, which are defined in `IO_CAN.h`:

Listing 16.17: Including `IO_CAN.h` in `IO.c`

```

1: ...
2: #include <CarMaker.h>
3:
4: #if defined (LYNXOS) || defined (XENO)
5: # include <mio.h>
6: #elif defined (DSPACE)
7: # include <dsio.h>
8: #endif
9: #include <FailSafeTester.h>
10:
11: #include "IO.h"
12: #if defined (WITH_IO_CAN)
13: # include "IO_CAN.h"
14: #endif
15: ...

```

Initialization

The initialization of the module `IO_CAN` is done in the two functions `IO_Init_First()` and `IO_Init()`. First, a call to the function `IO_CAN_Init_First()` has to be added to the function `IO_Init_First()`. This guarantees, that all data structures, which are generated by CANiogen, are properly initialized before the first use:

Listing 16.18: Low level initialization of module `IO_CAN`

```
1: int
2: IO_Init_First (void)
3: {
4:     ...
5:
6: #if defined(WITH_IO_CAN)
7:     /* Init I/O data structures */
8:     IO_CAN_Init_First ();
9: #endif
10:
11:    ...
12: }
```

Then, a call to the function `IO_CAN_Init()` has to be added to the function `IO_Init()`, in between the part of the appropriate I/O configuration.

Listing 16.19: Initialization of module `IO_CAN`

```
1: int
2: IO_Init(void)
3: {
4:     ...
5:
6:     /* hardware configuration */
7:     if (ConnectedIO == IO_None)
8:         return 0;
9:
10:    if (ConnectedIO == IO_DemoApp) {
11:        ...
12: #if defined (WITH_IO_CAN)
13:         /* Init CAN I/O */
14:         IO_CAN_Init();
15: #endif
16:    }
17:    ...
18:    return 0;
19: }
```

Reception of CAN messages

Received CAN messages will be handled in the function `IO_In()`. There are two possibilities how to realize that. If there are no other CAN messages to handle, than those, which are handled by the module `IO_CAN`, you can use the function `IO_CAN_RecvLoop()`:

Listing 16.20: Reception of CAN messages: `IO_CAN_RecvLoop()`

```

1: void
2: IO_In(
3:     unsigned CycleNo)
4: {
5:     ...
6:
7:     if (ConnectedIO == IO_None)
8:         return;
9:
10:    if (ConnectedIO == IO_DemoApp) {
11:        ...
12: #if defined (WITH_IO_CAN)
13:         IO_CAN_RecvLoop(Slot.CAN, 0, CycleNo);
14: #endif
15:        ...
16:    }
17: }
```



In Gateway Mode, this is done by the function `IO_CAN_RecvLoopGw()`, which does not require a Slot or Channel number as argument:

Listing 16.21: Reception of CAN messages: `IO_CAN_RecvLoopGw()`

```

1: void
2: IO_In(
3:     unsigned CycleNo)
4: {
5:     ...
6:
7:     if (ConnectedIO == IO_None)
8:         return;
9:
10:    if (ConnectedIO == IO_DemoApp) {
11:        ...
12: #if defined (WITH_IO_CAN)
13:         IO_CAN_RecvLoopGw(CycleNo);
14: #endif
15:        ...
16:    }
17: }
```

Otherwise, you have to use the conventional method, by receiving all available CAN messages in a loop and passing them to the function `IO_CAN_Recv()`:

Listing 16.22: Reception of CAN messages: `IO_CAN_Recv()`

```

1: void
2: IO_In (unsigned CycleNo)
3: {
4:     CAN_Msg     Msg;
5:
6:     ...
7:
8:     if (ConnectedIO == IO_None)
9:         return;
10:
11:    /*** Messages from the FailSafeTester */
12:    FST_MsgIn (CycleNo);
13:
14:    if (ConnectedIO == IO_DemoApp) {
15:        /* read CAN messages from i/o hardware */
16:        #if defined (LYNXOS) || defined(XENO)
17:            while (MIO_M51_Recv(Slot.CAN, 0, &Msg) == 0)
18:        #elif defined (DSPACE)
19:            while (DSIO_CAN_Recv(Slot.CAN, 0, &Msg) == 0)
20:        #endif
21:        {
22:            #if defined (WITH_IO_CAN)
23:                if (IO_CAN_Recv(&Msg, CycleNo) == 0)
24:                    continue;
25:            #endif
26:
27:                switch (Msg.MsgId) {
28:                    ...
29:                }
30:            }
31:        }
32:    }

```

Transmission of CAN messages

CAN messages are usually transmitted in the function `IO_Out()`. That is the place where a call to the function `IO_CAN_Send()` is needed:

Listing 16.23: Transmission of CAN messages

```

1: void
2: IO_Out (
3:     unsigned CycleNo)
4: {
5:     if (ConnectedIO == IO_None)
6:         return;
7:
8:     if (ConnectedIO == IO_DemoApp) {
9:         ...
10:    #if defined (WITH_IO_CAN)
11:        IO_CAN_Send(Slot.CAN, 0, CycleNo);
12:    #endif
13:    }
14: }

```



In Gateway mode, use the function `IO_CAN_SendGw()` instead. This function does not require a Slot or Channel number, but handles internally the association between CAN message and CAN interface:

Listing 16.24: Transmission of CAN messages

```

1: void
2: IO_Out(
3:     unsigned CycleNo)
4: {
5:     if (ConnectedIO == IO_None)
6:         return;
7:
8:     if (ConnectedIO == IO_DemoApp) {
9:         ...
10: #if defined (WITH_IO_CAN)
11:         IO_CAN_SendGw(CycleNo);
12: #endif
13:     }
14: }
```

16.5.2 Modifications to User.c

Having performed the steps above, CarMaker/HIL would now handle a lot of CAN traffic. But, this is quite useless, until you do not make sure, that received CAN signals affect the simulation models, as well as simulation results should be reflected in signals of transmitted CAN messages.

Similar to the previously performed step with `IO.c`, you should first include the header file `IO_CAN_User.h` in `User.c`:

Listing 16.25: Including `IO_CAN_User.h` in `User.c`

```

1: ...
2: #include <CarMaker.h>
3: ...
4:
5: #include "IO.h"
6: #include "User.h"
7:
8: #if defined (LYNXOS) || defined(XENO)
9: # include <mio.h>
10: #elif defined (DSPACE)
11: # include <dsio.h>
12: #endif
13:
14: #if defined (WITH_IO_CAN)
15: # include "IO_CAN_User.h"
16: #endif
17: ...
```

Input processing

Imminently after having gained new values for I/O variables through received CAN messages, an assignment to the appropriate simulation quantities is required. This step is usually performed in the function `IO_CAN_User_In()`, which should be called in `User_In()`:

Listing 16.26: Input processing

```

1: void
2: User_In(
3:     const unsigned CycleNo)
4: {
5:     ...
6:
7: #if defined (CM_HIL)
8:     if (ConnectedIO == IO_DemoApp) {
9: # if defined (WITH_IO_CAN)
10:         IO_CAN_User_In();
11: # endif
12:         ...
13:     }
14:     ...
15: #endif
16:     ...
17: }
```

Output processing

Before being sent to the hardware, the output variables have to be updated with simulation results. This is usually done in the function `IO_CAN_User_Out()`, which should be called in `User_Out()`:

Listing 16.27: Output processing

```

1: void
2: User_Out(
3:     const unsigned CycleNo)
4: {
5:     ...
6:
7: #if defined (CM_HIL)
8:     if (ConnectedIO == IO_DemoApp) {
9: # if defined (WITH_IO_CAN)
10:         IO_CAN_User_Out();
11: # endif
12:         ...
13:     }
14:     ...
15: #endif
16:     ...
17: }
```

16.5.3 Modifications to the Makefile

As the command line for CANiogen may get very long – especially when you make use of the option `-suppress -`, it is recommended to let the generation of `IO_CAN.c` and `IO_CAN.h` get managed by `make`. The modifications to the Makefile depend on the version of `make` and is described below.

GNU make (CarMaker/HIL and CarMaker/DS without Matlab/Simulink)

The easiest way is to use the automatically generated rules for CANiogen. Perform the following steps:

- add object `IO_CAN.o` to the *Makefile* variable `OBJS_CANIOPEN`
- define preprocessor macro `WITH_IO_CAN` by adding `-DWITH_IO_CAN` to the *Makefile* variable `DEF_CFLAGS`
- create the variable `IO_CAN_DB` and assign the path to the CAN database
- create the variable `IO_CAN_FLAGS` to hold the command line options for CANiogen

The *Makefile* should look something like:

Listing 16.28: Defining *Makefile* variables for `IO_CAN` modules (GNU make)

```

1:  ### I/O generator for CANdb files
2:  #
3:  OBJS_CANIOPEN :=      IO_CAN.o
4:  OBJS_CANIOPEN_USER := $(OBJS_CANIOPEN:.o=_User.o)
5:  DEF_CFLAGS +=         -DWITH_IO_CAN
6:  #
7:  # I/O Module IO_CAN
8:  IO_CAN_DB =          ../CANdb/CANIOPENGenericdbc
9:  IO_CAN_FLAGS =       -srvECU ESP
10: #
11: ### END (I/O generator for CANdb files)

```



If you want CANiogen to generate the output in a different directory, you just need to include the path to the object file when adding it to the variable `OBJS_CANIOPEN`:

Listing 16.29: Output to different directory (GNU make)

```

1: ...
2: OBJS_CANIOPEN := generated/IO_CAN.o
3: ...

```

In Listing 16.29 above, the output files `IO_CAN*` will be placed into the directory `generated`. CANiogen will then be automatically called with the option `-outdir generated`, and the compile rule for the object file `generated/IO_CAN.o` will be generated with proper dependencies to `generated/IO_CAN.h`, etcetera.

If you want to use more than one CAN database simultaneously, you simply have to duplicate the *Makefile* variables `IO_CAN_*`, while appending the new object file to the variable `OBJS_CANIOPEN` and appending a suitable preprocessor define to the variable `DEF_CFLAGS`. In order to avoid conflicts, you must also adapt the names of the *Makefile* variables and output files to the different prefixes:

Listing 16.30: Generating multiple `IO_CAN` modules (GNU make)

```

1:  ### I/O generator for CANdb files
2:  #
3:  OBJS_CANIOPEN :=      IO_CAN_ESP.o IO_CAN_Engine.o
4:  DEF_CFLAGS +=         -DWITH_IO_CAN_ESP -DWITH_IO_CAN_Engine
5:  #
6:  # I/O Module IO_CAN_ESP
7:  IO_CAN_ESP_DB =        ../CANdb/CANIOPENGenericdbc
8:  IO_CAN_ESP_FLAGS =     -srvECU ESP
9:  #
10: # I/O Module IO_CAN_Engine
11: IO_CAN_Engine_DB =    ../CANdb/CANIOPENGenericdbc
12: IO_CAN_Engine_FLAGS = -srvECU Engine
13: #
14: ### END (I/O generator for CANdb files)

```



You should never change the definition of the Makefile variable `OBJS_CANIOGEN_USER`. Besides this, it is not necessary to define any additional rule. The rules for the generated files `IO_CAN.c` and `IO_CAN.h` are generated automatically, which is controlled by appending the new object file to `OBJS_CANIOGEN`. The part of the `Makefile`, which creates the rules and dependencies, looks as follows:

Listing 16.31: Automatic generation of rules for CANiogen (GNU make)

```

1: %.depend_caniogen:Makefile
2:         $(QECCHO) " MK      $@"
3:         $Q $(CANIOGEN) -mkrules "$*.o" > $@
4: include $(OBJS_CANIOGEN:.o=.depend_caniogen)

```

If you want to use only static rules – e.g. when using another compiler – you will have to define the following rules for each `IO_CAN` module:

Listing 16.32: Using static rules (GNU make)

```

1: ### CANiogen specific rules
2:
3: # Module IO_CAN_ESP:
4: IO_CAN_ESP.o:          IO_CAN_ESP.c IO_CAN_ESP.h IO_CAN_ESP_User.h
5:         $(CC) $(CFLAGS) $(CANIOGEN_CFLAGS) -o $@ -c $<
6: IO_CAN_ESP_User.o:     IO_CAN_ESP_User.c IO_CAN_ESP_User.h IO_CAN_ESP.h
7:         $(CC) $(CFLAGS) $(CANIOGEN_CFLAGS) -o $@ -c $<
8: IO.o:     IO_CAN_ESP.h
9: User.o:  IO_CAN_ESP_User.h
10:
11: IO_CAN_ESP.c:          IO_CAN_ESP.h
12: IO_CAN_ESP_User.c:    IO_CAN_ESP_User.h IO_CAN_ESP.h
13: IO_CAN_ESP_User.h:    IO_CAN_ESP.h
14: IO_CAN_ESP.h:          $(IO_CAN_ESP_DB) Makefile
15:         $(CANIOGEN) -ioModule IO_CAN_ESP $(IO_CAN_ESP_FLAGS) $<
16:
17: # Module IO_CAN_Engine:
18: IO_CAN_Engine.o:       IO_CAN_Engine.c IO_CAN_Engine.h IO_CAN_Engine_User.h
19:         $(CC) $(CFLAGS) $(CANIOGEN_CFLAGS) -o $@ -c $<
20: IO_CAN_Engine_User.o: IO_CAN_Engine_User.c IO_CAN_Engine_User.h IO_CAN_Engine.h
21:         $(CC) $(CFLAGS) $(CANIOGEN_CFLAGS) -o $@ -c $<
22: IO.o:     IO_CAN_Engine.h
23: User.o:  IO_CAN_Engine_User.h
24:
25: IO_CAN_Engine.c:       IO_CAN_Engine.h
26: IO_CAN_Engine_User.c: IO_CAN_Engine_User.h IO_CAN_Engine.h
27: IO_CAN_Engine_User.h: IO_CAN_Engine.h
28: IO_CAN_Engine.h:       $(IO_CAN_Engine_DB) Makefile
29:         $(CANIOGEN) -ioModule IO_CAN_Engine $(IO_CAN_Engine_FLAGS) $<
30:
31: ### END (CANiogen specific rules)

```

Opus make (CarMaker/DS with Matlab/Simulink)

The easiest way is to use the automatically generated rules for CANiogen. Perform the following steps to the user makefile `<ModelName>_usr.mk`:

- add source file `IO_CAN.c` to the `Makefile` variable `SRCS_CANIOGEN`
- define preprocessor macro `WITH_IO_CAN` by adding `-DWITH_IO_CAN` to the `Makefile` variable `CM_SRC_CFLAGS`
- create the variable `IO_CAN_DB` and assign the path to the CAN database
- create the variable `IO_CAN_FLAGS` to hold the command line options for CANiogen

The makefile `<ModelName>_usr.mk` should look something like:

Listing 16.33: Defining *Makefile* variables for IO_CAN modules (Opus make)

```

1:  ### I/O generator for CANdb files
2:  #
3:  SRCS_CANIOGEN :=          IO_CAN.c
4:  SRCS_CANIOGEN_USER :=    $(SRCS_CANIOGEN:.c=_User.c)
5:  CM_SRC_CFLAGS +=         -DWITH_IO_CAN
6:  #
7:  # I/O Module IO_CAN
8:  IO_CAN_DB =      ../../CANdb/CANIogenGenericdbc
9:  IO_CAN_FLAGS =   -srvECU ESP
10: #
11: ...
12: #
13: ### END (I/O generator for CANdb files)

```

If you want to use more than one CAN database simultaneously, you simply have to duplicate the Makefile variables `IO_CAN_*`, while appending the new source file to the variable `SRCS_CANIOGEN` and appending a suitable preprocessor define to the variable `CM_SRC_CFLAGS`. In order to avoid conflicts, there must also be adapted the names of the *Makefile* variables and output files to different prefixes:

Listing 16.34: Generating multiple IO_CAN modules (Opus make)

```

1:  ### I/O generator for CANdb files
2:  #
3:  SRCS_CANIOGEN :=          IO_CAN_ESP.c IO_CAN_Engine.c
4:  CM_SRC_CFLAGS +=         -DWITH_IO_CAN_ESP -DWITH_IO_CAN_Engine
5:  #
6:  # I/O Module IO_CAN_ESP
7:  IO_CAN_ESP_DB =      ../../CANdb/CANIogenGenericdbc
8:  IO_CAN_ESP_FLAGS =   -srvECU ESP
9:  #
10: # I/O Module IO_CAN_Engine
11: IO_CAN_Engine_DB =      ../../CANdb/CANIogenGenericdbc
12: IO_CAN_Engine_FLAGS =   -srvECU Engine
13: #
14: ### END (I/O generator for CANdb files)

```



It is not necessary to define any additional rule. The rules for the generated files *IO_CAN.c* and *IO_CAN.h* are generated automatically. This is controlled by appending the new source file to *SRCS_CANIOGEN*. The part of the *Makefile*, which creates the rules, looks as followed:

Listing 16.35: Automatic generation of rules for CANiogen (Opus make)

```
1:  ### I/O generator for CANdb files
2:  #
3:  ...
4:  #
5:  # automatically generated rules for CANiogen:
6:  # =====
7:  %foreach CMOD $(SRCS_CANIOGEN,.c=)
8:  $(CMOD).$(CC_OBJ_EXT):           $(CM_SRC_DIR)/$(CMOD).c $(CM_SRC_DIR)/$(CMOD).h \
9:                                $(CM_SRC_DIR)/$(CMOD)_User.h
10:     %echo COMPILING $<
11:     --$(CC) -c $(CC_FLAGS) $(CANIOGEN_CFLAGS) $(CM_SRC_CFLAGS) $<
12:     $(CMOD)_User.$(CC_OBJ_EXT):    $(CM_SRC_DIR)/$(CMOD)_User.c \
13:                                $(CM_SRC_DIR)/$(CMOD)_User.h \
14:                                $(CM_SRC_DIR)/$(CMOD).h
15:     %echo COMPILING $<
16:     --$(CC) -c $(CC_FLAGS) $(CANIOGEN_CFLAGS) $(CM_SRC_CFLAGS) $<
17:     IO.$(CC_OBJ_EXT):             $(CM_SRC_DIR)/$(CMOD).h
18:     User.$(CC_OBJ_EXT):          $(CM_SRC_DIR)/$(CMOD).h \
19:                                $(CM_SRC_DIR)/$(CMOD)_User.h
20:
21:     $(CM_SRC_DIR)/$(CMOD).c:      $(CM_SRC_DIR)/$(CMOD).h
22:     $(CM_SRC_DIR)/$(CMOD)_User.c: $(CM_SRC_DIR)/$(CMOD)_User.h $(CM_SRC_DIR)/$(CMOD).h
23:     $(CM_SRC_DIR)/$(CMOD)_User.h: $(CM_SRC_DIR)/$(CMOD).h
24:     $(CM_SRC_DIR)/$(CMOD).h:      $( $(CMOD)_DB) $(WORKDIR)$ (MODEL)_usr.mk
25:     %echo GENERATING $(@,.h=.c) and $@
26:     --$(CANIOGEN) -ioModule $(CMOD) $( $(CMOD)_FLAGS) -outdir $(CM_SRC_DIR) \
27:                                $( $(CMOD)_DB)
28: %end
29: #
30: ### END (CANiogen specific rules)
```

If you want to use only static rules – e.g. when using another compiler – you will have to define the following rules for each IO_CAN module:

Listing 16.36: Using static rules (Opus make)

```

1:  ### CANiogen specific rules
2:
3:  # Module IO_CAN_ESP:
4:  IO_CAN_ESP.$(CC_OBJ_EXT):      $(CM_SRC_DIR)/IO_CAN_ESP.c \
5:                                $(CM_SRC_DIR)/IO_CAN_ESP.h \
6:                                $(CM_SRC_DIR)/IO_CAN_ESP_User.h
7:  %echo COMPILING $<
8:  --$(CC) -c $(CC_FLAGS) $(CANIOGEN_CFLAGS) $(CM_SRC_CFLAGS) $<
9:  IO_CAN_ESP_User.$(CC_OBJ_EXT): $(CM_SRC_DIR)/IO_CAN_ESP_User.c \
10:                                $(CM_SRC_DIR)/IO_CAN_ESP_User.h \
11:                                $(CM_SRC_DIR)/IO_CAN_ESP.h
12:  %echo COMPILING $<
13:  --$(CC) -c $(CC_FLAGS) $(CANIOGEN_CFLAGS) $(CM_SRC_CFLAGS) $<
14:  IO.$(CC_OBJ_EXT):            $(CM_SRC_DIR)/IO_CAN_ESP.h
15:  User.$(CC_OBJ_EXT):         $(CM_SRC_DIR)/IO_CAN_ESP.h \
16:                                $(CM_SRC_DIR)/IO_CAN_ESP_User.h
17:
18:  $(CM_SRC_DIR)/IO_CAN_ESP.c:   $(CM_SRC_DIR)/IO_CAN_ESP.h
19:  $(CM_SRC_DIR)/IO_CAN_ESP_User.c: $(CM_SRC_DIR)/IO_CAN_ESP_User.h \
20:                                $(CM_SRC_DIR)/IO_CAN_ESP.h
21:  $(CM_SRC_DIR)/IO_CAN_ESP_User.h: $(CM_SRC_DIR)/IO_CAN_ESP.h
22:  $(CM_SRC_DIR)/IO_CAN_ESP.h:    $(IO_CAN_ESP_DB) $(WORKDIR) $(MODEL)_usr.mk
23:  %echo GENERATING $(@,.h=.c) and $@
24:  --$(CANIOGEN) -ioModule IO_CAN_ESP $(IO_CAN_ESP_FLAGS) \
25:      -outdir $(CM_SRC_DIR) $(IO_CAN_ESP_DB)
26:
27:  # Module IO_CAN_Engine:
28:  IO_CAN_Engine.$(CC_OBJ_EXT): $(CM_SRC_DIR)/IO_CAN_Engine.c \
29:                                $(CM_SRC_DIR)/IO_CAN_Engine.h \
30:                                $(CM_SRC_DIR)/IO_CAN_Engine_User.h
31:  %echo COMPILING $<
32:  --$(CC) -c $(CC_FLAGS) $(CANIOGEN_CFLAGS) $(CM_SRC_CFLAGS) $<
33:  IO_CAN_Engine_User.$(CC_OBJ_EXT): $(CM_SRC_DIR)/IO_CAN_Engine_User.c \
34:                                $(CM_SRC_DIR)/IO_CAN_Engine_User.h \
35:                                $(CM_SRC_DIR)/IO_CAN_Engine.h
36:  %echo COMPILING $<
37:  --$(CC) -c $(CC_FLAGS) $(CANIOGEN_CFLAGS) $(CM_SRC_CFLAGS) $<
38:  IO.$(CC_OBJ_EXT):            $(CM_SRC_DIR)/IO_CAN_Engine.h
39:  User.$(CC_OBJ_EXT):         $(CM_SRC_DIR)/IO_CAN_Engine.h \
40:                                $(CM_SRC_DIR)/IO_CAN_Engine_User.h
41:
42:  $(CM_SRC_DIR)/IO_CAN_Engine.c: $(CM_SRC_DIR)/IO_CAN_Engine.h
43:  $(CM_SRC_DIR)/IO_CAN_Engine_User.c: $(CM_SRC_DIR)/IO_CAN_Engine_User.h \
44:                                $(CM_SRC_DIR)/IO_CAN_Engine.h
45:  $(CM_SRC_DIR)/IO_CAN_Engine_User.h: $(CM_SRC_DIR)/IO_CAN_Engine.h
46:  $(CM_SRC_DIR)/IO_CAN_Engine.h:   $(IO_CAN_Engine_DB) $(WORKDIR) $(MODEL)_usr.mk
47:  %echo GENERATING $(@,.h=.c) and $@
48:  --$(CANIOGEN) -ioModule IO_CAN_Engine $(IO_CAN_Engine_FLAGS) \
49:      -outdir $(CM_SRC_DIR) $(IO_CAN_Engine_DB)
50:
51:  ### END (CANiogen specific rules)

```

16.6 Integration into CarMaker/HIL (J1939 Mode)

This section applies only to CANiogen, if used in the J1939 mode.

As described in the sections above the output of CANiogen in this mode will create two parameter files the *J1939Parameter* and *J1939Mappings.default*.

The integration into a CarMaker/HIL project, is done in four steps. First, in order to optimize the automatic build process, you may then want to add some new rules to the *Makefile*. After that you will have to modify slightly the existing C module files *IO.h* and *IO.c* to enable the CAN I/O part.

Finally to complete the integration process you have to map the CAN I/O signals with the quantities in CarMaker_HIL, which is done in the *J1939Mapping.default* file. When you open this mapping file you will find an example for a correct mapping of the I/O Parameters with CarMaker_HIL quantities. After that you should rename that file to ensure that it is not overwritten during another generation process. The necessary modifications of the other files will be demonstrated in the following sections.

16.6.1 Modifications to the Makefile

As the command line for CANiogen may get very long, it is recommended to let the generation of the above mentioned parameter files get managed by the command *make*.

At first the J1939 specific library file *libj1939.a*, located in the CarMaker/HIL installation directory, has to be integrated to the building process. It includes all the necessary functions for implementing CAN I/O in J1939 mode. You also may copy this library-file to the *lib-folder* as well as the depending header-file *j1939.h* to the *include-folder* of your current project directory.

Listing 16.37: Integration of the J1939 library file may looks like this

```

1: LD_LIBS = (LIB_J1939)
2:
3: # Expand include path by local include/library directory:
4: INC_CFLAGS += -I../include -I../lib/${ARCH} -I../lib
5:
6: ## J1939
7: LIB_J1939 = ../lib/${ARCH}/libj1939.a
8: DEF_CFLAGS += -DWITH_1939

```

Furthermore specific J1939 rules and a *new make command* called *j1939* should be defined. If you just want to generate new J1939 Parameter files, you can use the command extension *make j1939*. This will not build a new CarMaker/HIL executable file!.

Listing 16.38: Define CANiogen J1939 specific rules

```

9: J1939_DB = ../CANbd/myCANDataBAsEdbc
10: J1939_PARAM = J1939Parameters.MyName
11: J1939_OD = ../Data/Config /*Output parameter files in this folder*/
12: J1939_FLAGS = -outfmt j1939 -srvECU MyECU -outdir $(J1939_OD)
13:
14: default: $(APP_NAME) j1939
15: ...
16:
17: ## J1939 rules
18: j1939: $(J1939_OD)/$(J1939_PARAM)
19:
20: $(J1939_OD)/$(J1939_PARAM): $(J1939_DB)
21:     $(CANIOGEN) $(J1939_FLAGS) \
22:         -outfile $(J1939_PARAM)
23: ## END (J1939 rules)

```

16.6.2 Modifications to IO.h

In *IO.h* the Definition of the specific *J1939_Handle** is done. It is needed for using the J1939 mode depending functions later on in the *IO.c* file.i

Listing 16.39: Adding *J1939_Handle** to *IO.h* looks like

```
1: #if !defined(_J1939_H_)
2: typedef struct J1939 *J1939_Handle;
3: #endif
4:
5: ...
6:
7: /**+ Input Vector, signals from hardware, ...*/
8: typedef struct {
9:
10: ...
11:
12: #if defined(WITH_J1939)
13: J1939_Handle J1939_MyName;
14: #endif
15: } tIOVec;
```

16.6.3 Modifications to IO.c

In the *IO.c* the communication with the Inputs and Outputs is carried out, in this case also the initialization, receiving and transmission via CAN bus.

Insert the J1939 functions into the default I/O-hook functions in *IO.c* like *IO_Init* etc.

Listing 16.40: Adding *J1939_Init*- function in *IO.c* looks like

```
1: #if defined(WITH_J1939)
2: #include "j1939.h"
3: #endif
4:
5: ...
6:
7: int IO_Init (void)
8: {
9:
10: ...
11:
12: #if defined(WITH_J1939)
13: /* initialize J1939 communication */
14: J1939_Init(&MyIO.J1939_MyName, Slot_CAN, 0, MIO_M51_Send):
15: J1939_Parm_Get(IO.J1939_MyName, SimCore.ECUParm.Inf, "MyName");
16: J1939_DeclQuants(IO.J1939_MyName, "MyPrefix");
17: #endif
18: }
19: ...
20:
21: int IO_Init_Finalize (void)
22: {
23: #if defined(WITH_J1939)
24: if (J1939_MapQuants(IO.J1939_MyName) < 0)
25: return -1;
26: #endif
27: return 0;
28: }
29:
```

For receiving or transmitting signals the appropriate J1939 functions has also be implemented to the depending hook functions in */O.c*.

Listing 16.41: Adding functions for communication in J1939 mode in */O.c* looks like

```

30:
31: ...
32:
33: void IO_In (unsigned CycleNo)
34: {
35:
36: ...
37:
38: #if defined(WITH_J1939)
39: while (1) {
40: if (MIO_M51_Recv(Slot_CAN, 0, &Msg) != 0)
41: break;
42: if (J1939_Recv_CAN(IO.J1939_MyName, &Msg) >= 0)
43: continue;
44: }
45: J1939_In(IO.J1939_MyName, CycleNo);
46: #endif
47: }
48:
49: void IO_Out (unsigned CycleNo)
50: {
51:
52: ...
53:
54: #if defined(WITH_J1939)
55: J1939_Out(IO.J1939_MyName, CycleNo);
56: #endif
57: }
58:
59: void IO_Cleanup (void)
60: {
61: #if defined(WITH_J1939)
62: J1939_Cleanup(IO.J1939_MyName);
63: #endif
64: }
```

16.7 Using databases with CAN FD messages

Newer versions of Vector CAN databases support messages with CAN FD protocol. There are some differences in CANiogen's generated C-Code, which will be shown up here.

First of all, CANiogen scans the database and tries to find the parameter “BusType”, which indicates whether CAN FD messages are within the database or not. Based on this information, CANiogen produces a slightly different output. Since the M51 hardware does not support the CAN FD protocol, receiving and transmitting CAN FD messages have to be handled with the help of the M410 functions. Therefore, the functions `MIO_M51_Send()` / `MIO_M51_Recv()` will be replaced by `MIO_M410_Send()` / `MIO_M410_Recv()`. The function calls to these M410 functions need another CAN message type. For this purpose, a new struct called `CANFD_Msg` is introduced, which replaces all `CAN_Msg` struct variables in the whole C-Code output. Standard CAN messages are still supported, but they also use the `CANFD_Msg` struct (the struct element `FDF` indicates the message type FD or standard).

Standard CAN databases (without CAN FD messages) induce CANiogen to build the output with the M51 functions and the `CAN_Msg` struct. In case of an installed M410 module, these functions will be used in a M410 compatibility mode. If the user want to switch to the M410 functions and the `CANFD_Msg` struct instead, the command line argument `-m410` forces CANiogen to do so.

16.8 Version History

Version	Changes
1.0	Initial version
1.0.2	Added Support for CAN databases, without information about cycle time of Tx messages
1.0.3	Support for suppressed Data Dictionary entries added
1.0.4	Bugs fixed: <ul style="list-style-type: none">• undefined variable <code>bitfield</code>, if signal is not of type INT
1.0.5	Bugs fixed: <ul style="list-style-type: none">• signals with same name in different CAN messages not imported correctly• extended CAN messages not handled correctly
1.1.0	Improvements: <ul style="list-style-type: none">• creation of <code>IO_CAN_VarList.txt</code> which lists all I/O variables Bugs fixed: <ul style="list-style-type: none">• problems with extended CAN messages• problems with signed/unsigned integers
1.1.1pre1	Improvements: <ul style="list-style-type: none">• option <code>-suppress</code> matches also CAN messages• unneeded information will be omitted in generated C code• consistency check and workarounds for CAN databases• added support for default values of CAN signals• additional output file generated with list of I/O variables• unknown ECUs, signals and messages will be reported• added comment with message id in hex to header file Bugs fixed: <ul style="list-style-type: none">• CAN messages, that are transferred between virtual ECUs were generated and handled, which led to unneeded code<ul style="list-style-type: none">-> omitting all messages and signals, that are not received or transmitted by real ECUs• duplicate CAN message names led to Tcl error<ul style="list-style-type: none">-> added consistency check to generate unique message names

Version	Changes
1.1.1pre2	<p>Improvements:</p> <ul style="list-style-type: none"> option <code>-srvECU</code>, that allows to give a list of real ECUs which should be served by CarMaker/HIL (this option cannot be used together with the <code>-simECU</code> option) option <code>-excECU</code> to exclude ECUs from I/O generated code option <code>-sndMsg</code> to explicitly send CAN messages <p>Bugs fixed:</p> <ul style="list-style-type: none"> Tcl error occurred when using <code>-rcvSig</code> option received CAN signals were not correctly handled, when signal was multiplexed and multiplexer missing in receive list -> add multiplexer to <code>ioSigList</code>, if needed in some cases, wrong data type was chosen for I/O variable. Minimum value and maximum value were not computed correctly -> compute minimum an maximum value with inclusion of factor and offset to get data type for I/O variable
1.1.1pre3	<p>Improvements:</p> <ul style="list-style-type: none"> <code>-ignore nCyclic</code> option to suppress sending of messages that are not marked as cyclic <code>-msgSCAv</code> option to give the attribute value, that marks a message as cyclic
1.1.1pre4	<p>Fixed problem with Signals in Intel Byte Order:</p> <ul style="list-style-type: none"> signals, which start at a bit position != %8 and ending in the same data byte, were not correctly handled max value of signal was not computed correctly added Test for length of signal
1.1.1	<p>Bugs fixed:</p> <ul style="list-style-type: none"> signal with data length < 8 bit was not encoded/decoded correctly
1.1.2	<p>Bugs fixed:</p> <ul style="list-style-type: none"> in function <code>IO_CAN_Recv(): Msg</code> and <code>MsgExt</code> were mixed, data was copied in wrong direction
1.1.3	<p>Bugs fixed:</p> <ul style="list-style-type: none"> When reading signal from CAN message or writing signal to CAN message, the offset was not added if scaling factor was 1 Wrong range checking for signals with data type <code>int</code>
1.1.4	<p>Improvements:</p> <ul style="list-style-type: none"> New option <code>-noRngChk</code> to disable automatic range checking for signals Consistency check: warning if min value of signal is <0, but signal has unsigned data type
1.1.5	<p>Bugs fixed:</p> <ul style="list-style-type: none"> Offset for <code>*_GetV()</code> macro of signal printed to header file even if <code>*_GetV()</code> macro not needed -> Syntax Error during compilation

Version	Changes
1.1.6	Bugs fixed: <ul style="list-style-type: none"> Offset for <code>*_GetV()</code> macro of signal printed to header file even if <code>*_GetV()</code> macro not needed
1.1.7	Bugs fixed: <ul style="list-style-type: none"> Missing brakes in <code>*_GetV()</code> macros -> Warnings during compilation
1.1.8	Bugs fixed: <ul style="list-style-type: none"> wrong data type for declared I/O variables due to error in min/max value computation
1.2	Improvements: <ul style="list-style-type: none"> Automatic distribution of periodic CAN messages DVA access to timing parameters Timing variables listed in <code>VarList.txt</code> Message length of Tx messages listed in Data Dictionary and <code>VarList.txt</code> (allows to be modified by DVA) Generated code ported to dSPACE: CANiogen can now be used with CarMaker/DS Sorted output of ECUs, messages and signals to generated files All generated I/O data structures are now properly named and can be referenced in other modules Bugs fixed: <ul style="list-style-type: none"> Tcl error while importing signals with more than 32 bits
1.2.2	Improvements: <ul style="list-style-type: none"> Suitable Defaults for <code>*_CAN_Send</code> function pointers (LYNXOS, DSPACE) <code>ReadyToSend()</code> function for all Tx CAN messages Rx- and Tx Hook functions allow access to raw CAN messages. Thus, the user has direct influence on which CAN messages are sent or received Output of program identification improved Bugs fixed: <ul style="list-style-type: none"> Spelling error in generated header file fixed (<code>*_CAN_Send()</code>)
1.2.3	Improvements: <ul style="list-style-type: none"> New option <code>-outdir</code> allows to define output directory for generated files Bugs fixed: <ul style="list-style-type: none"> When using <code>-suppress</code> option with only signal name as pattern, the pattern was ignored instead of testing with all CAN messages
1.2.4	Improvements: <ul style="list-style-type: none"> Automatic determination of <code>SendDistrib</code> improved for Tx CAN messages

Version	Changes
1.2.5	<p>Improvements:</p> <ul style="list-style-type: none"> Extended option <code>-suppress</code> allows now to suppress all generated variables of complete CAN messages, including timing variables and data length counter <p>Bugs Fixed:</p> <ul style="list-style-type: none"> Problems with white spaces and other special characters in file name
1.2.6	Minor fixes in usage (spelling, etc.)
1.2.7	<p>Improvements:</p> <ul style="list-style-type: none"> Basic support for dSPACE Processor Board DS1005 (CAN signals with size of not more than 32 Bits) <p>Bugs Fixed:</p> <ul style="list-style-type: none"> CANdb not correctly parsed, if last signal in msg with highest id is of type float Wrong cast to float and double when decoding and encoding signals of type float Wrong output of command line in <code>IO_CAN_VarList.txt</code> Missing tcl dll (on Windows version only) Tcl error ("can't read "msgid": no such variable ...") when using CANdb with extended CAN messages
1.3pre1	<p>Improvements:</p> <ul style="list-style-type: none"> Send function for single CAN messages Option <code>-ddPrefix</code> allows to define (shorter) prefix for data dictionary entries Option <code>-ddMsgId</code> to define data dictionary entries with msg id instead of message name Create macros for default/start value of signals and timing variables -> makes code/initialization of variables more clear Automatic generation of <code>IO_CAN_User.c</code> (<code>IO_CAN_User_In()</code>, <code>IO_CAN_User_Out()</code>, <code>IO_CAN_User_RxHook()</code>, <code>IO_CAN_User_RxHook()</code>, <code>IO_CAN_User_Init_First()</code>, <code>IO_CAN_User_Init()</code>) RollCount() function in <code>IO_CAN_User.c</code> Information / Warning message for user, if changes in CANdb are detected, which result in changed variables: <ul style="list-style-type: none"> - ECU new, removed - CAN message new, removed - CAN signal new, removed - CAN signal changed (data type, variable name) Name structure data dictionary entries of timing variables uses same <p>Bugs Fixed:</p> <ul style="list-style-type: none"> Name structure data dictionary entries of timing variables uses same scheme as normal signals (<code>IO_CAN_Timings.msgname.SendPeriod</code>) -> ECU name removed Generated header file <code>IO_CAN.h</code> defines macros which refer to static variables and macros generated in <code>IO_CAN.c</code> -> variables now declared in header file, not static anymore

Version	Changes
1.3pre2	<p>Improvements:</p> <ul style="list-style-type: none"> Switch <code>DeclareQuants</code> in <code>IO_CAN</code> and <code>IO_CAN_Timings</code> data structure to globally enable/disable declaration of quantities <p>Bugs Fixed:</p> <ul style="list-style-type: none"> Command line option <code>-suppress</code> did not affect generation of Data Dictionary entries for timing variables and for CAN message length Format of generated code Character Encoding cp1252 added for better Windows compatibility
1.3pre3	<p>Improvements:</p> <ul style="list-style-type: none"> Hook functions now called twice for better possibilities of intervention <ul style="list-style-type: none"> - before encoding / decoding of signals - after encoding / decoding of signals <p>Note: Hook functions require additional argument (int Part)</p> <pre>int (*IO_CAN_RxHook) (struct CAN_Msg *Msg, const unsigned CycleNo, int Part); int (*IO_CAN_TxHook) (struct CAN_Msg *Msg, const unsigned CycleNo, int Part);</pre> <p>Bugs Fixed:</p> <ul style="list-style-type: none"> In some circumstances CANiogen produced unneeded code, if CAN database contained duplicate signal names
1.3	<p>Improvements:</p> <ul style="list-style-type: none"> CANiogen automatically sources <code>.CANiogen.tcl</code>, if found in current directory New command line option <code>-prefs</code> allows to pass name of preferences file. This (Tcl-)file will be sourced Support for unit map table for Data Dictionary quantities: Tcl array <code>Unit_Map</code> overrides default unit mapping table, if specified in preferences file <code>.CANiogen.tcl</code> or with <code>-prefs</code> command line option
1.3.1	<p>Bugs Fixed:</p> <ul style="list-style-type: none"> Tcl Error in Debug message fixed Missing return in switch-case command in function <code>IO_CAN_SendMsg()</code> could cause transmission of CAN messages too often, if return value of function <code>IO_CAN_User_TxHook()</code> is < 0
1.3.2	<p>Improvements:</p> <ul style="list-style-type: none"> Special command line switch <code>-mkrules</code> to generate Makefile rules
1.3.3	<p>Bugs Fixed:</p> <ul style="list-style-type: none"> Erroneous conversion of float / double signals to integer value in <code>*_GetV()</code> macro Message variable <code>*_DLC</code> not automatically reset to default value after being manipulated with DVA Compiler warning <i>dereferencing type-punned pointer will break strict-aliasing rules</i> when compiling <code>*_SetV()</code> macro for float and double signals

Version	Changes
1.3.4	<p>Improvements:</p> <ul style="list-style-type: none"> Gateway mode for serving different ECUs on several CAN busses <p>Bugs Fixed:</p> <ul style="list-style-type: none"> Tcl Error “invalid command name “GatewayMode”” Endless loop during “check for changes since last run”
1.3.5	<p>Improvements:</p> <ul style="list-style-type: none"> Always write <code>IO_CAN_User.c.new</code> and <code>IO_CAN_User.h.new</code>, if User module files already exist -> user can now easily check, if User module needs update New options <code>-noRngChkRx</code> and <code>-noRngChkTx</code> to disable range checking for Tx and Rx messages separately Example CRC functions <code>GetCRC4()</code> (4-Bit CRC) and <code>GetCRC_J1850()</code> (according to SAE J1850) added to <code>IO_CAN_User.c</code> New User function <code>IO_CAN_User_CAN_SendGw()</code> for distribution of CAN messages to CAN busses of gatewayed ECUs: <ul style="list-style-type: none"> receives id of sender and CAN message decides on which CAN bus CAN message needs to be sent distribution fully controlled by user <p>Bugs Fixed:</p> <ul style="list-style-type: none"> <code>IO_CAN_SendMsgGw()</code>: Send Error on 1 bus results in unsent message on all following busses <code>IO_CAN_User_Out()</code>: Signal values are always reset to start value by default, even though if message is gatewayed to other ECU(s), only Problems with <code>MsgId</code> on extended CAN messages in <code>IO_CAN_User.c</code>
1.3.6	<p>Bugs Fixed:</p> <ul style="list-style-type: none"> Endless loop during “check for changes since last run” Tcl Error during “check for changes since last run” with extended CAN messages in CANdb
1.3.7	<p>Bugs Fixed:</p> <ul style="list-style-type: none"> Wrong Decoding of Float and Double Signals (interpreted as int) Wrong Encoding of Float and Double Signals (caused by Compiler Optimization) Definition of attributes of type STRING (<code>BA_DEF_ "" STRING;</code>) caused CANiogen to abort with Tcl error “error: invalid command name “<code>BA_DEF_</code>””

Version	Changes
1.4	<p>Improvements:</p> <ul style="list-style-type: none"> New Option <code>-ignore ignSigs</code> allows to optimize out signals without receiver State variables for all CAN signals Raw variables for all CAN signals Support for Invalid values of signals Option <code>-sigInvT</code> allows to specify text which qualifies signal to be interpreted as invalid <p>Bugs Fixed:</p> <ul style="list-style-type: none"> Signals without receiver were always ignored, even if part of Rx- or Tx- message CRC computed by generated CRC functions <code>GetCRC4()</code> / <code>GetCRC8()</code> was incorrect
1.4.1	<p>Bugs Fixed:</p> <ul style="list-style-type: none"> Error in format of <code>IO_CAN_VarList.txt</code> caused endless loop during check for changes since last run Rounding error when converting signal value from physical to raw value
1.4.2	<p>Improvements:</p> <ul style="list-style-type: none"> Support for <code>error</code>, <code>undefined</code> and <code>unavailable</code> state of signals Options <code>-sigErrT</code>, <code>-sigUDefT</code> and <code>-sigUAvlT</code> allow to specify text to qualify a signal value as error, undefined or unavailable When using the option <code>-ddMsgId</code>, the format of the message id can be customized by setting the variable <code>Imp(ddMsgIdFmt)</code> in the preferences file (<code>-prefs prefs.tcl</code>). The default is "%03xh" <p>Bugs Fixed:</p> <ul style="list-style-type: none"> User settings file <code>.CANiogen.tcl</code> was read too early. This caused a different behavior than with the alternative way by using the option <code>-prefs</code>
2.0	<p>Improvements:</p> <ul style="list-style-type: none"> Support for J1939 <p>Bugs Fixed:</p> <ul style="list-style-type: none"> Attribute values of ECUs were lost (CANdb::NodeGetAttrVal) Attribute values of ECUs not imported correctly (assigned to wrong ECU) Compiler warning due to missing prototype for function <code>round()</code>
2.0.1	<p>Bugs Fixed:</p> <ul style="list-style-type: none"> Special J1939 Frames, containing no real signals, were not ignored Generated J1939Parameters correctly (re-)imported, but outdated information was not deleted
2.0.2	<p>Bugs Fixed:</p> <ul style="list-style-type: none"> Default Mapping of signals in <code>J1939Mappings.default</code> was non-existing Data Dictionary variable (changed to <code>Const 0</code>)

Version	Changes
2.0.3	<p>Bugs Fixed:</p> <ul style="list-style-type: none"> Tcl Error “missing close-bracket” when using <code>-outdir</code> option and output directory does not exist When specifying object file names containing a directory component or using command line option <code>-outdir</code>, the created Makefile rules (with <code>-mkrules</code> option) were invalid
2.0.4	<p>Improvements:</p> <ul style="list-style-type: none"> Description of ECU / PDU / SPN parameter format in J1939Parameters Default PDU and SPN definitions from Spec for incomplete CANdb files <p>Bugs Fixed:</p> <ul style="list-style-type: none"> Generated Makefile rules with errors, if path to CANdb file contains blanks
2.0.5	<p>Bugs Fixed:</p> <ul style="list-style-type: none"> Compiler Warnings "comparison is always true due to limited range of data type" Compiler Warnings "integer constant is too large for 'long' type" Compiler Warnings "integer constant is so large that it is unsigned" Raw values of some signals were signed instead of unsigned, caused by invalid change of signal type during consistency check
2.0.6	<p>Improvements:</p> <ul style="list-style-type: none"> User can now specify his own configuration file for default PDU- and SPN definitions <p>Bugs Fixed:</p> <ul style="list-style-type: none"> PDU Timings are checked more fault-tolerant, when extracting from PDU name, and are automatically updated with default settings from PDU definition file
2.0.7	<p>Improvements:</p> <ul style="list-style-type: none"> Quiet mode with reduced informational output for use with Makefile
2.1	<p>The generated C-code was adapted to CarMaker 4.0, in order to meet the new Data Dictionary API</p>
2.1.1	<p>Bugs Fixed:</p> <ul style="list-style-type: none"> No support for integer signals with more than 32 bits on DS1005 The consistency check of CANiogen failed with the error message ‘error: can't read “nMsg(***)”: no such element in array’, if the name of one or more CAN signal started with “ECU”

Version	Changes
2.2	<p>Improvements:</p> <ul style="list-style-type: none"> Import from FIBEX data base <p>Bugs Fixed:</p> <ul style="list-style-type: none"> Multiple definitions of variables, preprocessor macros and functions, if CAN message has more than one sender Tcl Error "error: expected integer but got "" if attribute SPN missing (J1939 mode only) Corrupted output, if J1939 attributes not found in CANdb. Fixed with warnings and default values as replacement
2.2.1	<p>Bugs Fixed:</p> <ul style="list-style-type: none"> When importing CAN data bases with CAN messages having more than one sender, the CAN messages were only assigned to the last sender
2.2.2	<p>The generated C-code was ported to CarMaker for LabVIEW</p> <p>BugsFixed:</p> <ul style="list-style-type: none"> Tcl Error "error: expected integer but got "" if attribute SPN missing (J1939 mode only) Corrupted output, if J1939 attributes not found in CAN data base. Missing J1939 attributes are reported with warnings, default values are used as replacement
2.3	<p>Improvements:</p> <ul style="list-style-type: none"> CAN FD Support Eclipse Support (Command line argument -eclipse added) Command line argument -m410 added
2.3.1	<p>Bugs Fixed:</p> <ul style="list-style-type: none"> Wrong signal values caused by implicit type casts between double / float and integer (IO_CAN_FixMin() / IO_CAN_FixMax()) Compiler warnings “converting ‘long long unsigned int’ from ‘double’”
2.3.2	<p>Bugs Fixed:</p> <ul style="list-style-type: none"> Wrong data type for signals without minimum and maximum [0 0] Compiler warnings "condition always true/false due to limited range of data type"
2.3.5	<p>Bugs Fixed:</p> <ul style="list-style-type: none"> Generated Code did not compile without errors

Version	Changes
2.4	<p>Improvements:</p> <ul style="list-style-type: none">• Support for FIBEX Version 4 (CAN)• Function parameter MsgId splitted into MsgId & IDE for IO_CAN_TxHookData / IO_CAN_RxHookData. <p>Bugs Fixed:</p> <ul style="list-style-type: none">• Bugfix: A Tx message was not discarded in IO_CAN_SendMsg, if requested by the user definable hook function (negative return value of IO_CAN_User_TxHook or IO_CAN_User_TxHookFD).• Bugfix: Dummy functions added for dSpace because of missing CAN FD functions.

Chapter 17

ScriptControl

17.1 Introduction

ScriptControl is a tool used to create, manage and run scripts that automate the testing process of the CarMaker simulation environment. For example, scripts can be created to load a predefined TestRun, start the TestRun with a change to some input parameters, evaluate the results and write a report summary to a log file.

All of the functionality of the CarMaker Interface Toolkit (CIT)¹ can be automated with ScriptControl. In addition, ScriptControl can interface directly to the Virtual Vehicle Environment (VVE)² of CarMaker allowing user defined scripts to add functionality that is not contained in the CIT. This facilitates the creation of very useful and powerful scripts that can completely automate the testing process by, e.g., executing an unlimited number of predefined TestRuns (ISO lane-change maneuver, braking maneuver, etc.) filtering and logging the results, performing the same tests with changes to some of the input parameters, comparing the results, and so on.

The ScriptControl scripting language is based on Tcl/Tk, an open-source, freely available, and well documented programming language. The Tcl/Tk language is robust and powerful, allowing for all of the standard programming constructs (loops, conditions, functions, etc.) and also includes a number of standard function libraries that are used to interface to the system (e.g. perform file I/O, spawn a process, etc.). More information about programming in Tcl/Tk can be found at the end of this chapter.

The ScriptControl scripting language also contains a number of CarMaker specific commands. For example, commands are included that load and start a TestRun, perform DVA access, control the FailSafeTester, log data, and a number of other useful tasks.

The remainder of this document will introduce you to test automation in ScriptControl and provides a complete reference of all ScriptControl commands available.

1. The CIT is the set of tools used to interface to the virtual vehicle environment (VVE) of CarMaker, primarily consisting of the graphical user interfaces such as the CarMaker GUI, IPGControl, etc. Please refer to the CarMaker User's Guide for more information on the CIT.

2. The VVE is the simulated vehicle environment that contains the vehicle model, driver model, etc. Please refer to the CarMaker User's Guide for more information on the VVE.

17.2 Using the ScriptControl Window

The ScriptControl window is opened in the CarMaker GUI by going to the menu **Simulation / ScriptControl**. The figure below shows the parts of the ScriptControl window. In the following paragraphs we will describe each section:

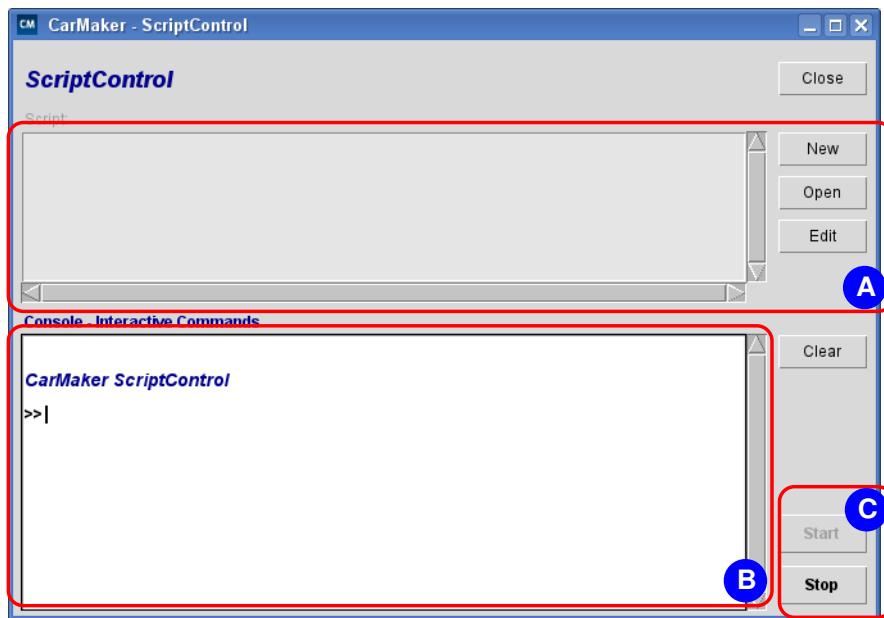


Figure 17.1: The ScriptControl dialog

A. Loaded Script Preview Area

In the upper part of the dialog you see the currently loaded script (read only). The buttons on the right side allow you to create a new script, load an existing one or open a script for editing.

B. Console, Interactive Commands

The console is used to directly interact with the CarMaker ScriptControl environment (as well as the operating system). ScriptControl commands (and Tcl commands in general) can be entered at the command prompt “>>” and will be executed immediately. When a command has a return value, it will be shown. The cursor keys can be used to access former commands and to edit the command line.

The console can also be used to view output from the scripts via the *Log* command. It is also very helpful for debugging scripts. While the script is running, you can enter commands to examine script variables, ...

C. Start/Stop Buttons

These buttons are used to start and stop the execution of the currently loaded script.

17.3 ScriptControl by Example

The easiest way to learn is by doing, so in this section we will write a few small scripts that demonstrate the basic functionality of ScriptControl.

17.3.1 Example 1 – “Hello World!”

As is the custom when learning a programming language, we will start with the “Hello World!” example that prints a text string to the screen. In this case the screen is the ScriptControl command window. The following description gives step-by-step instructions for writing a simple script that prints the line “Hello World!” to the ScriptControl command window.

- Step 1** Open the ScriptControl dialog – CarMaker GUI menu **Simulation / ScriptControl**.
 - Step 2** Create a new script file – Press **New**. Use *HelloWorld.tcl* as name for the new script.
 - Step 3** Type the following in the text editor:

```
Log "Hello World!"
```
 - Step 4** To test the script, simply load it and press the Start button. The script will be displayed in the Preview Area and the *Start* and *Stop* buttons will become active. Press the *Start* button to run the script. The test string “Hello World!” should be printed to the ScriptControl Console. If not, check that there were no spelling errors or that there are no upper-case letters where there should be lowercase letters, and vice-versa (i.e. the commands are case sensitive). Correct any syntax errors and try again.
- Analysis** The command *Log messagestr* is used to print text to the console. The *Log* command can also be used to print to a log file, or to both the command window and a log file. See the function description for more detailed information.

17.3.2 Example 2 – Starting a TestRun

In this example we will demonstrate how ScriptControl can be used to start and stop CarMaker TestRuns.

- Step 1** Create a new script file – Press **New**. Use *StartStopTest.tcl* as name for the new script.
 - Step 2** Type the following text and save the file:

```
LoadTestRun "Examples/VehicleDynamics/Road_Hockenheim_R8"  
StartSim  
WaitForTime 30  
StopSim
```
 - Step 3** Load the script and press the *Start* button. The example TestRun named Hockenheim is loaded, started, ran for 30 seconds and stopped.
- Analysis** This example uses four commands. The first, *LoadTestRun*, will load the TestRun specified in its arguments. The second command, *StartSim*, starts the TestRun that was loaded with *LoadTestRun*. The third, *WaitForTime*, is a function that waits for the specified amount of time before allowing the script to continue. The argument is in seconds. The last command used in this example, *StopSim*, will stop the simulation.

The two commands *StartSim* and *StopSim* perform the same function as the big red and green *Start* and *Stop* buttons in CarMaker’s main dialog (NOTE: they are not the same as the red and green *Start/Stop* buttons in the ScriptControl GUI).

17.3.3 Example 3 – Subscribing to Quantities

During a simulation there are a number of variables that are updated and that can be viewed during a running simulation (e.g. with tools such as IPGControl). Simulation variables like the vehicle's velocity, acceleration, yaw rate, etc. can be viewed and plotted in both realtime and offline by using the appropriate evaluation utility (e.g. IPGControl or Matlab). The simulation variables are called *Quantities* in the CarMaker vernacular.

To have access to the quantities, first you have to subscribe the quantities you are interested in. Subscribing means the values of selected quantities will get transferred regularly in the background (update rate approximately 50..100 Hz) from the simulation program to ScriptControl. The values are placed in a shadow variable, the global array *Qu*.

The following example shows how to subscribe to one of the quantities in the CarMaker data dictionary. The quantity is then used in a conditional statement to branch the execution according to its value.

Step 1 Create a new script file – Press New. Use *QuantSubTest.tcl* as name for the new script.

Step 2 First, you have to subscribe all quantities you want to deal with. This is done with the command *QuantSubscribe* followed by the list of quantities to be subscribed.

```
QuantSubscribe {Time Car.v}
```

Step 3 The following commands will load a TestRun, start the simulation and wait for the simulation to start.

```
LoadTestRun "Examples/VehicleDynamics/Road_Hockenheim_R8"  
StartSim  
WaitForStatus running
```

Step 4 Wait for the simulation time to reach 50 seconds.

```
WaitForCondition {$Qu(Time) > 50}
```

Step 5 Check the velocity when the simulation time is > 50. If the speed is less than 20 m/s then stop the simulation. Otherwise, continue until the simulation time is > 100.

```
if {$Qu(Car.v) < 20} {  
    StopSim  
} else {  
    WaitForCondition {$Qu(Time) > 100}  
    StopSim  
}
```

Step 7 Log the time and velocity before exiting the script.

```
Log "Speed = $Qu(Car.v) Time = $Qu(Time)"
```

Step 8 Save the script. Load it into ScriptControl and run it.

Analysis The whole script looks like the following:

```
QuantSubscribe {Time Car.v}  
  
LoadTestRun "Examples/CarMakerFunctions/IPGRoad/Hockenheim"  
StartSim  
  
WaitForStatus running  
WaitForCondition {$Qu(Time) > 50}  
  
if {$Qu(Car.v) < 20} {  
    StopSim
```

```
    } else {
        WaitForCondition {$Qu(Time) > 100}
        StopSim
    }
    Log "Speed = $Qu(Car.v), Time = $Qu(Time)"
```

Being able to subscribe to quantities and use them inside of scripts is one of the most important features of ScriptControl. When combined with the standard Tcl commands and the ScriptControl command extensions, the complexity of the script is only limited by the imagination of its creator.

17.3.4 More Examples

Included in the distribution is an examples directory *Data/Script/Examples* that contains basic scripts created to demonstrate some of the functionality of ScriptControl. The examples include:

- *Boolean.tcl*
Example of some of the simple boolean expressions that can be formed in Tcl
- *Clock.tcl*
Example showing how to use the clock timer commands
- *DVA.tcl*
Example of the Direct Variable Access commands
- *FST.tcl*
Example of FailSafeTester commands (CarMaker/HIL only)
- *ImportResFile.tcl*
Example of the ImportResFile command
- *Import_VehicleData.tcl*
Demonstrates how to import selected vehicle parameters from other vehicle datasets.
- *IndicatorLights.tcl*
Shows how conditions can be used to show pass/fail status (CarMaker/HIL only)
- *InfoFileModify.tcl*
Example showing how to make changes directly to the InfoFile parameter database of CarMaker
- *Math.tcl*
Simple Tcl math function example
- *MiniManJump.tcl*
Shows how to jump to a specific minimaneuver
- *OutputQuants.tcl*
Example showing how to make changes to the output quantities list and how to save the results
- *QuantAudit.tcl*
Small script setting up a few QuantAudit-jobs, running a simulation and retrieving results from the Scratchpad afterwards.
- *Print.tcl*
Printing example
- *QuantSubscribe.tcl*
Example showing how to subscribe to a quantity
- *RunScripts.tcl*
Example showing how multiple scripts can be run from a single script
- *SessionLog.tcl*
Example that prints to the CarMaker session log

- *SimStatus.tcl*
Shows a command that will return the simulation status
- *StartStop.tcl*
Example showing how to start and stop a TestRun
- *WaitForCondition.tcl*
Example showing the WaitForCondition command and how it can be used to pause script execution until the specified condition is met.

17.3.5 Tcl/Tk Documentation Links

Documentation available locally on Linux hosts



Under Linux, a special Tcl/Tk installation is provided by IPG. The complete documentation can be viewed with the web browser of your choice under the following URL:

file:/opt/tcl/html/contents.htm



Documentation available locally on MS Windows hosts

On Windows systems, the ActiveStateTcl package recommended for the CarMaker version to be used must be installed (available at IPG or on the Internet). The complete documentation can be found following the corresponding entry in the Start menu, in **Programs / ActiveStateTcl...**

Documentation available on the internet

The Tcl Developer Xchange is the home of Tcl/Tk development:

http://www.tcl.tk

The Tcler's Wiki is a collaborative web site with many useful pages of tips and tricks:

http://wiki.tcl.tk

Tcl/Tk literature

Tcl/Tk books can be purchased at all major bookstores. Lists of available books are accessible on the web sites mentioned before.

17.4 ScriptControl Command Reference

17.4.1 Loading and Running a Simulation

NewTestRun

Syntax

```
NewTestRun ?force?
```

Description

Clears already loaded TestRun data.

If called with force==1, no check for modified data is done, that means no popup occurs.

LoadTestRun

Syntax

```
LoadTestRun path ?force?
```

Description

Loads the predefined CarMaker TestRun stored in *path*. Specification of a relative path causes the TestRun to be loaded relative to the *Data/TestRun* directory of your CarMaker project directory. In case the TestRun is not available there, CarMaker searches first in data pools (if defined) and then in the installation directory.

If called with force==1, no check for modified data is done, that means no popup occurs.

The function returns ““ in case of success or a keyword indicating the error reason: *failed*, *cancel* or *incomplete*.

Example

```
LoadTestRun "Examples/Braking"
```

StartSim

Syntax

```
StartSim
```

Description

Starts the simulation of the currently loaded TestRun.

This function corresponds to the big green *Start* button in the CarMaker main dialog.

StopSim

Syntax

StopSim

Description

Stops the current simulation.

This function corresponds to the big red *Stop* button in the CarMaker main dialog.

SimStatus

Syntax

SimStatus

Description

Returns the current simulation status as shown in GUI main dialog as an integer value:

Value	Description
>= 0	Current maneuver number, simulation is running
-1	Preprocessing
-2	Idle
-3	Postprocessing
-4	Model Check
-5	Driver Adaption
-6	FATAL ERROR / Emergency Mode
-7	Waiting for License
-8	Simulation paused
-10	Starting application
-11	Simulink initialization

Example

```
set status [SimStatus]
```

WaitForStatus

Syntax

WaitForStatus *status* ?*timeout_ms*?

Description

Causes the script to wait until the specified simulation status is reached. Legal values for *status* are:

idle – The simulation has stopped, no simulation is running

running – The simulation is running

When the requested simulation status has been reached, the function returns 0.

An optional timeout value specified in milliseconds can be supplied. If the simulation does not enter the requested simulation status within *timeout_ms* milliseconds (wall clock time, not simulation time), the functions stops waiting and returns -1. The default timeout is set to 1.000.000 milliseconds.

Examples

```
WaitForStatus running 5000  
WaitForStatus idle
```

WaitForTime

Syntax

`WaitForTime seconds`

Description

Causes the script to wait until the simulation time reaches the specified time.

Tip: for very precise timings use RExpressions.

Example

```
WaitForTime 20
```

WaitForCondition

Syntax

`WaitForCondition condition ?timeout_ms?`

Description

Causes the script to wait until the specified condition is satisfied. *condition* will be evaluated in the context of the caller using Tcl's *expr* command.

When the requested condition has been met, the function returns 0.

An optional timeout value specified in milliseconds can be supplied. If the simulation does not reach the condition within *timeout_ms* milliseconds (wall clock time), the functions stops waiting and returns -1.

Tip: for very precise timings use RExpressions.

Example

```
WaitForCondition {$Qu(Car.v) > 50}
```

ManJump – Jump to Mini-Maneuver

Syntax

```
ManJump man_no
ManJump label
```

Description

Causes the CarMaker simulation program to jump to the specified mini-maneuver *man*, which can be specified in several ways:

- *number* – number of mini-maneuver step (0 is the first mini-maneuver),
- *+/-count* – relative jump *count* mini-manuevers forward/backward,
- *label* – user defined *label* of mini-maneuver.

Example

```
ManJump 3
ManJump -1
ManJump Finish
```

SetMiniManCmd – Set Mini-Maneuver Command

Syntax (deprecated command)

```
SetMiniManCmd manid cmdstr ?activatenow?
```

Description

Causes the commands of the mini-maneuver *manid* to be replaced by the specified command string. *manid* must be a valid a mini-maneuver identifier of the TestRun currently running. *cmdstr* is a string of mini-maneuver commands, separated by a newline character (\n), to set for this mini-maneuver.

A boolean value (0 or 1) for the parameter *activatenow* indicates whether CarMaker should jump to this mini-maneuver immediately after setting its command string. Specifying this parameter is optional, default is 1 (i.e. activate the mini-maneuver).

This command now is deprecated, please use the more powerful RTexpressions. Starting with the next major release of CarMaker this command will not be available.



17.4.2 Subscribing to Quantities

QuantSubscribe

Syntax

```
QuantSubscribe quantity_list
QuantSubscribe -show
```

Description

Subscribes all quantities in the list *quantity_list*, before returning, it will wait for the first quantity vector to arrive.

Subscribing to a quantity allows the value of the quantity to be read by the script. Any of the quantities that are defined in the CarMaker data dictionary can be accessed, allowing the value to be used in the user's test automation script.

To have access to the quantities, first you have to subscribe the quantities you are interested in. The subscribed quantities will be transferred from the simulation program to the ScriptControl interpreter at regular intervals in the background (update rate approximately 100 Hz). The values are placed in a shadow variable, the global array *Qu*.

Subscribing variable generates some amount of background activity, simulation program runs on a different host (as in CarMaker/HIL), it also generates network traffic. Therefore only a small subset of the available quantities is subscribed. Just to give an idea of the needed bandwidth: subscribing 20 float quantities results in a traffic of 10 KByte/sec, this is about 1% of the bandwidth of fast ethernet (100 MBit).

Some special cases:

- Calling QuantSubscribe with an empty list {} will stop the subscription.
- *QuantSubscribe -show* returns the list of currently subscribed quantities.
Note: You may find some additional quantities in this list, they are needed internally.

Changes compared to CarMaker 2.0



- Please note that in ScriptControl of CarMaker 2.0 the values were also available as a formatted string in the *Quantities* array. For performance reasons this is no longer done. If you need a formatted string, for example for right justified output or to have a fixed number of digits right of the decimal point, please use the *Log* command with a format string as first parameter (shown in the example below). Alternatively you can use the Tcl *format* command (see Tcl documentation).
- Previously a different technique was used to subscribe quantities. First a global array *Quantities* had to be filled with the requested quantity names and their default value. After that the command QuantSubscribe was called without parameters.

```
# OLD STYLE !
array set Quantities {
    Car.v      0
    Time       0
}
QuantSubscribe
```

The old style still supported, but should be avoided.

Example

```
# subscribe two quantities: Time, Car.v
```

```
QuantSubscribe {Time Car.v}

# values output with full precision
Log "speed=$Qu(Car.v) at time=$Qu(Time)"

# output with limited precision (here 1 digit right of the decimal point)
Log "speed=%1f at time=%1f" $Qu(Car.v) $Qu(Time)

# cancel subscription
QuantSubscribe {}
```

17.4.3 Clock Timers

Clock timers (or simply timers) are used to measure the amount of time that a condition is true. For example, the time that the vehicle speed is greater than 50 kph can be measured by using a timer.

The clock commands work using the simulation time Global.Time and are only relevant when a simulation is running. When no simulation is running the clock functions will not give predictable results.

ClockCreate

Syntax

ClockCreate *clockname* *condition*

Description

Creates a clock timer by giving it a user specified *clockname* and associating a condition with it. Once the clock timer is started with *ClockStart*, *condition* will be evaluated regularly at global scope using Tcl's *expr* command and is expected to yield a boolean value.

Example

```
ClockCreate MyClock {$Qu(DM.Steer.Ang) > 0}
```

ClockDelete

Syntax

ClockDelete *clockname*

Description

Deletes a clock timer previously created with *ClockCreate* and frees the resources allocated for it. Whenever a clock timer will no longer be used, it should be deleted with this command.

Example

```
ClockDelete MyClock
```

ClockStart

Syntax

ClockStart *clockname*

Description

Starts a clock timer previously created with *ClockCreate*.

Example

```
ClockStart MyClock
```

ClockStop

Syntax

`ClockStop clockname`

Description

Stops a clock timer previously created with *ClockCreate* and started with *ClockStart*.

Example

```
ClockStop MyClock
```

ClockGetTime

Syntax

`ClockGetTime clockname`

Description

Returns the elapsed clock time of a clock timer. The time can be read any time after *ClockStart* is called.

Example

```
set TotalTime [ClockGetTime MyClock]
```

ClockReset

Syntax

`ClockReset clockname`

Description

Resets the clock time to 0 without stopping it.

Example

```
ClockReset MyClock
```

17.4.4 Logging

In ScriptControl, two different concepts of logging exist.

The *session log* is a log file maintained by the CarMaker simulation program. This is the place where messages pertaining to each simulation are logged. The CarMaker can display the session log in a separate window.

In your ScriptControl scripts, occasionally you may also want to issue messages describing the progress of the automation script or inform about some important condition. Often execution of your script spans multiple simulations and it is sufficient for a log message to appear only in the ScriptControl console window. This is why a second logging mechanism independent of CarMaker's session log was introduced.

The *ScriptControl log* is a log that uses the ScriptControl console window for output. Optionally a separate log file can be opened and messages will be put into that file, too.

It is recommended that ScriptControl scripts stick to the ScriptControl log for logging and keep messages to the CarMaker log to a minimum.

OpenSLog – Open the ScriptControl log file

Syntax

`OpenSLog path ?directory?`

Please note: Specification of the *directory* parameter is unnecessary and supported for compatibility reasons only. Its use is strongly deprecated.

Description

Opens the file with the specified *path* as the ScriptControl log file. If the file already exists, log messages will be appended to the end of the file. Using the optional *directory* argument is equivalent to invoking the function as `OpenSLog directory/path`, i.e. it will simply be prepended to the *path* argument.

If *path* consists only of a single filename, the file will be opened in the *SimOutput/ScriptLog* subdirectory. In order to open a log file in the current working directory, specify it as `./filename`.

Examples

```
OpenSLog MyScript.log  
OpenSLog ./LogFileInCurrentDir.log  
OpenSLog SubDir/LogFileInOtherDir.log
```

CloseSLog – Close the ScriptControl log file

Syntax

`CloseSLog`

Description

Closes the ScriptControl log file that was opened in the previous call to *OpenSLog*.

Log – Write to the ScriptControl log

Syntax

```
Log ?where? messagestr  
Log ?where? formatstr arg ?arg ...?
```

Description

Write a ScriptControl log message. The command supports two ways of specifying the message to be logged.

The simple form consists of a single message string *messagestr*, that will be directly written to the log. The more complex form consists of a format string controlling how the remaining *arg* argument(s) should be formatted. Internally Tcl's *format* command is used for this purpose; details about the content of *formatstr* can be looked up in the corresponding Tcl manual page. For a first impression see the examples below.

Depending on the *where* parameter, the log message will be output on either the ScriptControl console window, the ScriptControl log file, or both. Possible values for *where* are:

screen – The message is output only to the ScriptControl console window.

file – The message is output only to the ScriptControl log file. The log file must be opened with *OpenSLog* before, otherwise an error is issued.

screen+ or **file+** – The message is output to both the ScriptControl console window and the ScriptControl log file. The log file must be opened with *OpenSLog* before, otherwise an error is issued.

If the *where* parameter is not specified, the behaviour is as follows: The message is output to the ScriptControl console window. If and only if a ScriptControl log file was opened with *OpenSLog* before, the message is also output to the log file.

For most uses, there is no need to bother with the *where* parameter. Invoking the function with the message you want to log will just do the right thing.

Examples

```
Log "Message sent to the ScriptControl window and possibly to the ScriptControl log file."  
Log screen "Message sent only to the ScriptControl window"  
Log file "Message sent only to the ScriptControl log file"  
Log screen+ "Message sent to both the ScriptControl window and log file."  
Log file+ "Message sent to both the ScriptControl window and log file."  
  
Log "v = %6.2f m/s" $Qu(Car.v)  
Log "Gas = %2f %, Brake = %2f %%" [expr $Qu(DM.Gas) * 100.0] [expr $Qu(DM.Brake) * 100.0]
```

SessionLogMsg

SessionLogWarn

SessionLogError – Issue a message to the CarMaker log

Syntax

```
SessionLogMsg messagestr  
SessionLogMsg formatstr arg ?arg ...?
```

SessionLogWarn messagestr

SessionLogWarn formatstr arg ?arg ...?

```
SessionLogError messagestr
SessionLogError formatstr arg ?arg ...?
```

Description

Writes a message to the CarMaker session log, i.e. the log that is maintained by the Car-Maker simulation program. Depending on the function that is invoked, the message will be logged either as a normal message, a warning or an error. The commands support two ways of specifying the message to be logged.

The simple form consists of a single message string *messagestr*, that will be directly written to the log. The more complex form consists of a format string controlling how the remaining *arg* argument(s) should be formatted. Internally Tcl's *format* command is used for this purpose; details about the content of *formatstr* can be looked up in the corresponding Tcl manual page. For a first impression see the examples below.

It is recommended to keep session log messages to a minimum and issue log entries line by line, not en bloc with embedded newline characters. Log messages that exceed the internal buffer size limit (currently about 2000 characters) will be truncated silently. Verbose logging can have negative effects on readability of the log file and increases network and CPU usage.

Examples

```
SessionLogError "Error: There was an error detected! Take the appropriate action."
SessionLogWarn "Vehicle is too fast: v = %6.2f m/s" $Qu(Car.v)
```

17.4.5 Infofile Parameter Access

An infofile is an ASCII file that contains key-value pairs which represent simulation input parameters. The key-value pairs are used to parameterize the simulation models and allow the user the flexibility to customize the VVE so it behaves as desired (usually making it behave like a real world vehicle, driver, etc. would). ScriptControl allows direct access to the infofile database information and can therefore be used to read or make parameter changes “on the fly”.

A minimal understanding about how information is stored in infofiles is required in order to understand the commands described below. Two kinds of keys can be present in an infofile:

A *string key* is, roughly spoken, “a key with an equal sign”, e.g.

```
CarLoad.0.mass = 50  
PowerTrain.Clutch.Kind = Manual
```

An *text key* is, roughly spoken, “a key with a colon” and spans multiple lines, e.g.

```
SuspR.Spring:  
-0.01000 -250.00  
0.00000 0.00  
1.00000 25000.00
```

Keys should always be accessed using commands that match their key kind, e.g. for text keys always use *IFileReadTxt* and *IFileModifyTxt*.

Please note: For the purpose of parameter variations the preferred way of modifying parameters are via *named values* and *key values*. Both share the advantage that the actual file containing a parameter does not need to be modified for each new parameter value. A description of the corresponding *NamedValue* and *KeyValue* commands can be found in the chapters immediately following.

IFileRead

IFileReadTxt – Get a parameter's value

Syntax

```
IFileRead paramfile key ?defaultvalue?  
IFileReadTxt paramfile key ?defaultvalue?
```

Description

IFileRead returns the value of the specified string key.

IFileReadTxt returns the value of the specified text key as a list of strings.

If an entry for *key* does not exist, both commands return the value of the *defaultvalue* parameter; in case *defaultvalue* is not specified, an empty string is returned.

The *paramfile* parameter is a shortcut for the infofile in question, legal values are:

SimParameters – The *Data/Config/SimParameters* file

ECUParameters – The *Data/Config/ECUParameters* file

TestRun – The TestRun file currently loaded

Vehicle – The vehicle parameter file as specified in the current TestRun

Trailer – The trailer parameter file as specified in the current TestRun

Brake – The brake parameter file as specified in the current vehicle

Tire0, Tire1,..., Tire7 – The tire parameter file as specified in the current TestRun

TrBrake – The brake parameter file as specified in the current trailer

TrTire0, TrTire1,..., TrTire7 – The tire parameter file as specified in the current trailer
path/file – explicit specified name of the info file (path must contain a '/')

Examples

```
set Load_0 [IFileRead TestRun CarLoad.0.mass]
set AeroData [IFileReadTxt Vehicle SuspR.Spring]
```

IFileModify

IFileModifyTxt – Set a parameter's value

Syntax

```
IFileModify fileparam key value
IFileModifyTxt fileparam key value
```

Description

IFileModify sets the value of the specified string key.

IFileModifyTxt sets the value of the specified text key. In this case *value* should be a valid Tcl list; each list element will be stored in a separate line in the infofile.

If the key does not exist then one will be created, otherwise its value will be overwritten.

The *fileparam* parameter is a shortcut for the infofile in question, for legal values see the description of *IFileRead/IFileReadTxt* in this chapter.

Do not forget to invoke *IFileFlush* before starting the next TestRun with *StartSim*.



Examples

```
IFileModify TestRun CarLoad.0.mass 50
IFileModifyTxt Vehicle SuspR.Spring {
    { -0.001 250.0 }
    { 0.0     0.0 }
    { 1.0     25000.0 }
}
```

IFileFlush

Syntax

```
IFileFlush
```

Description

Writes the changes made to infofiles by the *IFileModify* commands to the actual files. This operation is required since out of performance and resource considerations *IFileModify* operations are cached in memory.

Always call this command after having made changes to infofiles with *IFileModify* and before starting the next TestRun using *StartSim*.



17.4.6 KeyValue Parameter Substitution

Key values provide a means of changing parameters in infofiles without actually modifying the physical file. Whereas named values provide something like a macro substitution mechanism inside the data associated with an infofile key, key values redefine complete keys.

A key value consists of a name and an associated value and must be defined before a simulation. The CarMaker GUI provides all currently defined key values to the simulation program whenever a TestRun is started, right before the actual simulation begins to run. When the simulation program accesses an infofile parameter for which a key value definition exists, it reads the predefined value instead of the one contained in the physical file.

In the following command reference *name* can be just the name of an infofile key, meaning to override the definition of the key in *all* infofiles read for the TestRun. Alternatively, in order to avoid possible ambiguities, *name* may be specified as *place:name*, where *place* designates a certain file or subsystem, and only the key appearing in this place should be overridden. Possible values for *place* are:

```
SimParameters  
ECUParameters  
TestRun  
Vehicle  
Aero  
Brake  
Steering  
SuspExtFrcs  
TireFL, TireFR, TireRL, TireRR, TireRL2, TireRR2, TireFL2, TireFR2  
PowerTrain  
PTEngine  
PTClutch  
PTGearBox  
PTDriveLine  
UserDriver  
VehicleControl  
Trailer  
TrAero  
TrBrake  
TrSuspExtFrcs  
TrTireFL, TrFireFR, TrTireRL, TrFireRR, TrTireML, TrFireMR
```

Infofile key names and possible values used by CarMaker are described in the CarMaker Reference Manual. Alternatively one may inspect an infofile directly using any ordinary ASCII text editor.

KeyValue set – Set a key value

Syntax

KeyValue set *name value*

Description

Defines a key value called *name* with a value *value*. Any previous definition of *name* will be replaced. The key value will also be exported to the Matlab workspace (CarMaker for Simulink only).

Please note that the resulting infofile key is always a string key (*name = value*), regardless of the original key. Future versions of this command may also support text keys (*name ...*).

Example

```
KeyValue set Body.mass 1500
```

KeyValue get – Retrieve the value of a previously set key value

Syntax

```
KeyValue get name
```

Description

Returns the value of the specified key value *name*.
In case key value *name* is not defined, an empty string is returned.

Example

```
KeyValue set ThisParam 1234  
KeyValue get ThisParam  
1234
```

KeyValue del – Delete a previously set key value

Syntax

```
KeyValue del name
```

Description

Deletes the key value *name*. The specified key value is not required to exist.

Example

```
KeyValue set ThisParam 42  
KeyValue exists ThisParam  
1  
KeyValue del ThisParam  
KeyValue exists ThisParam  
0
```

Please note, that this command accepts patterns using an asterisk (*), e.g:



```
KeyValue set k1 1  
KeyValue set k2 2  
KeyValue set k12 22  
KeyValue del k1*  
KeyValue names k*  
k2
```

KeyValue exists – Check if a key value exists

Syntax

KeyValue exists *name*

Description

Returns 1 if the specified key value *name* is currently defined, 0 if not.

Example

```
KeyValue set ThisParam 42
KeyValue exists ThisParam
    1
KeyValue exists NoSuchThing
    0
```

KeyValue names – List all currently defined key values

Syntax

KeyValue names

Description

Returns the (possibly empty) list of all currently defined key values.

Example

```
KeyValue set ThisParam 42
KeyValue set ThasParam 43
KeyValue names
    ThisParam ThatParam
foreach p [KeyValues names] { Log "<$p> is set to <[KeyValue get $p]>" }
    <ThisParam> is set to <42>
    <ThatParam> is set to <43>
```

Please note, that this command accepts patterns using an asterisk (*), e.g:



```
KeyValue set k1 1
KeyValue set k2 2
KeyValue names k*
    k1 k2
```

17.4.7 NamedValue Infofile Parameter Substitution

Named values provide a means of changing parameters in infofiles without actually modifying the physical file. Contrary to key values, which serve to redefine complete keys, named values provide some simple kind of macro substitution inside the data associated with an infofile key.

The data associated with an infofile key may contain embedded references to named values like `$name` and `$name=defaultvalue`. The latter example shows how to provide a meaningful value even in case that for the currently running simulation no definition for `name` was given.

Named values consist of a `name` and an associated value and must be defined before a simulation. The CarMaker GUI provides all currently defined named values to the simulation program whenever a TestRun is started, right before the actual simulation begins to run. When the simulation program accesses an infofile parameter that contains a reference to a named value, the reference will automatically be replaced with the value.

NamedValue set – Set a named value

Syntax

`NamedValue set name value`

Description

Defines a named value called `name` with a value `value`. Any previous definition of `name` will be replaced. The named value will be also exported to the Matlab workspace (CarMaker for Simulink only).

Example

```
NamedValue set xy 42
```

NamedValue get – Retrieve the value of a previously set named value

Syntax

`NamedValue get name`

Description

Returns the value of the specified named value `name`. In case named value `name` is not defined an empty string is returned.

Example

```
NamedValue set xy 42
NamedValue get xy
42
```

NamedValue del – Delete a previously set named value

Syntax

NamedValue del *name*

Description

Deletes the named value *name*. The specified named value is not required to exist.

Example

```
NamedValue set xy 42
NamedValue exists xy
    1
NamedValue del xy
NamedValue exists xy
    0
```

Please note, that this command accepts patterns using an asterisk (*), e.g:



```
NamedValue set k1 1
NamedValue set k2 2
NamedValue set k12 22
NamedValue del k1*
NamedValue names k*
    k2
```

NamedValue exists – Check if a named value exists

Syntax

NamedValue exists *name*

Description

Returns 1 if the specified named value *name* is currently defined, 0 if not.

Example

```
NamedValue set xy 42
NamedValue exists xy
    1
NamedValue exists nosuch
    0
```

NamedValue names – List all currently set named values

Syntax

NamedValue names

Description

Returns the (possibly empty) list of all currently defined named values.

Example

```
NamedValue set xy 42
NamedValue set zz 43
NamedValue names
    xy zz
foreach p [NamedValues names] { Log "<$p> is set to <[NamedValue get $p]>" }
    <xy> is set to <42>
    <zz> is set to <43>
```

Please note, that this command accepts patterns using an asterisk (*), e.g:



```
NamedValue set k1 1
NamedValue set k2 2
NamedValue names k*
    k1 k2
```

17.4.8 Managing the Scratchpad

To get a picture of what is behind, think of somebody observing the simulation, writing remarks on differently colored post-it notes each time something noteworthy happens, and gluing them to a board, ordered by color.

The CarMaker Scratchpad corresponds to the board, the differently colored post-its are *notes* of different *types*, the remarks correspond to data stored in *fields* on a note, and the person creating the post-its and writing on them is some C-code using the Scratchpad API as defined in header file *<Scratchpad.h>*.

Contrast the Scratchpad with the quantities from the data dictionary, as each concept has its purpose: Quantities stand for continuous data, Scratchpad notes are created only sporadically, often triggered by some kind of asynchronous event. Quantities are “high-volume”, scratchpad notes are “few”.

Examples sources of Scratchpad notes are the QuantAudit facility (see below) and the pylon detection module. Many of the examples below refer to *PylonHit* Scratchpad notes resulting from an example TestRun *Examples/TestRun/VehicleDynamics/Slalom_18m*. Future versions of CarMaker may also introduce Scratchpad notes generated by Real-Time-Expressions.

Scratchpad notes are stored, when a result file is written during a simulation. Scratchpad data inside a CarMaker result file can be read with the *ImportResFile* command.

Scratchpad containers

Scratchpad notes can be put into separate containers. Each container has a unique name, which must be specified when accessing the container's data. The container name may also be omitted, in which case the so called default container is used. The default container can also be accessed by specifying “default” as the container name. Here is the scheme, but see also the examples paragraph at the end of each command's description:

container::notetype – This form refers to Scratchpad notes of type *notetype* stored in container *container*.

notetype – This simplified form refers to Scratchpad notes of type *notetype* in the default container and is the shorthand notation for *default::notetype*.

Scratchpad notes sent by the simulation program are always stored in the default container. When reading Scratchpad data from a file with the *ImportResFile* command, a container name for file's data must be specified explicitly.

Scratchpad basics

All existing Scratchpad data will be cleared at the beginning of a TestRun.

The field definitions for a note type are transmitted to the CarMaker GUI automatically the first time a note of this type is created during a TestRun.

During a simulation the default scratchpad is filled continuously as notes are sent by the simulation program, so it is possible to monitor the scratchpad e.g. with ScriptControl for the arrival of a certain type of note.

By definition a Scratchpad note provides at least the following two fields:

time – The current simulation time when the note was created.

dist – The driven distance when the note was created.

Beyond these two a note may, but does not need to, provide additional fields. Field data is always one line, ASCII. A field may contain arbitrary items such as one or more numerical values and strings. As such, field data is completely application defined.

Scratchpad types – List all note types currently available

Syntax

Scratchpad types ?*pattern*?

Description

Returns a (possibly empty) list of all note types currently available and matching the specified (glob-style) *pattern*. If pattern is omitted all types available in the default container are returned.

Please note that pattern matching is applied only to note types, not to container names.

Example

```
Scratchpad types
    QuantAudit-Car.v QuantAudit-Car.ax PylonHit
ImportResFile SimOutput/rt1/20100504/Slalom_18m.erg {Time Car.v} qdata sp
Scratchpad types sp::*:
    sp::PylonHit
Scratchpad list sp::PylonHit sRoad
    0 372      1 390      2 408      3 426      4 444
```

Scratchpad list – List notes and note contents

Syntax

Scratchpad list *type* ?*field...?*

Description

Returns a list of note indices and, if specified, selected fields from each note of type *type*. The returned list is flat. For n notes stored on the Scratchpad and fields *field-a*, *field-b*,... specified for the command, the list would be organized as follows (see also the example below):

0 *field-a*₀ *field-b*₀ ... 1 *field-a*₁ *field-b*₁ ... n-1 *field-a*_{n-1} *field-b*_{n-1}

If no notes of type *type* currently exist, an empty list is returned.

Please note, that field values of individual notes may also be retrieved using the *Scratchpad get* command.

Example

```
Scratchpad list PylonHit
    0      1      2      3      4
Scratchpad list PylonHit sRoad
    0 372      1 390      2 408      3 426      4 444
Scratchpad list PylonHit sRoad side
    0 372 0 1 390 1 2 408 0 3 426 1 4 444 0
foreach {i sRoad side} [Scratchpad list PylonHit sRoad side] {
    Log "Note #{$i}: sRoad=$sRoad side=$side"
}
Note #0: sRoad=372 side=0
Note #1: sRoad=390 side=1
Note #2: sRoad=408 side=0
Note #3: sRoad=426 side=1
Note #4: sRoad=444 side=0
```

Scratchpad get – Retrieve note data

Syntax

```
Scratchpad get type index ?field?
```

Description

Returns, from the specified note of type *type*, the value of the specified field. If *field* is omitted, a list of all field values is returned. If the note specified by *index* does not exist, an empty string is returned.

Example

```
Scratchpad list PylonHit
0 1 2 3 4
Scratchpad dump PylonHit 0
PylonHit[0]:
    time      : 27.363
    dist      : 363.785918
    index     : 0
    id        : 10
    side      : 0
    sRoad     : 372
    pos_0.x   : 372
    pos_0.y   : 0
Scratchpad get PylonHit 0
27.363 363.785918 0 10 0 372 372 0
Scratchpad get PylonHit 0 dist
363.785918
Scratchpad get PylonHit 0 5
372
```

Scratchpad exists – Check if a note exists

Syntax

```
Scratchpad exists type ?index?
```

Description

Returns 1 if the specified note of type *type* exists. If *index* is omitted, returns 1 if any note of the specified type exists. Otherwise 0 is returned.

Example

```
Scratchpad list PylonHit
0 1 2 3 4
Scratchpad exists PylonHit
1
Scratchpad exists PylonHit 4
1
Scratchpad exists PylonHit 5
0
Scratchpad exists NoSuchType
0
```

Scratchpad fields – List field names of a note type

Syntax

Scratchpad fields *type*

Description

Returns a list of all fields and their position defined for note type *type*, ordered by position. Either the field name or the field position may be used to access a note's content with the *Scratchpad get* command. If no notes of type *type* currently exist an empty list is returned.

Example

```
Scratchpad fields PylonHit
    time 0 dist 1 index 2 id 3 side 4 sRoad 5 pos_0.x 6 pos_0.y 7
```

Scratchpad containers – List existing containers

Syntax

Scratchpad containers ?*pattern*?

Description

Returns a list of currently existing containers for Scratchpad notes. A (glob-style) pattern may be specified, so only containers are listed, whose name matches *pattern*. If no pattern is specified, all containers are listed.

Example

```
ImportResFile SimOutput/rt1/20100504/Slalom_18m.erg {Time Car.v} qdata myContainer
Scratchpad containers
    default myContainer
Scratchpad containers my*
    myContainer
```

Scratchpad dump – Show the content of Scratchpad notes

Syntax

Scratchpad dump *type* ?*index*?

Description

Dumps the contents of the specified note of type *type* on the ScriptControl console. If *index* is omitted, all notes of type *type* will be dumped.

This command is intended as a debugging aid. A list of valid note indices can be obtained with the *Scratchpad list* command.

Example

```
Scratchpad list PylonHit
    0 1 2 3 4
Scratchpad get PylonHit 0
    27.363 363.785918 0 10 0 372 372 0
```

```
Scratchpad dump PylonHit 0
PylonHit[0]:
  time      : 27.363
  dist      : 363.785918
  index     : 0
  id        : 10
  side      : 0
  sRoad     : 372
  pos_0.x   : 372
  pos_0.y   : 0
```

Scratchpad clear – Clear the scratchpad

Syntax

```
Scratchpad clear ?pattern...?
```

Description

Removes all notes of the type(s) matching the specified (glob-style) *pattern*. Specifying the special value *all* as the type completely empties all containers of the Scratchpad.

Please note that pattern matching is applied only to note types, not to container names.

Example

```
Scratchpad exists PylonHit
  1
Scratchpad clear PylonHit
Scratchpad exists PylonHit
  0

Scratchpad clear QuantAudit-*
Scratchpad clear *
Scratchpad clear mySpad::*
Scratchpad clear all
```

17.4.9 QuantAudit – Pin-point Quantity Monitoring

Quantity auditing generates Scratchpad notes, which can be accessed using ScriptControl's *Scratchpad* command (see above). QuantAudit jobs are evaluated in the simulation program, in real-time. It is intended as a better, much more accurate and efficient replacement for former techniques like subscribing to quantities and continuously observing their values.

When a simulation is started, all QuantAudit jobs are communicated from the CarMaker GUI to the simulation program, which then takes care of the rest.

QuantAudit jobs for quantity *name* will create scratchpad notes of type QuantAudit-*name* on the default Scratchpad. Each note contains the following fields:

- tstart – The time observation for this quantity job did start.
- dstart – dto. for the driven distance.
- mstart – dto. for the minimaneuver number.
- tstop (or time) – The time observation for this quantity did stop.
- dstop (or dist) – dto. for the driven distance.
- mstop – dto. for the minimaneuver number.
- last – The last value of the quantity in the observed interval.
- min – The minimum value of the quantity in the observed interval.
- max – The maximum value of the quantity in the observed interval.
- mean – The mean value of the quantity in the observed interval.

A QuantAudit job can generate a maximum of 16 scratchpad notes. To increase this number please see the "Reference Manual", chapter 3.1.11.

QuantAudit add – Define a QuantAudit job

Syntax

```
QuantAudit add name -time interval
QuantAudit add name -dist interval
QuantAudit add name -man interval
QuantAudit add name -cond rtexpr
```

Description

Defines a new QuantAudit job for quantity *name* during the specified interval. When the end of the interval is reached a scratchpad note will be issued.

An interval can be related to the simulation time (-time), the driven distance (-dist), or the minimaneuver number (-man). In this case valid specifications for *interval* are as follows:

- val*
A single point *val*.
- fromval*-*t oval*
A range starting at *fromval* up to and including *t oval*.
- fromval*-end
A range starting at *fromval*, stretching to the end of the simulation.

No blanks are allowed in the specification of *interval*.

Alternatively a condition (-cond) can be specified with a Realtime Expression *rteexpr*. The first time *rteexpr* evaluates to true (i.e. non-zero), quantity auditing starts and lasts until *rteexpr* evaluates to false (zero).

In order to differentiate multiple QuantAudit jobs on the same quantity, using the special notation *name/tag* a user defined tag can be attached to the *name* parameter. Resulting scratchpad notes will be of type QuantAudit-*name/tag* correspondingly.

Example

```
QuantAudit add Car.v -time 0-end
QuantAudit add Car.ay -man 2
QuantAudit add Car.ax -cond {Car.Roll<-0.01 || Car.Roll>0.01}
QuantAudit names
    Car.v Car.ax Car.ay
```

QuantAudit del – Delete a QuantAudit job

Syntax

```
QuantAudit del name -time interval
QuantAudit del name -dist interval
QuantAudit del name -man interval
```

Description

Removes the QuantAudit job of the given specification. This command is completely symmetrical to the *QuantAudit add* command, for further details see above.

QuantAudit names – List all QuantAudit jobs

Syntax

```
QuantAudit names ?pattern?
```

Description

Returns a (possibly empty) list of all currently defined QuantAudit jobs matching *pattern*. If *pattern* is omitted, a list of all current jobs is returned..

Example

```
QuantAudit add Car.v -time 0-end
QuantAudit add Car.ax -time 0-end
QuantAudit add Vhcl.Yaw -time 0-end
QuantAudit names
    Vhcl.Yaw Car.ax Car.v
QuantAudit names V*
    Vhcl.Yaw
```

QuantAudit dump – Dump QuantAudit jobs

Syntax

```
QuantAudit dump ?name...?
```

Description

Dumps the definition of the specified QuantAudit job(s) on the ScriptControl console, in alphabetical order. If no name is given, all QuantAudit jobs will be dumped.

This command is intended as a debugging aid. A list of valid QuantAudit jobnames can be obtained with the *QuantAudit names* command.

Example

```
QuantAudit add Car.v -time 0-end
QuantAudit add Car.ax -dist 0-end
QuantAudit dump
    Car.ax      -time 0-end
    Car.v       -dist 0-end
```

17.4.10 Data Storage

The quantities which CarMaker write to a simulation results file are called output quantities. They are stored in the *Data/Config/OutputQuantities* file that is read by the CarMaker simulation program at the beginning of a TestRun.

Result storage itself can be done in number of ways. Save all results, save only the last 30 seconds, etc. Saving can be done during a simulation or after the simulation stopped.

Finally results files can be read into memory for postprocessing purposes.

OutQuantsAdd – Add to output quantities

Syntax

```
OutQuantsAdd quantity_list
OutQuantsAdd quantity ?quantity...?
```

Description

Adds quantities to the list of output quantities that will be buffered by CarMaker and saved to the result file when requested. Quantities may be specified either as a list or individually, but not both ways at the same time.

Examples

```
set quants {Time Car.v Car.ax Car.ay}
OutQuantsAdd $quants

OutQuantsAdd DM.Gas DM.Cluch
```

OutQuantsDel

Syntax

```
OutQuantsDel quantity_list
OutQuantsDel quantity ?quantity...?
```

Description

Deletes the specified quantities from the *OutputQuantities* file. Quantities may be specified either as a list or individually, but not both ways at the same time.

Examples

```
set quants {Time Car.v Car.ax Car.ay}
OutQuantsDel $quants

OutQuantsDel DM.Gas DM.Cluch
```

OutQuantsDelAll

Syntax

```
OutQuantsDelAll
```

Description

Deletes all quantities from the *OutputQuantities* file.

OutQuantsRestoreDefs

Syntax

OutQuantsRestoreDefs

Description

OutQuantsRestoreDefs restores the default quantities defined in the *OutputQuantities.default* file. The default file is created the first time any change is made to *OutputQuantities* using one of these functions, and can be modified manually at any point after it was created.

OutQuantsGetFName

Syntax

OutQuantsGetFName

Description

Retrieves the currently active OutputQuantities filename. The returned path is relative to Data/Config.

OutQuantsSetFName

Syntax

OutQuantsSetFName *outquants_filename*

Description

Sets the currently active OutputQuantities filename to the given filename which is interpreted relative to Data/Config.

Examples

```
OutQuantsSetFName MyOutputQuantities
```

SaveMode

Syntax

SaveMode *mode*

Description

Specifies the saving mode that is entered when a save is started with the *SaveStart* function. Legal values for *mode* are:

Mode	Description
collect (or: collect_only)	Collect only mode, no automatic saving to disk in the background. The data is collected in the results buffer and can be written later interactively or by using the <i>SaveStart</i> command. ("do not save" is a synonym for compatibility reasons, which should no longer be used)
save (or: save_all)	Save all results to disk. The data saving is started immediately when the simulation starts and is done in the background while the simulation is running.
hist_10s	Save the last 10 seconds
hist_30s	Save the last 30 seconds
hist_60s	Save the last 60 seconds
hist_max	Save all results recorded in the results buffer

This function corresponds to the CarMaker GUI's *Mode* menubutton in the *Storage of Results* part of the GUI's main window.

Example

```
SaveMode hist_10s
```

SaveStart

Syntax

```
SaveStart
```

Description

Starts saving the simulation results to a results file.

This function corresponds to the CarMaker GUI's *Save* button in the *Storage of Results* part of the GUI's main window.

SaveStop

Syntax

```
SaveAbort
```

Description

Stops the saving of the simulation results previously initiated with the *SaveStart* command. No more data is written to the results file but the file is kept.

This function corresponds to the CarMaker GUI's *Stop* button in the *Storage of Results* part of the GUI's main window.

SaveAbort

Syntax

SaveAbort

Description

Aborts the saving of the simulation results previously initiated with the *SaveStart* command. No more data is written to the results file and the file is deleted.

This function corresponds to the CarMaker GUI's *Abort* button in the *Storage of Results* part of the GUI's main window.

ImportResFile – Read a CarMaker results file

Syntax

ImportResFile *path channels varname ?spad?*

Description

Reads the values for the specified *channels*. Upon success, an array is created/modified with the name specified in the *varname* parameter. The array elements will be the channel names specified in the Tcl-list *channels*, and the element values will be the list of result values for the respective channel. To read all channels from the file without explicitly specifying them, pass “*” as the value for *channels*.

For example, if *varname* is “Results” and the channels read are “Car.v” and “DM.Gas”, the array created will be *Results* and it will contain two elements called *Results(Car.v)* and *Results(DM.Gas)*. The value of each element will be the channel’s list of values that is read from the result file.

Optionally a container *spad* for Scratchpad notes may be specified. Any Scratchpad data contained in the results file will be read into that container for later access with ScriptControl’s *Scratchpad* command. If the *spad* parameter is omitted no Scratchpad data will be read.

The command returns 0 on success and -1 in case of an error.

Examples

```
set fname SimOutput/rt1/20100504/Examples_Hockenheim_132054.erg
ImportResFile $fname {DM.Gas Car.vFL} Results

Log "Number of values in Car.vFL: [llength $Results(Car.vFL)]"
Log "Example value from Car.vFL: [lindex $Results(Car.vFL) 5]"

set fname2 SimOutput/rt1/20100504/Examples_VehicleDynamics_Slalom_18m_123456.erg
ImportResFile $fname2 * Results2 mySpad ;# Read _all_ channels and Scratchpad data

Log "Number of channels in file $fname2: [llength [array names Results2]]"
Log "Number of pylons hit: [llength [Scratchpad list mySpad::PylonHit]]"
```

SetResultFName

Syntax

```
SetResultFName fname
```

Description

Sets the filename of result files created during subsequent data storing operations. *fname* can be an absolute or relative path and may contain one or more of the following %-macros expanded right before the file is created:

%o – Output directory. Default: *SimOutput*.
%h – Hostname. Name of the computer where the simulation program is running.
%r – Result folder. Default: Value of %h.
%u – User name.
%f – TestRun file. Example: *Braking*.
%t – TestRun path, with mapped separators. Example: *Examples_Driving_Braking*.
%v – Variation. The current test series parameter variation.
%V – Vehicle path, with mapped separators: Example: *Examples_DemoCar*.
%W – Vehicle file. Examples: *DemoCar*.
%D – TestRun start date (yyyymmdd).
%T – TestRun start time (hhmmss).
%d – Current date (yyyymmdd).
%c – Current time (hhmmss).
%s – Sequence number. Expands to an empty string for the first result file created during a TestRun, and to “nnn” for subsequent result files during that TestRun. It will always expand to “nnn” for every resultfile created during a TestRun, if the Data/Config/SimParameters file contains an entry *DStore.ForceSequenceNo = 1*.
%% – The percent character “%”.
%?_s – Conditional output of a separation character. The separation character (“_” in this case, but any other may be specified) will be part of the expanded string if, and only if the specified %-macro (%s in this case, but any other may be specified) does not expand to an empty string. Examples: %?_s , %?+s.

The current default result filename is “%o/%r/%D/%t_%T%?_s”. This expands to paths like the following ones:

```
SimOutput/rt1/20080620/Examples_Driving_Braking_114215.erg  
SimOutput/rt1/20080620/Examples_Driving_Braking_114215_001.erg
```

The default result filename can be changed with a *DStore.OutPath* entry in the *Data/Config/SimParameters* file.

If an empty string “” is passed for *fname*, the result filename is reset to its default.

After data storing the resulting result filename with expanded %-macros can be retrieved using the *GetLastResultFName* command.

Example

```
SetResultFName "%o/%r/%D/%t_%T%?_s"
```

GetLastResultFName

Syntax

GetLastResultFName

Description

Returns the name of the last results file written by CarMaker during your current CarMaker session.

Note: At least one simulation producing a results file must have been run. Otherwise a warning is issued and the function returns an empty string.

Example

```
set fname [GetLastResultFName]  
ImportResFile $fname {DM.Gas Car.vFL} Results
```

17.4.11 Direct Variable Access (DVA)

Direct Variable Access (DVA) allows quantities to be modified during a simulation. The commands listed below are used for DVA with ScriptControl. See also the description of the corresponding mini-maneuver commands in the Reference Manual.

DVAWrite

Syntax

```
DVAWrite name ?val0? ?count? ?mode? ?val1? ?val2?
```

Description

Change a quantity's value specified by *name* and according to *mode* and parameters *val0*, *val1* and *val2* given, for a duration of *count* cycles.

The *mode* parameter determines how the quantity's value should be manipulated. Possible values for *mode* are:

Mode	Description
Abs	Absolute value <i>val0</i>
Off	Offset <i>val0</i>
Fac	Factor <i>val0</i>
FacOff	Factor <i>val0</i> , offset <i>val1</i>
AbsRamp	Absolute value <i>val0</i> , ramp during <i>val1</i> cycles
OffRamp	Offset <i>val0</i> , ramp during <i>val1</i> cycles
FacRamp	Factor <i>val0</i> , ramp during <i>val1</i> cycles
FacOffRamp	Factor <i>val0</i> , offset <i>val1</i> , ramp during <i>val2</i> cycles

Except for *name* all parameters are optional. Defaults are *val0* = 0.0, *count* = 1, *mode* = Abs, *val1* = 0.0, *val2* = 0.0.

It should be noted that modes applying an offset or a factor to a quantity only work right if the quantity is reset/recalculated during each cycle. Otherwise unexpected behaviour may result, e.g. exponential growth of a quantity if a factor is applied.

Example

```
DVAWrite DM.Steer.Ang 5 100
```

DVAReleaseQuants

Syntax

```
DVAReleaseQuants
```

Description

Releases the quantities from DVA control. The quantities are not reset, and unless they are modified by some internal function, they will keep the value they were given.

17.4.12 Executing Matlab Commands

MatlabEval – Execute Matlab Command

Note: This command is only available in CarMaker for Simulink.

Syntax

```
MatlabEval ?options? cmd ?arg...?
```

Description

Execute a Matlab command *cmd*, possibly with one or more arguments *arg*, in the running Matlab interpreter currently connected to the CarMaker GUI.

cmd and all *arg(s)* will be concatenated using Tcl's *join* command before being sent to the Matlab interpreter, without further treatment of interpretation. As in normal Matlab command execution a semicolon (";") at the end of a command must be used if output of the result in the Matlab console is to be suppressed.

Possible options are:

- async** – Return immediately, do not wait for completion of the command.
- timeout *ms*** – Wait for completion of the Matlab command, but at most *ms* milliseconds. Negative values mean "wait forever", a zero value means "don't wait", i.e. -async.

Default behaviour, if no option is specified, is to wait until completion of *cmd* (with no timeout, i.e. possibly forever).

On success, the function returns the command's result as a string or the error message issued by Matlab if the command failed. If the command did not produce a result in Matlab (like e.g. an assignment to a variable), an empty string is returned. On timeout or communication failure, the result will be an empty string, too.

Limitations

When returning the result of a command, *MatlabEval* currently supports only a limited subset of data types and dimensions: Scalars, vectors, and simple character arrays (strings).

The maximum allowed length of *cmd* as well as the length of the returned result is limited by the maximum length of an APO application defined message, i.e. about 700 characters. If *cmd* is too long it will be truncated in length.

Internally, when executing a command, *MatlabEval* manipulates the special workspace variable *ans* as needed. As a consequence, commands issued via *MatlabEval* should not expect to find the result of a previously issued command in *ans* – simply do not use *ans*.

Examples

```
MatlabEval "alpha = 42.0;"  
set a [MatlabEval alpha]  
MatlabEval "SomeFunction(3.1, 2.7);"
```

MatlabEvalCancel – Cancel Matlab Command Execution

Note: This command is only available in CarMaker for Simulink.

Syntax

```
MatlabEvalCancel ?result?
```

Description

Force the *MatlabEval* function currently running (waiting for the Matlab interpreter to finish the execution of a Matlab command) to stop waiting and return immediately. If no *MatlabEval* is currently running just do nothing.

Please note that there is no way to interrupt Matlab's execution of the command itself, i.e. the Matlab interpreter will continue to work on the current command until it completes.

An optional *result* argument may be specified as the value to be returned by *MatlabEval*. If not specified, *MatlabEval* will return an empty string.

Example

```
MatlabEvalCancel  
MatlabEvalCancel "CANCELED"
```

17.4.13 Application commands

Application connect

Syntax

Application connect ?*target*?

Description

Connects to already running application. Optional argument target can be used to specify a specific host and/or process. Valid values are:

hostname

hostname:pid (pid = process id)

192.168.0.1 (IP address)

192.168.0.1:655

localhost

same (same as previously connected host)

Note: not all HIL systems support all forms.



Application disconnect

Syntax

Application disconnect

Description

Disconnects the running application.

Application isconnected

Syntax

Application isconnected

Description

Returns 1 if connected, 0 otherwise.

Application exe(cutable)

Syntax

Application exe ?*name*?

Description

Returns current/previous executable name. If *name* is specified, sets executable name, as shown in application dialog.

Application opt(options)

Syntax

Application opt ?*optionstring*?

Description

Returns current/previous value. If *optionsstring* is specified, sets the command options, as shown in application dialog.

Application start

Syntax

Application start

Description

Starts (on some HIL platforms: downloads and starts) the application and also tries to connect to the application. Same as Start button in the application dialog.

Application stop

Syntax

Application stop

Description

Terminates the application. Same as HIL-System / Terminate Application in the application dialog.

Application shutdown

Syntax

Application shutdown

Description

HIL: shutdown/reboot of the target system. Same as HIL-System / Shutdown System in the application dialog.

Application appinfo

Syntax

Application appinfo

Description

Returns name and version info of current application.

Example return value: CarMaker 5.0 - Car_Generic (build 1380)

Application cmversion

Syntax

Application cmversion

Description

Returns CarMaker version and the release date of this version.

Example return value: 5.0 2015-06-11 13:36:42

Application compileinfo

Syntax

Application compileinfo

Description

Returns some build infos of current: user, host, date, time.

Example return value: walter saturn.ipg 2015-06-11 13:37:17

Application target

Syntax

Application target ?*host*?

Description

CM/HIL only: Returns current setting of target system name. If optional argument *host* is given, sets name of target system. Same as Target System entry field in the application dialog.

Application consoleoutput

Syntax

Application consoleoutput

Description

CM/HIL only: Returns console output during application start as shown in the Application Console.

17.4.14 PopupCtrl Commands

The following commands handle the popup behavior which occur using batch operations like ScriptContol scripts, Testseries or Remote GUI control. They can be used to e.g. prevent the freeze of a testautomation due to error popups.



They do not affect the popups occurring from interactively using the GUI.

PopupCtrl timeout

Syntax

PopupCtrl timeout ?*time*?

Description

Sets or return the current timeout of popups occurring from batch operations. If the optional argument *time* is given, the timeout is set to *time* seconds and the previous timeout is returned. If the argument is not given, the current timeout value is returned. If a popup has choices the default answer is given.

Special values for *time*:

time = 0: Timeout after 0 seconds, the popup will not be shown

time = -1 Default value. Means no timeout forcing the user to interact manually.

PopupCtrl lastmsg

Syntax

PopupCtrl lastmsg

Description

Returns last popup message as a tcl-list: [*type* {*text in multiple lines*} *answer*]

Type: info, warn (warning), err (error), question, busy, or bug

Answer : index of the given answer (0 for first answer, 1 for second...)

PopupCtrl nextmsg

Syntax

PopupCtrl nextmsg

Description

Returns the next message in the message buffer as a tcl-list (same as for lastmsg) starting with the oldest message. A maximum of five messages will be buffered. A new message will dismiss the oldest message. After reading a message it is deleted from the message buffer.

PopupCtrl clearmsgs

Syntax

PopupCtrl clearmsgs

Description

Deletes all messages in the message buffer. The command nextmsg and lastmsg will return {} then.

17.4.15 Vehicle Operator / Power Control (KL15)

VhclOp

Syntax

VhclOp *cmd*

Description

Sends a vehicle operator message to the CarMaker simulation program with the command *cmd*. Following commands are supported:

Command	Description
disable	Disables the vehicle operator
enable	Enables the vehicle operator with the previous target operation state
absent	Enables the vehicle operator with the target operation state “Absent”
poweroff	Enables the vehicle operator with the target operation state “Power Off”
poweracc	Enables the vehicle operator with the target operation state “Power Accessory”
poweron	Enables the vehicle operator with the target operation state “Power On”
drive	Enables the vehicle operator with the target operation state “Drive”
user<i>	Enables the vehicle operator with any user defined target operation state “User<i>”

PowerOn

Syntax

PowerOn

Description

Sends a KL15-power-on message to the CarMaker simulation program.
This command was retained for compatibility to CarMaker version 4.5 with slightly modified functionality.
If the vehicle operator is activated, the command sets the target operation state to “Power On”, otherwise the vehicle key is set to the position “Power On”.

PowerOff

Syntax

PowerOff

Description

Sends a KL15-power-off message to the CarMaker simulation program.

This command was retained for compatibility to CarMaker version 4.5 with slightly modified functionality.

If the vehicle operator is activated, the command sets the target operation state to “Power Off”, otherwise the vehicle key is set to the position “Power Off”.

TogglePower

Syntax

TogglePower

Description

Sends a KL15-toggle-power message to the CarMaker simulation program.

This command was retained for compatibility to CarMaker version 4.5 with slightly modified functionality.

If the vehicle operator is activated, the command sets the target operation state to “Power On” or “Power Off” depending on the previous state, otherwise the vehicle key is set to the position “Power On” or “Power Off” depending on the previous key position.

17.4.16 FailSafeTester Commands

FST_Init

Syntax

FST_Init

Description

Initializes all connected FailSafeTesters. Must be called before any other FST commands are used.

FST_Reset

Syntax

FST_Reset

Description

Resets all plugged in cards to their initial state.

FST_ChAdd, FST2_ChAdd, FST_ChAdd_TC, FST2_ChAdd_TC

Syntax

FST_ChAdd *group channel*
FST2_ChAdd *group channel*
FST_ChAdd_TC *time group channel*
FST2_ChAdd_TC *time group channel*

Description

Adds *channel* to *group*.

This in effect means: Connect this channel to each other channel which is in the same group.

The term “group” is synonymous with the term “interconnection lines” and can be visualized as a single wire running perpendicular to a channel/signal. When you add the channel/signal to the group/interconnection line the wires are connected. Any number of signals/channels can be added to the same group, effectively shorting the signals in a controlled way. If the group does not exist before, it is created.

A trailing “/i” or “/o” behind *channel* decides whether the connection is established on the input side or on the output side of the channel. If neither “/i” nor “/o” is given, CarMaker decides, where to establish the connection!

The functions without "_TC" also define the "original state" of the given channels for a subsequent call of any function ending with "_TC".

The functions ending with "_TC" are for temporarily configuration of channels and do the same as FST_ChAdd and FST2_ChAdd but only for the specified amount of time. After the specified time is over the configuration of this channel automatically returns to the "original state".

The reset to the previous state is done in the FailSafeTester and therefore also works if the CarMaker's executable has cancelled.

FST_xxx controls the first FST while FST2_xxx controls the second FST.

The "_TC"-functions only work correctly with the FST-CC2 with firmware revision >= 2.1.3.

FST_ChRem,
FST2_ChRem,
FST_ChRem_TC,
FST2_ChRem_TC

Syntax

FST_ChRem *group channel*
FST2_ChRem *group channel*
FST_ChRem_TC *time group channel*
FST2_ChRem_TC *time group channel*

Description

Removes the specified *channel* from a *group* of signals.

The term "group" is synonymous with the term "interconnection lines" and can be visualized as a single wire running perpendicular to a channel/signal. When you add the channel/signal to the group/interconnection line the wires are connected. Any number of signals/channels can be added to the same group, effectively shorting the signals in a controlled way.

A trailing "/i" or "/o" behind *channel* decides whether the connection is cut on the input side or on the output side of the channel. If neither "/i" nor "/o" is given, CarMaker decides, where to cut the connection!

The functions without "_TC" also define the "original state" of the given channels for a subsequent call of any function ending with "_TC".

The functions ending with "_TC" are for temporarily configuration of channels.

This functions do the same as FST_ChRem and FST2_ChRem but only for the specified amount of time. After the specified time is over the configuration of this channel automatically returns to the "original state". The original state is defined by the last call of FST_ChRem and FST2_ChRem.

The reset to the previous state is handled in the FailSafeTester and therefore also works if the CarMaker's executable has cancelled.

FST_xxx controls the first FST while FST2_xxx controls the second FST.

The "_TC"-functions only work correctly with the FST-CC2 with firmware revision >= 2.1.3.

FST_ChCut,
FST2_ChCut,
FST_ChCut_TC,
FST2_ChCut_TC

Syntax

FST_ChCut channel
FST2_ChCut channel
FST_ChCut_TC time channel
FST2_ChCut_TC time channel

Description

This command cuts the connection between the input connector and the output connector of *channel*.

The functions without "_TC" also define the "original state" of the given channels for a subsequent call of any function ending with "_TC".

The functions ending with "_TC" are for temporarily configuration of channels.

This functions do the same as FST_ChCut and FST2_ChCut but only for the specified amount of time. After the specified time is over the configuration of this channel automatically returns to the "original state". The original state is defined by the last call of FST_ChRem and FST2_ChRem.

The reset to the previous state is handled in the FailSafeTester and therefore also works if the CarMaker's executable has cancelled.

FST_xxx controls the first FST while FST2_xxx controls the second FST.

The "_TC"-functions only work correctly with the FST-CC2 with firmware revision >= 2.1.3.

FST_ChCon,
FST2_ChCon,
FST_ChCon_TC,
FST2_ChCon_TC

Syntax

FST_ChCon channel
FST2_ChCon channel
FST_ChCon_TC time channel
FST2_ChCon_TC time channel

Description

This command establishes the connection between the input connector and the output connector of *channel*.

The functions without "_TC" also define the "original state" of the given channels for a subsequent call of any function ending with "_TC".

The functions ending with "_TC" are for temporarily configuration of channels. This functions do the same as FST_ChCut and FST2_ChCut but only for the specified amount of time. After the specified time is over the configuration of this channel automatically returns to the "original state". The original state is defined by the last call of FST_ChRem and FST2_ChRem.

The reset to the previous state is handled in the FailSafeTester and therefore also works if the CarMaker's executable has cancelled.

FST_xxx controls the first FST while FST2_xxx controls the second FST.

The "_TC"-functions only work correctly with the FST-CC2 with firmware revision >= 2.1.3.

FST_GrpRem, FST2_GrpRem

Syntax

`FST_GrpRem group`
`FST2_GrpRem group`

Description

Removes all channels from *group* and then removes the group itself.

The term "group" is synonymous with the term "interconnection lines" and can be visualized as a single wire running perpendicular to a channel/signal. When you add the channel/signal to the group/interconnection line the wires are connected. Any number of signals/channels can be added to the same group, effectively shorting the signals in a controlled way.

FST_GrpRem controls the first FST, FST2_GrpRem controls the second FST.

FST_SCAdd, FST2_SCAdd

Syntax

`FST_SCAdd sc group`
`FST2_SCAdd sc group`

Description

Adds signal connector *sc* on the Controller Card to *group*. Herewith the signal level on the interconnection Line can be measured.

FST_SCAdd controls the first FST, FST2_SCAdd controls the second FST.

FST_SCRem, FST2_SCRem

Syntax

`FST_SCRem sc group`
`FST2_SCRem sc group`

Description

Disconnects the signal connector *sc* on the Controller Card from *group*.

FST_SCRem controls the first FST, FST2_SCRem controls the second FST.

FST_SetRstr, FST2_SetRstr

Syntax

FST_SetRstr *rstr value*
FST2_SetRstr *rstr value*

Description

Sets resistor *rstr* to *value* on a programmable resistor card (PRC).

FST_SetRstr controls the first FST, FST2_SetRstr controls the second FST.

Please find further information about how to add a resistor to a group in the User's Guide, appendix "Minimaneuver Command Language - FailSafeTester Commands": "Resistors - Naming Conventions".

FST_ContactCare

Syntax

FST_ContactCare

Description

Initiates a contact care operation on all connected FailSafeTesters.

FST_ReloadConfig

Syntax

FST_ReloadConfig

Description

This command causes CarMaker to reload the configuration of all FailSafeTesters manually.

17.4.17 IPGDriver Commands

Driver importparameters – Import driver parameters from TestRun

Syntax

Driver importparameters *path*

Description

Import driver parameters from the TestRun specified by *path*. Specification of a relative path causes the TestRun to be loaded relative to the *Data/TestRun* directory of your CarMaker project directory.

On success the function returns 0, otherwise -1 is returned.

Example

```
Driver importparameters "Examples/VehicleDynamics/Braking"
```

Driver importknowledge – Import driver knowledge from TestRun

Syntax

Driver importknowledge *path*

Description

Import driver knowledge from the TestRun specified by *path*. Specification of a relative path causes the TestRun to be loaded relative to the *Data/TestRun* directory of your CarMaker project directory.

On success the function returns 0, otherwise -1 is returned.

Example

```
Driver importknowledge "Examples/VehicleDynamics/Braking"
```

Driver clearknowledge – Clear driver knowledge in current TestRun

Syntax

Driver clearknowledge

Description

Clear all existing driver knowledge in the currently loaded TestRun.

Example

```
Driver clearknowledge
```

Driver adaptbasicknowledge – Basic driver adaption

Syntax

Driver adaptbasicknowledge *doVehicleLimits doControllerDynamics ?doGearSwitching?*

Description

Based on the currently loaded vehicle data perform a driver adaption acquiring basic knowledge about the vehicle. The boolean parameters *doVehicleLimits*, *doControllerDynamics* and *doGearSwitching* correspond to the similarly named checkboxes in the **Driver Adaption** dialog of the CarMaker GUI.

On success the function returns 0, otherwise -1 is returned.

Examples

```
Driver adaptbasic 1 1
```

Driver adaptracedriver – Race driver adaption

Syntax

Driver adaptracedriver *learningrate*

Description

Based on the currently loaded vehicle data perform a race driver adaption. The numeric *learningrate* parameter corresponds to the similarly named entry field in the **Driver Adaption** dialog of the CarMaker GUI.

On success the function returns 0, otherwise -1 is returned.

A common cause of failure is that the currently loaded TestRun does not contain basic knowledge learned during a previously performed basic driver adaption.

Examples

```
Driver adaptracedriver 0.75
Driver adaptrace .5           (abbreviated form)
```

17.4.18 IPGMovie Commands

Movie attach

Syntax

Movie attach

Description

This command establishes a connection to an already running and not connected IPGMovie process.

Movie background select

Syntax

Movie background select *backgroundName*

Description

This command activates the background with name *backgroundName*. Background names that contain a space should be defined like "background name with space".

Movie camera select

Syntax

Movie camera select *cameraName* ?*options*?

Description

In its simplest form, this command activates the camera named *cameraName* within the currently active view (the view with the ipg-logo) in the active IPGMovie-window. Other views can be defined with the option *-view viewId*. Other windows can be defined with the option *-window windowId*. If the window specified with *-window windowId* doesn't exist, the command has no effect. If another window is specified with the option *-window windowId* but no view within this window is specified with the option *-view viewId*, the view with *viewId* 0 is used. Camera names that contain a space should be specified like \"camera name with space\"

The following options can be used:

- window *windowId*** Specifies the relevant IPGMovie-window that is used for this command. *WindowId* is the id of the window (0 for the main window, 1 for the first additional window etc.)
- view *viewId*** Specifies the relevant IPGMovie-view ID that is used for this command. *ViewId* is the id of the view (0 for the main view, 1 for the first additional view etc.)

Example

```
Movie camera myPredefinedCamera -window 1 -view 2
```

Movie detach

Syntax

Movie detach

Description

Cuts the connection between the CarMaker GUI and IPGMovie. Prevents, for example, that IPGMovie gets closed when the CarMaker GUI is closed.

Movie export

Syntax

Movie export *targetFormat* *targetPath* ?*srcType* *srcTypeInfo*? ?*options*?

Description

This command can be used for exporting images, image sequences or videos of a specified IPGMovie-window, a vds-channel or a 3d-object. In the following, the input parameters are explained:

targetFormat Specifies the format of the created image(s) or video. “Table: Export formats” shows all supported format-strings that can be used as a value for *targetFormat*

targetPath full path (including directory, filename and file extension) of the created image or video

?srcType srcTypeInfo? This combination of parameters defines what will be exported. The following table shows what has to be specified for *srcType* and *srcTypeInfo*

srcType	srcTypeInfo	When to use
3dobject	full path including directory, file name and file extension to created image / video	creating images from 3dobjects
vds	id of vds-channel	creating images, image sequences or videos of a vds-channel
window	id of window	creating images, image sequences or videos of the specified Movie window

?options? for srcType vds:

Option	Description
-vds <i>vdsChannelId</i>	Specification of the vds channel that is used for image / video export.

?options? for srcType window:

none

?options? for *srcType* vds and window:

Option	Description
<code>-start startTimeInSeconds</code>	Specification of the start time for the video or image / image sequence export. <i>StartTimeInSeconds</i> is the simulation-Time.
<code>-end endTimeInSeconds</code>	Specification of the end time for the video or image / image sequence export. <i>EndTimeInSeconds</i> is the simulation-Time.
<code>-width widthInPixels</code>	Horizontal resolution of the created image(s) or video.
<code>-height heightInPixels</code>	Vertical resolution of the created image(s) or video.
<code>-framerate fps</code>	Specification of the framerate in case of video export or image sequence export
<code>-datarate KByte/s</code>	Specification of the maximum datarate

?options? for *srcType* 3dobject:

Option	Description
<code>-bbox {length width height}</code>	Specification of the bounding box of the 3d-object which is used for creating the images. <i>length</i> , <i>width</i> , <i>height</i> are expected to be object length [m], object width [m] and object height [m] (in this order). Default: 25 x 25 x 25 m.
<code>-view viewType</code> or <code>-view {viewType1 viewType2 ...}</code>	Specification which of the possible predefined camera views you want the object images to be taken. <i>ViewType</i> can have the single values <i>top</i> , <i>side</i> , or <i>front</i> or a combination thereof. Default: <i>{top side}</i> .
<code>-projectionType projType</code>	Specification which of the 2 possible predefined projection types (orthogonal and projection) you want the images to be taken. <i>projType</i> can have the values " <i>perspective</i> ", " <i>orthographic</i> " or <i>{perspective orthographic}</i> . Default: orthographic.
<code>-pivotPos posType</code>	Specification of the 2d-position of the 3d-object-coordinate systems' origin in the rendered image. Possible values for <i>posType</i> are: "bottomleft", "bottommiddle", "bottomright", "center", "topleft", "topmiddle" and "topright". Default: center.
<code>-topSuffix val</code> <code>-sideSuffix val</code> <code>-orthoProjTypeSuffix val</code> <code>-perspProjTypeSuffix val</code>	These options define the suffixes in the filename for "-view top", "-view side", "-projectionType ortho" and "-projectionType persp". <i>Val</i> can be any string, but "." is used to define an empty string (= no suffix in filename). Default: -topSuffix persp, -sideSuffix ortho.
<code>-zoomFactor val</code>	This option is a possibility to zoom in/away to/from the object, resulting in a higher/lower image resolution. A zoom-Factor of 1.0 results in no zooming. A factor smaller than 1.0 defines a "zoom in" and a factor greater than 1.0 defines a "zoom out". Default: 1.0.

Option	Description
-recursive val	Specification if the directory defined with <i>srcTypeInfo</i> should be searched for 3dobjects recursively, that means subfolders, sub-sub-folders etc. are also searched for 3dobjects. Val can be "true", "false", "1" or "0". This option only makes sense if <i>srcTypeInfo</i> defines a directory. Default: 0.

Table: Export Formats

Table 17.1:

targetFormat	Type	OS	
avi-mpeg4	video	linux	AVI DivX 5 (ISO MPEG-4), best compression, excellent quality
avi-mpeg4v3	video	linux	AVI DivX 3 (MPEG-4 v3), high compression, very high quality
wmv1	video	linux	MS WMV7 (WMV1), high compression, very high quality
wmv2	video	linux	MS WMV8 (WMV2)
mpeg2	video	linux	MPEG-2 native, medium compression, high quality
avi-mjpg	video	linux	AVI MJPEG medium compression, often used for digital video editing
flv	video	linux	Flash Video
avi-divx	video	win*	AVI DivX 5 (needs 3rd party codec), best compression, excellent quality
avi-xvid	video	win*	AVI XviD (needs 3rd party codec), best compression, excellent quality, DivX compatible
avi-mjpg	video	win*	AVI MJPEG (needs 3rd party codec), medium compression, often used for digital video editing
avi-iv50	video	win*	AVI Indeo Video 5, high compression/quality
jpeg	image	all	JPEG images, numbered sequence of JPEG image files
png	image	all	PNG images, numbered sequence of PNG image files
ppm	image	all	PPM images (24bit, true color), numbered sequence of PPM image files (uncompressed)
pgm-g8	image	all	PGM Greyscale (8bpp), grey = 0.3*red + 0.59*green + 0.11*blue
pgm-g16	image	all	PGM Greyscale (16bpp), grey = 0.3*red + 0.59*green + 0.11*blue
pgm-z	image	all	PGM Depth images (z buffer), depth image (16bpp, z buffer values)
pgm-zl	image	all	PGM Depth images (linear), depth image (16bpp, linear, resolution 0.01m)
pgm-zf	image	all	PGM Depth images (float), depth image (float values)

win* = VideoForWindows (VfW) Codecs need to be installed

Example

```
Movie export png C:\Test\testImage.png  
Movie export png C:\Test\testImage.png window 2  
Movie export png C:\Test\testImage.png 3dobject C:\3DObjects\testObject.mobj
```

Movie fog

Syntax

Movie fog *mode ?options?*

Description

This command enables, configures or disables fog in IPGMovie. Mode can have the values *off*, *linear*, *exp* or *exp2*. *Each mode allows the declaration of different options.*

The following options can be used:

Mode off:

No options

Mode linear:

Option	Description
-start <i>nearDistanceVal</i>	Specifies the distance between camera and start of fog. <i>NearDistanceVal</i> is a value in meters
-end <i>endDistanceVal</i>	Specifies the distance between the camera and end of fog. <i>EndDistanceVal</i> is a value in meters.
-duration <i>timeInSeconds</i>	Specification of the time duration that the fog needs to interpolate from the current configuration to the target configuration (defined with -start <i>nearDistanceVal</i> and -end <i>endDistanceVal</i>). <i>TimeInSeconds</i> defines this value in seconds. The default value for <i>timeInSeconds</i> is 0, that means the new fog configuration instantly becomes active.

Mode exp and exp2:

Option	Description
-density <i>densityVal</i>	Specifies the density of the fog. The value range of <i>densityVal</i> is limited to float values between 0 and 1
-duration <i>timeInSeconds</i>	Specification of the time duration that the fog needs to interpolate from the current configuration to the target configuration (defined with -density <i>densityVal</i>). <i>TimeInSeconds</i> defines this value in seconds. The default value for <i>timeInSeconds</i> is 0, that means the new fog configuration instantly becomes active.
-color <i>colorVal</i>	Specifies the color of the fog. <i>colorVal</i> is defined as list { <i>redVal greenVal blueVal</i> }

Example

```
Movie fog off  
Movie fog linear -start 10 -end 30 -duration 5  
Movie fog exp -density 0.2 -duration 4 -color {1 0 0}
```

Movie help

Syntax

```
Movie help
```

Description

This command returns a list of all IPGMovie-ScriptControl-commands in alphabetic order.

Movie loadsimdata

Syntax

```
Movie loadsimdata erg_file ?options?
```

Description

This command does the same as what the menu item *File > Load > Complete Simulation...* does: IPGMovie tries to rebuild a complete simulation environment (testrun, road, 3D objects, etc.), starting from the data available in the ERG file.

The following options can be used:

- | | |
|--------------------------------|--|
| -projectDir <i>path</i> | The path to a project directory, where specific and needed information may be available (such as 3D objects, for instance) |
| -installDir <i>path</i> | The path to an installation directory of CarMaker,, where specific and needed information may be available. |

Movie quit

Syntax

```
Movie quit
```

Description

Quits Movie

Movie start

Syntax

```
Movie start ?args?
```

Description

This command starts IPGMovie with the specified command line parameters

Movie settim

Syntax

Movie settim *time*

Description

Sets the simulation time when in playback (has no effect in online mode, during a simulation). This command can also be used to set the distance when in road preview mode.

Movie sun position

Syntax

Movie sun position *mode ?options?*

Description

This command can be used for positioning the sun in IPGMovie. The parameter *mode* can have the following values: *attachedToCam*, *angle*, *geographic* and *demo*. Each mode allows the declaration of different options. Options are processed in the order they are specified. This might be important for options that overwrite parameters other options already set.

Movie sun position attachedToCam sets the sunlight to the camera position. When the camera moves, the sunlight inherits the position (there is a small offset to the exact camera position) as long as the active positioning mode is *attachedToCam*. Therefore this mode can be compared to a helmet light or flashlight.

Movie sun position angle allows the positioning of the sun via specification of azimuth and elevation angle.

Movie sun position geographic allows the positioning of the sun via specification of world time, latitude and longitude.

Movie sun position demo starts a demonstration-/presentation mode that automates the modification of the daytime and therefore shows the course of the sun in time lapse.

The following options are available for the specified mode:

Mode attachedToCam:

no options

Mode angle:

Option	Description
-azimuth <i>azimuthVal</i>	Specification of the azimuth angle in degrees (value range from -180 to +180)
-elevation <i>elevationVal</i>	Specification of the elevation angle in degrees (value range from 0 to 90)

Mode geographic:

Option	Description
-day <i>dayVal</i>	Specification of the day in the format -day yyyy-mm-dd
-time <i>timeVal</i>	Specification of the time in the format -time hh:mm
-interpolationDuration <i>interpolationTimeVal</i>	Specification of the time duration that the sun needs to move from the current position to the specified target position (specified with -time timeVal). InterpolationTimeVal defines this value in seconds. The default value for interpolationTimeVal is 0, that means the sun instantly “moves” to the target location. This option only works in conjunction with -time timeVal.
-dst <i>dstVal</i>	Specification if the defined time (specified with -time timeVal) is interpreted as daylight saving time or as standard time. dstVal is an appropriate boolVal (true, false, 1, 0)
-utc <i>utcVal</i>	Specification of the utc-Offset (difference to the coordinated world time). utcVal is an integer value within the range from -12 to +14
-longitude <i>longitudeVal</i>	Specification of the longitude. LongitudeVal can be defined in decimal notation or in the form <degrees> [°] <minutes>'<seconds>"
-latitude <i>latitudeVal</i>	Specification of the latitude. LatitudeVal can be defined in decimal notation or in the form <degrees> [°] <minutes>'<seconds>"
-location <i>locationVal</i>	Specification of an existing location preset. LocationVal has to be an existing location-string as it can be found in IPG-Movie Sun GUI in the location dropdown menu. The specification of this option sets the preferences connected with this location string. These are: name of location, latitude, longitude and utc-Offset.

Mode demo:

Option	Description
-stop	This option stops the demo mode

Example

```
Movie sun position attachedToCam
Movie sun position angle -azimuth 40 -elevation 25
Movie sun position geographic -day 2015-02-14 -time 17:14 -longitude 42°15'2" -latitude 45°16'22"
```

Movie version

Syntax

Movie version ?options?

Description

In its simplest form this command returns the version number of IPGMovie.

In addition, this command provides the possibility to specify one of the following options resulting in different return values. A declaration of multiple options makes no sense for this command (in this case the first option that is specified is used).

The following options can be used:

- date: The return value is composed of the version date and version number separated by a space

Example

```
Movie version  
Movie version -files
```

Movie window create

Syntax

Movie window create ?options?

Description

This command creates and opens a new IPGMovie-window. In its simplest form the newly created window consists of exactly one view (no split-screen). IPGMovie supports up to 7 additional windows, so this command won't have any effect if there already exist 7 additional windows.

The following options can be used:

- columns *nColumns* Specifies the number of horizontal views within the newly created window
- rows *nRows* Specifies the number of vertical views within the newly created window

Example

```
Movie window create  
Movie window create -columns 3 -rows 2
```

Movie window delete

Syntax

Movie window delete *windowId*

Description

Deletes the window with id *windowId*

Example

```
Movie window delete 1
```

Movie window fullscreen

Syntax

Movie window fullscreen *mode* ?options?

Description

This command activates or deactivates fullscreen mode for an IPGMovie-window depending on the value of the parameter *mode*. *Mode* can be *on* or *off*. *Movie window fullscreen on* activates fullscreen mode and *Movie window fullscreen off* deactivates fullscreen mode, setting the window resolution and position back to the values they had before activating fullscreen mode. If not specified differently via the option *-window windowId*, the currently active IPGMovie-window (the window with the ipg-logo) is used for this command.

The following options can be used:

- window *windowId*** Specifies the relevant IPGMovie-window that is used for this command. *WindowId* is the id of the window (0 for the main window, 1 for the first additional window etc.)

Example

```
Movie window fullscreen on  
Movie window fullscreen off -window 2
```

Movie window position

Syntax

Movie window position *xOffset* *yOffset* ?options?

Description

In its simplest form this command sets the position of the currently active IPGMovie-window (the window with the ipg-logo) but it also allows you to specify another already existing window for positioning instead via the option *-windowId windowId*. The parameters *xOffset* and *yOffset* are integer values that define the number of pixels between the left edge of the screen and the left edge of the relevant IPGMovie-window, respectively between the upper edge of the screen and the upper edge of the IPGMovie-window.

The following options can be used:

- window *windowId*** Specifies the relevant IPGMovie-window that will be positioned. *WindowId* is the id of the IPGMovie-window (0 for the main window, 1 for the first additional window etc.)

Example

```
Movie window position 0 0  
Movie window position 100 20 -window 1
```

Movie window size

Syntax

Movie window size *width height ?options?*

Description

This command sets the resolution of an IPGMovie window. In its simplest form, the currently active window (the window with the ipg-logo) is set to the resolution defined with the parameters *width* (*window width in pixels*) and *height* (*window height in pixels*). Other already existing IPGMovie-windows can be specified for a resolution change with the option **-window <windowId>**.

The following options can be specified:

- window *windowId*:** Defines the relevant IPGMovie-window for the resolution change. *windowId* is the id of the IPGMovie-window (0 for the main window, 1 for the first additional window etc.)

Example

```
Movie window size 640 480  
Movie window size 1920 1080 -window 1
```

Movie window split

Syntax

Movie window split *nColumns nRows ?options?*

Description

This command reconfigures the number of views within an IPGMovie-window. The parameters *nColumns* and *nRows* define the number of horizontal and vertical views. In its simplest form, the currently active IPGMovie-window (the window with the ipg-logo) is used for this command.

The following options can be used:

- window *windowId*:** Defines the relevant window for this command. *Windold* is the id of the IPGMovie-window (0 for the main window, 1 for the first additional window etc.)

Example

```
Movie window split 3 2  
Movie window split 2 1 -window 3
```

17.4.19 Concerto Commands

The *Concerto* command makes it possible to use CONCERTO as a postprocessing tool in a test automation process, e.g. with CarMaker's TestManager, running CONCERTO script commands for powerful evaluation of simulation results.

Concerto eval - Evaluate CONCERTO script

Syntax

```
Concerto eval ?options value ...? ?ConcertoScriptCmds?
```

Description

This command initiates CONCERTO to run a dynamically created script carrying out the following steps:

- Change the CONCERTO *WorkEnvironment* and read the *WorkEnvironmentData.dxv* file.
- Load the layout given by the option *Layout*.
- Load the latest result file.
- Evaluate *ConcertoScriptCmds*.
- Finally Concerto will delete the script file.

The *WorkEnvironmentData.dxv* file is also dynamically created. In the CarMaker environment, this file is used for the automatic creation of CONCERTO Data Sources.

The following options can be used to override actual CONCERTO settings:

- Layout:** Specifies an existing layout to load. The layout file can be given with an absolute path or relative to the CONCERTO *work environment* directory. If not specified, no layout will be loaded.
- File:** Specifies the data file to load. This file can be given with an absolute path or relative to the CarMaker *project* directory. Multiple files can be loaded at the same time. They have to be divided by a space character. If not specified, the latest results file found below the *DS* directory is loaded.
- DS:** Specifies a directory for the search of CONCERTO *Data Sources*. All directories containing .erg-files found below this directory will be available as *Data Source* in CONCERTO. If not specified, *SimOutput* will be searched for Data Sources.
- Id:** Specifies the CONCERTO instance to use for the evaluation of the script. If the given instance does not exist, a new CONCERTO instance will be started. If not specified, the identifier is set to *CM*.

With *ConcertoScriptCmds* a set of CONCERTO script commands can be passed to CONCERTO.

If no *options* or *ConcertoScriptCmds* are given, CONCERTO will at least change to the *WorkEnvironment* inside the CarMaker project directory and load the latest results file found in the directory given by *DS*.

Examples

```
Concerto eval
```

```
Concerto eval -Layout Layouts/VehicleDynamics/Braking.cly -File SimOutput/Braking.erg
```

Useful hints for CONCERTO scripting

- Make sure that strings in the created CONCERTO script are embedded in quotation marks.
- Use MS Windows path notation, i.e. with backslash \ as the directory separator.

- Use absolute paths rather than relative paths.
- Put the *ConcertoScriptCmds* in curly braces { } to avoid problems with Tcl's backslash and command (squared brackets) substitutions.

Example

```
Concerto eval { MsgBox("Hello World!") }
```

Concerto exit – Close CONCERTO

Syntax

```
Concerto exit ?-Id name?
```

Description

Closes CONCERTO.

To close a certain CONCERTO instance the option *-Id* can be used. If *-Id* is not specified then the CONCERTO instance with the default identifier *CM* will be closed.

Examples

```
Concerto exit
```

```
Concerto exit -Id MyConcerto
```

17.4.20 Test Manager commands

The *TestMgr* command provides access to the CarMaker Test Manager. It allows the user to define, manage and run test series from ScriptControl level. In addition to the available commands the *PopupCtrl* command may be used to prevent undesired user interaction during the automation process.

The following subcommands are provided:

TestMgr new - Create a new test series

Syntax

```
TestMgr new ?-force?
```

Description

Creates a new empty test series. In case of an already loaded test series which has been modified before, the user is prompted to confirm the operation unless the *-force* flag is specified,

Example

```
TestMgr new -force
```

TestMgr load - Load a test series

Syntax

```
TestMgr load ?filename?
```

Description

Loads the test series with the given filename which will be treated relative to the subfolder *Data/TestRun* of the current project directory. If filename is omitted the user will be prompted to select the desired test series in a file browser window.

Example

```
TestMgr load "Examples/CarMakerFunctions/TestManager/Slalom.ts"
```

TestMgr save - Save a test series

Syntax

```
TestMgr save ?filename?
```

Description

Saves the test series to disk at the specified location. The given filename will be treated relative to the subfolder *Data/TestRun* of the current project directory. If filename is omitted the user will be prompted to select a location.

Example

```
TestMgr save "Examples/CarMakerFunctions/TestManager/Slalom.ts"
```

TestMgr start - Start a test series

Syntax

```
TestMgr start ?-async?
```

Description

Starts the currently selected test series. If the `-async` flag has been specified this command returns immediately and the test series is executed in background, otherwise the command blocks and returns after the test series has finished completely.

Example

```
TestMgr start
```

```
TestMgr start -async
```

TestMgr stop - Stop a running test series

Syntax

```
TestMgr stop ?-immediate?
```

Description

Stops the active test series. If the `-immediate` flag is specified the current TestRun is aborted immediately, otherwise the execution of the test series stops after the current TestRun has finished.

Example

```
TestMgr stop
```

```
TestMgr stop -immediate
```

TestMgr get - Retrieve general TestManager information

Syntax

```
TestMgr get option
```

Description

This command returns general TestManager information. The following options are currently supported:

FName	Returns the filename of the loaded test series
Description	Returns the description of the loaded test series
Status	Returns either <i>running</i> or <i>idle</i> depending on whether a test series is currently running or not.

ActiveItem In case of a running test series, returns the item number of the currently active item in the test series tree view. The item number can be used to retrieve additional information from the item.

Result Returns the result of the currently active TestRun or Variation. Values can be either *good*, *warn*, *bad*, *err*, *skip* or the empty string. This command may be used for example in a TestRun EndProc to retrieve the result of the just finished simulation.

Example

```
TestMgr get FName  
TestMgr get ActiveItem
```

TestMgr clearresults - Clears the results of a test series

Syntax

```
TestMgr clearresults ?itemNo ... ?
```

Description

This command clears the result information of the current test series. If no further argument is provided, the result information of the whole test series is cleared, otherwise only the specified items are affected.

Example

```
TestMgr clearresults  
TestMgr clearresults 4 5 8
```

TestMgr additem - Add a new item to the test series

Syntax

```
TestMgr additem type name ?args?
```

Description

This command adds a new item to the test series tree and returns the identification number of the new item. The insert position can be specified by the optional *-index* argument followed by an item number after which the current item should be inserted. If no insert position is specified the new item is appended to the last recent item.

Depending on the specified *type* the command behaves differently and supports varying arguments:

Group Add a new group item which starts a new level in the test series tree view. All following items are inserted within the new group

Example

```
TestMgr additem Group "My new group" -index 12
```

EndGroup Signal the end of a group and leaves the current level in the test series tree view. All following items are appended on the same level as the previous group item.

Example

```
TestMgr additem EndGroup
```

Cmd Add a new ScriptControl block *name* to the test series. The following options are supported by this command:

-*cmd* Followed by a proper list whose elements define the individual lines of the script

Example

```
TestMgr additem Cmd "My SC block" -cmd {{set MyVar 2} {Log $MyVar}}
```

Settings Add a new Settings block to the test series. The following options are supported by this command:

-*param* Followed by a parameter list whose elements are formed by triples describing the variable's name, its type and its value. For valid variable types refer to the respective chapter in the User's Guide.

Example

```
TestMgr additem Settings "My Settings" -param {{ScriptFile CM  
MyScript.tcl} {StartProc CM MyStartProc}}
```

```
TestMgr additem Settings "My Settings" -param {{MyParam NValue 12}}
```

TestRun Add a new TestRun item to the test series. The parameter *name* specifies the filename of the TestRun. This command supports the following additional options:

-*param* Followed by a parameter list whose elements are formed by pairs describing the variable's name and its type. For valid variable types refer to the respective chapter in the User's Guide.

Example

```
TestMgr additem TestRun "Examples/Braking" -param {{vel NValue}}
```

```
TestMgr additem TestRun "Examples/Braking" -param {{velocity NValue}  
{mue NValue}}
```

Variation Append a Variation to a TestRun. Note that this command only succeeds if the parent item is a TestRun item. The following options are supported:

-*param* Followed by a proper list whose elements are the individual values of the parameters defined for the adjacted TestRun.

Example

```
TestMgr additem Variation "Variation 1" -param {100 0.8}
```

Characteristic Append a Characteristic to a TestRun. This command only succeeds if the parent item is a TestRun item. The following options are supported:

- desc Define a description of this characteristic value.
- ident Specify the identifier of this characteristic value.
- unit Define the unit of this characteristic value.
- param Followed by a proper list whose elements are pairs of keys and values. Valid keys are *RTexpr*, *StartProc* and *EndProc*.

Example

```
TestMgr additem Characteristic "Max Speed" -desc "Calculates the  
maximum speed" -ident "MaxSpd" -param {{RTexpr  
"first() ?Qu::MaxSpd=0:MaxSpd=max(Car.v,MaxSpd)"}}  
TestMgr additem Characteristic "Max Speed" -ident "MaxSpd" -param  
{{StartProc PrepMaxSpeed} {EndProc CalcMaxSpeed}}
```

Criterion

Append a Criterion to a TestRun. This command only succeeds if the parent item is a TestRun item. The following options are supported:

- desc Define a description of this criterion.
- good The criterion result will be set to *good* if the expression specified by this option evaluates to true.
- warn The criterion result will be set to *warn* if the expression specified by this option evaluates to true.
- bad The criterion result will be set to *bad* if the expression specified by this option evaluates to true.

Example

```
TestMgr additem Criterion "Brake Distance" -desc "TestRun evaluation is  
carried out on the basis of Brake Distance. TestRun is  
considered good if Brake Distance < 14m." -good "[get  
BrakeDist] < 14.0" -warn "[get BrakeDist] >= 14.0 &&  
[get BrakeDist] < 17.0" -bad "[get BrakeDist] >= 17.0"
```

Diagram

Append a Diagram item to a TestRun. This command only succeeds if the parent item is a TestRun item. The following options are supported:

- mode Define the diagram mode. The supplied string must be either *Quantity vs Time*, *Quantity vs Quantity* or *Characteristic vs Variation*. Default is *Quantity vs Time*.
- type Define the diagram type. The supplied string must be either *Line Diagram*, *Point Diagram* or *Vertical Bar Diagram*. Default is *Line Diagram*.
- grid Specifies if a grid should be shown in the diagram. Valid values are *Horizontal*, *Vertical*, *Both* or *None*. Default is *None*.
- allvars Defines whether all variations should be plotted in one common diagram. Expects a boolean value (0 or 1).

-axisdata	Defines the properties of the axis tab. The argument is formed by a list consisting of two nested lists, where the first one describes the x axis and the second one the y axis. Each of these two lists holds exactly four elements, specifying the range (<i>Auto</i> or <i>Manual</i>), the start and end value and the label of the respective axis.
-contentdata	Defines the properties of the content tab. The argument is formed by a list consisting of two or more nested lists, where the first one describes the quantity on the x axis and the following ones the quantities on the y axis. Each of these lists holds exactly four elements, specifying the quantity name, its label in the diagram legend, followed by a conversion factor and a offset.
-reffile	Defines the absolute path of the reference file to be used.
-refdata	Defines the properties of the reference tab. The argument is formed by a list consisting of two or more nested lists, where the first one describes the quantity on the x axis and the following ones the quantities on the y axis. Each of these lists holds exactly four elements, specifying the quantity name, its label in the diagram legend, followed by a conversion factor and a offset.

Example

```
TestMgr additem Diagram "My Diagram" -mode "Quantity vs Time" -type  
"Line Diagram" -grid "Both" -axisdata {{{"Manual" 0 10  
""} {"Auto" "" "" ""}} -contentdata {{{"Time" "" "" ""}  
("Car.v" "" 3.6 "")}}
```

TestMgr deleteitem - Deletes an item from a test series

Syntax

```
TestMgr deleteitem itemNo
```

Description

This command deletes the given item from the test series. If the provided argument is not a valid item number within the test series' tree view an error is returned.

Example

```
TestMgr deleteitem 15
```

TestMgr itemget - Retrieve information for an item

Syntax

```
TestMgr itemget itemNo option
```

Description

This command returns detailed information on a specific item within the test series tree view. The following options are supported:

- kind** Returns the kind (*Settings*, *TestRun*, *Variation*,...) of the item.
- name** Returns the name of the item.
- parent** Returns the item number of the item's parent.
- next** Returns the item number of the item's next sibling. The return value may be empty if the specified item is the last item on its tree level.
- prev** Returns the item number of the item's previous sibling. The return value may be empty if the specified item is the first item on its tree level.
- firstchild** Returns the item number of the item's first child if it exists, otherwise returns an empty string.
- lastchild** Returns the item number of the item's last child if it exists, otherwise returns an empty string.
- children** Returns a list with the item numbers of the item's children.

Additionally, depending on the specific kind of the item, further options are supported. For a complete list of supported options please refer to the description of the *TestMgr additem* command.

Example

```
set childlist [TestMgr itemget 5 -children]
if {[TestMgr itemget 1 -kind] eq "Settings"} { set param [TestMgr itemget 1 -param] }
```

TestMgr itemconfigure - Configure an existing item

Syntax

TestMgr itemconfigure *itemNo option value*

Description

This command allows to configure some attributes of an item. The following options are supported:

- expand** Expand the view of the specified item. This option does not require an additional value.
- expandtree** Expand the whole subtree below the specified item. This option does not require an additional value.
- collapse** Collapse the view of the specified item. This option does not require an additional value.
- collapsetree** Collapse the whole subtree below the specified item. This option does not require an additional value.

Additionally, depending on the specific kind of the item, further attributes can be configured. For a complete list of supported attributes refer to the description of the *TestMgr additem* command.

Example

```
TestMgr itemconfigure 4 -expand
if {[TestMgr itemget 8 -kind] eq "Diagram"} { TestMgr itemconfigure 8 -grid Both }
```

TestMgr testlog - Add an entry to the TestManager TestLog

Syntax

TestMgr testlog *result description*

Description

This command adds an additional TestLog entry to the currently active item. This is only supported for TestRun and Variation items within a TestManager Start- or EndProc. It is possible to overwrite the internal result calculation for the simulation by specifying a non empty *result* argument. Valid values are *good*, *bad*, *warn*, *skip*, *err* or the empty string if the internal result should not be changed. The string defined as *description* is appended the the TestManager TestLog.

Example

```
TestMgr testlog "" "This is an informative TestLog entry"
```

```
TestMgr testlog "err" "Simulation failed because of ..."
```

TestMgr assocfile - Associate a file with the current item

Syntax

TestMgr assocfile *type filename*

Description

This command adds a file to the internal data of the currently active item. This is only supported for TestRun and Variation items within a TestManager Start- or EndProc. Type may be any kind of string, although *Image* and *Link* are the only types which are recognized at the moment.

Adding an arbitrary image this way allows it to appear in the generated report document of the TestSeries. Supported image formats are jpg and png.

Adding a link to an external file will integrate the link into the final report document. When clicking the link later, the system will try to open the link target with a matching default application.

Example

```
TestMgr assocfile Image "C:/Absolute/Path/To/Image.jpg"
```

```
TestMgr assocfile Image "Relative/Path/To/Image.png"
```

```
TestMgr assocfile Link "Relative/Path/To/MyDocument.doc"
```

17.4.21 Report commands

The *Report* command provides basic access to the CarMaker Report functionality. This allows the user to create his own reports in an automated way from ScriptControl level or from within a ScriptControl block of a CarMaker test series..

The *Report* command provides the following subcommands:

Report create - Create a new report

Syntax

Report create *filename*

Description

Creates a report of the loaded testseries by using the currently selected report template for layout and content definition. The given filename is relative to CarMaker's project directory.

Example

```
Report create MyReport.pdf
```

Report loadtemplate - Load a report template

Syntax

Report loadtemplate *filename*

Description

Loads the specified report template which defines the report's content and layout. If a template with the given name cannot be found an error is returned. The filename is relative to the Data/Config folder of the CarMaker project directory.

Example

```
Report loadtemplate MyTemplate.cmrep
```

Report configure - Configure a report's content

Syntax

Report configure ?*option value ...?*

Description

Allows the modification of basic information contained in the report. Must be called before "Report create". The following options are available:

- author Changes the author's name on the first page of the report.
- title Changes the title of the report which is displayed as header on each page.
- notes Modifies the notes section on the first page of the report.
- coverage Defines the coverage of the report. Possible values are *all* (cover all testruns in the current test series), *executed* (cover only the executed testruns) or *failed* (cover only the failed ones).

Example

```
Report configure -title "My Report Title" -coverage failed
```

Report savetemplate - Save a report template

Syntax

Report savetemplate *filename*

Description

Saves the specified report template which defines the report's content and layout. Previous changes made to the currently selected template via "Report configure" or the Report GUI are taken into affect. The filename is relative to the Data/Config folder of the CarMaker project directory.

Example

Report savetemplate MyTemplate.cmrep

17.4.22 INCA commands

The *INCA* command allows the user to access the third-party application INCA via its COM interface in an automated way. INCA is a widely-used tool for measurement, calibration and diagnostic tasks involving electronic control units and is developed by the ETAS Group for MS Windows operating systems. For further information about INCA please refer to the ETAS website and the original INCA documentation. Note that this command is only available on Windows systems.

The *INCA* command was tested with INCA V. 7.1 and provides the following subcommands which cover all basic tasks the user faces when working with INCA:

INCA StartApp - Start INCA

Syntax

INCA StartApp

Description

Establishes a connection to INCA via COM and opens the tool. The most recently used database and workspace are loaded automatically.

Examples

INCA StartApp

INCA CloseApp - Close INCA

Syntax

INCA CloseApp

Description

Closes all views and leaves INCA. After INCA was closed, it is essential to wait approximately 5 seconds before INCA is started again. If *INCA StartApp* is called during this time, the call will block until INCA is ready again.

Examples

INCA CloseApp

INCA OpenDataBase - Load a database

Syntax

INCA OpenDataBase *db_path*

Description

Opens an INCA database. The argument *db_path* must match the full path of the desired database within the filesystem hierarchy. If the specified database does not exist within the given path it will be created.

Examples

```
INCA OpenDataBase "C:\\\\ETASData\\\\INCA\\\\Database\\\\MyDataBase"
```

INCA GetAllWorkspaces - Return all workspace names

Syntax

```
INCA GetAllWorkspaces
```

Description

Return the names of all available workspaces in the current database.

Examples

```
INCA GetAllWorkspaces
```

INCA SelectWorkspace - Activate a workspace

Syntax

```
INCA SelectWorkspace ws_path
```

Description

Select a workspace and make it active. The specified workspace must exist in the current database and must be described by its full path within the database hierarchy.

Examples

```
INCA SelectWorkspace "Tutorial\\\\Workspace\\\\OneETK"
```

INCA GetAllExperiments - Return all experiment names

Syntax

```
INCA GetAllExperiments
```

Description

Return the names of all experiments which are available for the currently active workspace.

Examples

```
INCA SelectWorkspace "Tutorial\\\\Workspace\\\\OneETK"
```

INCA OpenExperiment - Open an experiment

Syntax

```
INCA OpenExperiment exp_path
```

Description

Select an experiment, connect it to the active workspace and open the experiment's view in a new window.

Examples

```
INCA OpenExperiment "Tutorial\\Experiment\\LambdaControl"
```

INCA CloseExperiment - Close an experiment

Syntax

```
INCA CloseExperiment
```

Description

Close an experiment view if one was opened before.

Examples

```
INCA CloseExperiment
```

INCA GetAllDevices - Return all device names

Syntax

```
INCA GetAllDevices
```

Description

Get the names of all available devices within the currently openend experiment.

Examples

```
INCA GetAllDevices
```

INCA SelectDevice - Select a device

Syntax

```
INCA SelectDevice dev_name
```

Description

Select the device specified by *dev_name*. The name of the device must match one of the device names returned by *INCA GetAllDevices*.

Examples

```
INCA SelectDevice "ETK test device:1"
```

INCA SwitchToWorkingPage - Change view

Syntax

INCA SwitchToWorkingPage

Description

Switch from reference page view to working page view for the currently opened experiment.

Examples

INCA SwitchToWorkingPage

INCA SwitchToReferencePage - Change view

Syntax

INCA SwitchToReferencePage

Description

Switch from working page view back to reference page view.

Examples

INCA SwitchToReferencePage

INCA GetAllMeasureVars - Return all measure variable names

Syntax

INCA GetAllMeasureVars

Description

Return the names of all available measure variables for the currently selected device.

Examples

INCA GetAllMeasureVars

INCA SelectMeasureVar - Select a measure variable

Syntax

INCA SelectMeasureVar *var_name*

Description

Select the specified measure variable and add it to the set of active variables. Only active variables can be visualized and captured by the recorder. The argument *var_name* must match the name of a measure variable within the currently selected device.

Examples

```
INCA SelectMeasureVar FRPS
```

INCA UnselectAllMeasureVars - Deselect all measure variables

Syntax

```
INCA UnselectAllMeasureVars
```

Description

Deletes all measure variables which were added by previous calls to *INCA SelectMeasureVar* from the set of active variables. After executing this command the respective variables can no longer be visualized or captured by the recorder.

Examples

```
INCA UnselectAllMeasureVars
```

INCA GetMeasureValue - Get value of measure variable

Syntax

```
INCA GetMeasureValue var_name
```

Description

Return the current value of the specified measure variable. The argument *var_name* must match the name of a measure variable within the currently selected device. In addition to that, a measurement process must be running for this call to succeed.

Examples

```
set value [INCA GetMeasureValue FRPS]
```

INCA GetAllCalibrationVars - Return all calibration variable names

Syntax

```
INCA GetAllCalibrationVars
```

Description

Return the names of all available calibration variables for the currently selected device.

Examples

```
INCA GetAllCalibrationVars
```

INCA SelectCalibrationVar - Select a calibration variable

Syntax

INCA SelectCalibrationVar *var_name*

Description

Select the specified calibration variable and add it to the set of active variables so that the variable can be modified by the user. The argument *var_name* must match the name of a calibration variable within the currently selected device.

Examples

```
INCA SelectCalibrationVar FRMAX
```

INCA UnselectAllCalibrationVars - Deselect all calibration variables

Syntax

INCA UnselectAllCalibrationVars

Description

Deletes all calibration variables which were added by previous calls to *INCA SelectCalibrationVar* from the set of active variables. After executing this command, the respective variables can no longer be modified.

Examples

```
INCA UnselectAllCalibrationVars
```

INCA GetCalibrationValue - Get value of calibration variable

Syntax

INCA GetCalibrationValue *var_name*

Description

Return the current value of the specified calibration variable.

Examples

```
set value [INCA GetCalibrationValue FRMAX]
```

INCA SetCalibrationValue - Set value of calibration variable

Syntax

INCA SetCalibrationValue *var_name value*

Description

Modify the value of the specified calibration variable. The argument *var_name* must match the type und format of the variable. This call will only succeed when no measurement is active at that time.

Examples

```
INCA SetCalibrationValue FRMAX 0.5
```

INCA ConfigureDataStorage - Modify data storage settings

Syntax

```
INCA ConfigureDataStorage filepath filename ?autoincrementflag?
```

Description

Modify the data storage settings for a measurement process. When calling the command with *autoincrementflag* = 1, an incremental number is appended to the specified filename during subsequent recording operations. The new settings will be applied from the next recording process on.

Examples

```
INCA ConfigureDataStorage "C:\\\\ETASData\\\\INCA" "MyFileName" 1  
INCA ConfigureDataStorage "D:\\\\ETASData" "RecFile"
```

INCA ConfigureRecorderStartCond - Set recorder start condition

Syntax

```
INCA ConfigureRecorderStartCond Trigger pretime_s formula  
INCA ConfigureRecorderStartCond Manual pretime_s  
INCA ConfigureRecorderStartCond NoDelay
```

Description

Before starting a capturing process, the recorder start condition must be specified with this command. It takes one of the following three forms:

Specifying *Trigger* as first argument will create a trigger which activates the recorder when a certain criteria is met. The criteria is given by *formula* which will be tested for its validity by the time of this call. With the argument *pretime_s* it is possible to specify the number of seconds which will be prepended to the actual recording start time

With *Manual* as first argument the recording will not start automatically. The recording will only start when *INCA StartRecording* is called. A *pretime* in seconds will be added before the actual call to *INCA StartRecording*.

Calling the command with the argument *NoDelay* will lead to an immediate recording start as soon as a measurement process is started.

Examples

```
INCA ConfigureRecorderStartCond Trigger 2 "TVLR > 850"  
INCA ConfigureRecorderStartCond Manual 0  
INCA ConfigureRecorderStartCond NoDelay
```

INCA ConfigureRecorderEndCond - Set recorder end condition

Syntax

```
INCA ConfigureRecorderEndCond Trigger formula
INCA ConfigureRecorderEndCond Duration duration_s
INCA ConfigureRecorderEndCond Manual
```

Description

Before starting a capturing process, the recorder start condition must be specified with this command. It takes one of the following three forms:

Specifying *Trigger* as first argument will create a trigger which stops the recorder when a certain criteria is met. The criteria is given by *formula* which will be tested for its validity by the time of this call.

Calling the command with the argument *Duration* will result in a recording operation of exactly *duration_s* seconds from the start of recording on.

When choosing *Manual* as argument, the recorder will record as long as no call to INCA *StopRecording* is made

Examples

```
INCA ConfigureRecorderEndCond Trigger 2 "TVLR <= 100"
INCA ConfigureRecorderEndCond Duration 10
INCA ConfigureRecorderEndCond Manual
```

INCA ConfigureRecorder - Apply recorder settings

Syntax

```
INCA ConfigureRecorder
```

Description

After specifying the recorder start and end condition via *INCA ConfigureRecorderStartCond* and *INCA ConfigureRecorderEndCond* a call to *INCA ConfigureRecorder* is essential for applying the previously made changes to the actual recorder configuration. The settings will be applied to all subsequent capture processes as long as they are not modified again.

Examples

```
INCA ConfigureRecorder
```

INCA StartMeasurement - Start measurement process

Syntax

```
INCA StartMeasurement ?-noRecording?
```

Description

Starts a measurement for the currently opened experiment. If *-noRecording* is specified the automatic recording will be suppressed as long as no call to *INCA StartRecording* is made.

Examples

```
INCA StartMeasurement -noRecording  
INCA StartMeasurement
```

INCA StopMeasurement - Stop measurement process

Syntax

```
INCA StopMeasurement
```

Description

Stops a running measurement immediately regardless of a previously specified recorder end condition.

Examples

```
INCA StopMeasurement
```

INCA StartRecording - Start recording operation

Syntax

```
INCA StartRecording
```

Description

Starts the recording process manually. A measurement must already be active for this call to succeed.

Examples

```
INCA StartRecording
```

INCA StopRecording - Stop recording operation

Syntax

```
INCA StopRecording
```

Description

Stops the recording process manually. This command can be called whenever a recording process is active with no respect to previous changes of the recorder settings.

Examples

```
INCA StopRecording
```

INCA GetApplicationHandle - Return application handle

Syntax

INCA GetApplicationHandle

Description

Returns a handle to the INCA application object. The returned handle may be used to call additional COM functions of the respective interface which are not already supported by ScriptControl's INCA command. For a full description of the INCA application interface consult the official INCA documentation.

Examples

```
set apphandle [INCA GetApplicationHandle]
```

INCA GetDataBaseHandle - Return database handle

Syntax

INCA GetDataBaseHandle

Description

Returns a handle to the INCA database object. The returned handle may be used to call additional COM functions of the respective interface which are not already supported by ScriptControl's INCA command. For a full description of the INCA database interface consult the official INCA documentation.

Examples

```
set dbhandle [INCA GetDataBaseHandle]
```

INCA GetWorkspaceHandle - Return workspace handle

Syntax

INCA GetWorkspaceHandle

Description

Returns a handle to the INCA workspace object. The returned handle may be used to call additional COM functions of the respective interface which are not already supported by ScriptControl's INCA command. For a full description of the INCA workspace interface consult the official INCA documentation.

Examples

```
set wshandle [INCA GetWorkspaceHandle]
```

INCA GetExperimentHandle - Return experiment handle

Syntax

INCA GetExperimentHandle

Description

Returns a handle to the INCA experiment object. The returned handle may be used to call additional COM functions of the respective interface which are not already supported by ScriptControl's INCA command. For a full description of the INCA experiment interface consult the official INCA documentation.

Examples

```
set ehandle [INCA GetExperimentHandle]
```

INCA GetExperimentEnvHandle - Return exp. environment handle

Syntax

INCA GetExperimentEnvHandle

Description

Returns a handle to the INCA experiment environment object. The returned handle may be used to call additional COM functions of the respective interface which are not already supported by ScriptControl's INCA command. For a full description of the INCA experiment environment interface consult the official INCA documentation.

Examples

```
set eehandle [INCA GetExperimentEnvHandle]
```

INCA GetExperimentViewHandle - Return experiment view handle

Syntax

INCA GetExperimentViewHandle

Description

Returns a handle to the INCA experiment view object. The returned handle may be used to call additional COM functions of the respective interface which are not already supported by ScriptControl's INCA command. For a full description of the INCA experiment view interface consult the official INCA documentation.

Examples

```
set evhandle [INCA GetExperimentViewHandle]
```

INCA GetExperimentDevHandle - Return exp. device handle

Syntax

INCA GetExperimentDevHandle

Description

Returns a handle to the INCA experiment device object. The returned handle may be used to call additional COM functions of the respective interface which are not already supported by ScriptControl's INCA command. For a full description of the INCA experiment device interface consult the official INCA documentation.

Examples

```
set edhandle [INCA GetExperimentDevHandle]
```

17.4.23 tcom command family

Among Windows applications Microsoft's COM (component object model) is widely spread and often used for interaction between different tools including the automation of an application by another one. ScriptControl provides the possibility to access other applications via their COM interface by using the tcl extension tcom.

Here is an overview of the most used tcom commands. For further information about tcom and its possibilities please take a look at the official tcom online-documentation. Note that all tcom commands are only available on Windows systems.

::tcom::ref createobject - Create an instance of a COM object

Syntax

```
::tcom::ref createobject ProgID
::tcom::ref createobject -clsid CLSID
```

Description

When trying to access an application via COM the first step the user has to accomplish is to retrieve the class identifier (CLSID) of the respective application interface. A CLSID is a globally unique identifier and is stored in the Windows registry for every installed application which provides COM server functionality on the machine. Since a CLSID is hardly human-readable a string representation of it is also provided, the so called programmatic identifier (ProgID).

This command returns a handle to the application specified by *ProgID* or *CLSID*. The returned handle should generally be stored in a tcl variable for later use. A handle represents the underlying object and is used to invoke the object's methods and to access its properties. For every handle a Tcl command with the same name as the handle is created (see below).

Examples

```
set handle [::tcom::ref createobject "Excel.Application"]
set handle [::tcom::ref createobject -clsid "00024500-0000-0000-C000-00000000046"]
```

::tcom::ref equal - Compare handles for identity

Syntax

```
::tcom::ref equal handle1 handle2
```

Description

This command compares two handles *handle1* and *handle2* for identity. It returns 1 if the two handles refer to the same COM object, or 0 if not.

Examples

```
::tcom::ref equal $handle1 $handle2
```

handle methodName ?arguments? - Invoke a method

Syntax

```
handle methodName ?arguments?
```

Description

A method of an object represented by *handle* is simply invoked by calling the handle, followed by the method's name and its arguments. The return value of such a method can be a string, a number, a list or even a new handle to another object which itself can have its own properties and publish its own methods.

Examples

```
$handle Visible 1  
$handle Quit
```

handle -get propertyName - Get a property's value

Syntax

```
handle -get propertyName
```

Description

The property *propertyName* of an object is accessed by using the keyword -get for retrieving the property's value.

Examples

```
$handle -get IsRunning
```

handle -set propertyName value - Set a property's value

Syntax

```
handle -set propertyName value
```

Description

The property *propertyName* of an object is modified by using the keyword *-set* for setting the property to a new value.

Examples

```
$handle -set IsRunning 1
```

::tcom::import - Import type library information

Syntax

```
::tcom::import TypeLibraryFileName
```

Description

To retrieve feasible information about the methods and properties a COM object exposes to the outside the so called type library files can be used. These binary files are often delivered with the application and include information about available interfaces, methods and properties of the application.

This command converts the type information from a type library file into Tcl commands to access COM classes and interfaces.

Examples

```
::tcom::import SampleTypeLibFile.tlb
```

::tcom::info interface - Return interface description

Syntax

```
::tcom::info interface handle
```

Description

Given a handle to an object one can retrieve the interface information of the object by executing this command. It returns a handle to an interface description which holds the interface's name, its identifier and the available methods and properties. A call to the interface description handle followed by the respective keywords *name*, *iid*, *methods* or *properties* delivers all necessary information about the interface.

Examples

```
set interfacedescr [::tcom::info interface $handle]
$interfacedescr name
$interfacedescr iid
$interfacedescr methods
$interfacedescr properties
```

Further information about tcom may be obtained from:

- [1] The developers webpage: <http://www.vex.net/~cthuang/tcom/>
- [2] The Tcl'ers wiki tcom webpage: <http://wiki.tcl.tk/1821>
- [3] How one discovers the API of a COM exporting application: <http://wiki.tcl.tk/4472>

17.4.24 Miscellaneous Commands

RunScript – Run a ScriptControl script

Syntax

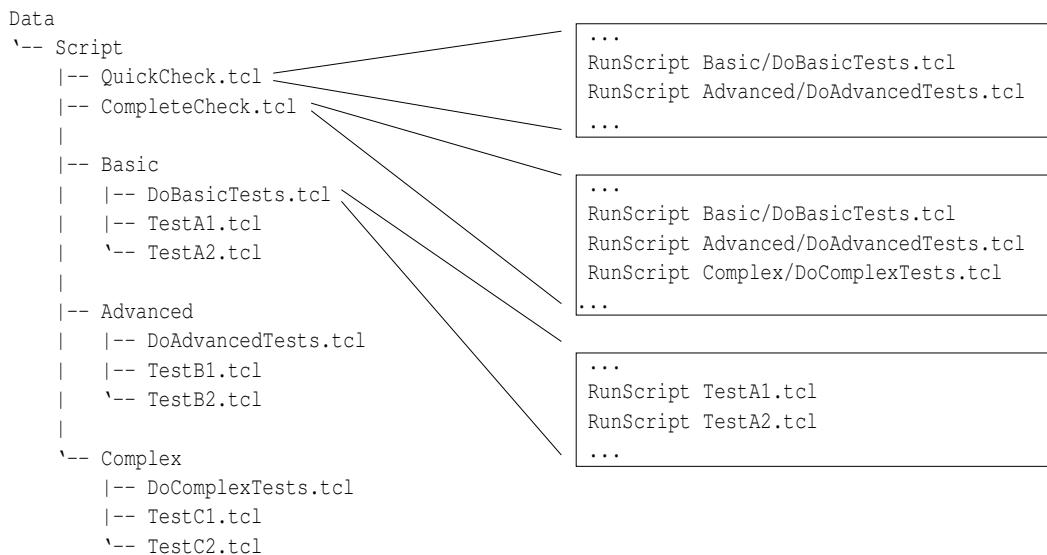
RunScript *path*

Description

Runs the ScriptControl script specified by *path*.

Relative path names are interpreted in a special way, so as to make running scripts from within other scripts simpler. Specifying a relative path to a script will cause the script to be searched relative to the directory where the calling script resides. If no script with this name can be found at this place, *RunScript* tries makes a second attempt searching the script relative to the current working directory.

Simply stated, this treatment of relative paths makes it easy for you to organize your tests hierarchically, as shown in the following figure:



Note how the subordinate scripts *DoBasicTests.tcl*, *DoAdvancedTests.tcl*, etc. can be written without depending on the location of the calling script. If Tcl's *source* command would have been used instead, more scripts were needed and, because one is forced to specify the complete path to the individual scripts, they would depend on each other. As a consequence, few changes in script organization would result in changing lots of paths in lots of files.

It should be noted, however, when invoking *RunScript* interactively from the ScriptControl console window, relative pathnames are interpreted relative to the *Data/Script* directory.

RunScript is intended as a handy alternative to Tcl's *source* command.

Examples

```
RunScript /space/TestCollection/TypeB/SpecificTypeBTest.tcl  
RunScript Stability/TypeA/RunTypeATests.tcl  
RunScript SpecificTypeATest_1.tcl
```

CheckBreak – Check for user interrupt

Syntax

CheckBreak

Description

Checks whether the user pressed the ScriptControl Stop button in order to interrupt script execution.

Normally, a press on the Stop button can only be detected while the Tcl interpreter is able to process input/output requests or when a ScriptControl command is executed. However, when Tcl is busy with execution of e.g a loop to do a longer running, complicated postprocessing calculation of your test script, i.e. a loop in which no ScriptControl commands are used, the press of the Stop button will go unnoticed. You will not be able to interrupt your script, the GUI will appear to be blocked.

A CheckBreak command executed regularly inside the loop solves this problem.

AbortScript – Stop script execution

Syntax

AbortScript

Description

Aborts script execution. No further commands of the script will be executed.

GetTime – Obtain the current date and/or time

Syntax

GetTime ?*format*?

Description

Returns the current date and/or time. *GetTime* serves as a shortcut to Tcl's more elaborate [clock] function. The *format* option determines value and format to be returned.

Valid *format* options are:

- d current date as "yyyy-mm-dd"
- D current date in country specific format ("mm/dd/yyyy" for example)
- t current time as "hh:mm:ss"(24-hour format)
- T current time in country specific format
(12-hour format with AM/PM appended or 24-hour format)
- a current date and time like "yyyy-mm-dd hh:mm:ss"
- A current date and time like "Thu Jun 23 12:15:01 PM CEST 2005".

If no format option is specified, or the format specified is not valid, then -a is assumed.

Example

```
set now [GetTime -T]
```

Sleep – Delay execution for a certain amount of time

Syntax

Sleep *ms*

Description

Causes the script to sleep/wait for *ms* milliseconds.

This command should be used in favour of Tcl's built-in [*after ms*] since it does not keep the Tcl interpreter from doing input/output during the waiting time.

Example

```
Sleep 2000
```

SimInfo – Inquire about the most recently started simulation

Syntax

SimInfo *what*

Description

The *SimInfo* command delivers information about the most recently started simulation (which may either have finished or still be running). Parameter *what* specifies the desired information:

<i>what</i>	Description
testrun	The path to the currently loaded TestRun relative to the <i>Data/TestRun</i> directory.
resultfile	Identical to the <i>GetLastResultFName</i> command.
simstatus	Identical to the <i>SimStatus</i> command.
dist	The distance driven so far during the simulation.
time	The current simulation time.
enddist	The driven distance as transmitted at the end of the TestRun.
endtime	The simulated time as transmitted at the end of the TestRun.
endstatus	While a simulation is running, the command returns an empty string. In all other cases one of the following string values is returned: completed – The simulation ran until the end. aborted – The simulation was stopped prematurely. failed – The simulation ended with an error. unknown – No simulation was run yet.

Example

```
set fname [SimInfo resultfile]
```

AddMenuEntry – Add a menu entry to the main menubar

Syntax

`AddMenuEntry type options`

Description

The `AddMenuEntry` acts like Tcl's `menu add` command. Menu entries which are added via this command are appended to the menu item `Extras` as part of the main menubar. Valid types are `cascade`, `checkbutton`, `command`, `radiobutton` and `separator`. Options are type-specific and resemble the options supported by the respective Tcl menu items.

Example

```
AddMenuEntry command -label "My Menu Entry" -command {::MyCommand}  
AddMenuEntry cascade -label "My Submenu" -menu .mymenu
```

Chapter 18

Remote GUI Control

The CarMaker GUI offers several ways of letting external applications exert some kind of remote control over it. Over each of these interfaces arbitrary ScriptControl and Tcl commands may be sent to the CarMaker GUI for execution.

18.1 TCP Command

The TCP command server provides remote command execution through a TCP network socket. Applications wanting to issue commands over this network connection may be running on the local host as well as on an arbitrary host on the local network.

18.1.1 Starting the TCP command server

The TCP command server is started only if provided the port number of an unused TCP port at startup. This number <p> must be in the range 1024 - 65535 and can be set in one of the following ways:

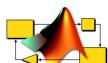
- Start the CarMaker GUI with the command line option -cmdport <p>.
- In the *.CarMaker.tcl* file, set the port number with a statement like this:

```
set Pgm(TcpCmdPort) <p>
```

Note: *Pgm* is a global Tcl variable.

- When starting the CarMaker GUI from within Matlab, the port number must be assigned to a Matlab workspace variable:

```
CMDData.CmdPort = <p>;
```



18.1.2 Protocol specification

The protocol is purely textual.

Command message

A command message that is sent to the TCP command server consists of three items, with the first two being optional and only the third item being mandatory:

- Item 1: The number of lines that follow. After reading that many lines from the socket, the command is executed.
This item is optional. If not present, the command will be executed as soon as the command is considered to be complete. This is called “telnet mode”; see below.
- Item 2: An exclamation mark ‘!’ which signals, that on the sender side there is no interest in information about execution status or result value of the command, and that no result message should be sent.
This item is optional. If not present, a result message will be sent back after execution of the Tcl command. See below for the specification of the result message.
- Item 3: The Tcl command, which may consist of one or more lines, each line terminated by a line feed character (LF, \n).

Result message

The return value of the executed Tcl command may consist of zero or more lines of output. In the result message these lines are sent one after the other, each line introduced by the status code and terminated by a line feed character (LF, \n).

The status code tells about success or failure of the execution of the whole Tcl command. The protocol currently defines two different status codes:

- ‘O’ (the letter, as in “Ok”): Execution succeeded
- ‘E’ (as in “Error”): Execution failed

The status code is only determined once, so each result line of the message is introduced by the same status code.

Always after the last result line an empty line consisting only of a line feed character (no status code!) is added, terminating the result message.

Examples

Note: \n denotes a line feed character.

Message: expr {21+21}\n
Answer: 042\n\n (no error, return value 42)

Message: !expr {21+21}\n
Answer: (none because of the exclamation mark)

Message: Log Hello\n
Answer: \n (no error, return value is an empty string)

Message: stupid\n
Answer: Einvaliid command name "stupid"\n\n (Tcl error)

Message: 4while {1} {\nLog hello\nbreak\n}\n
Answer: \n (no error, return value is an empty string)

Message: 3while {1} {\nLog hello\nbreak\n
Answer: Emissing close-brace\n\n (Tcl error)

Message: 3!while {1} {\nLog hello\nbreak\n
Answer: (none because of the exclamation mark)

18.1.3 Testing the TCP command server with telnet

As the TCP command protocol is “pure ASCII”, the telnet program seems to be a natural choice for experimenting with the command server.

Let us assume the TCP command server is listening on TCP port 16660.

Linux



In a terminal window, run ‘telnet localhost 16660’.

MS Windows



In a DOS command window, run ‘telnet’.

Once inside telnet, type ‘set localecho’ (under MS Windows 2000: ‘set local_echo’), otherwise you will not see what you type.

Then connect to the CarMaker GUI by typing ‘open localhost 16660’.

After the telnet session is established, enter your Tcl commands according to the protocol specification and the examples described above.



Hint: If the focus is less on the command protocol and more on experimenting with the available ScriptControl and Tcl commands, better change to the ScriptControl console, which provides much more line editing comfort.

18.2 Tcl send (Linux only)

On Linux systems the GUI is open for commands sent from other Tcl/Tk applications using Tcl’s built-in *send* command. The “tk appname” of the GUI is “CarMaker”.

18.3 DDE (MS Windows only)

On MS Windows systems the CarMaker GUI can execute Tcl commands sent to it via DDE. The service name of the CarMaker GUI is “TclEval”, the topic is “CarMaker”.

To start a communication over a DDE channel with CarMaker, the DDE package for Tcl/Tk applications needs to be loaded and a DDE communication server needs to be set up. For this, execute the following commands in the ScriptControl console (or alternatively the commands can be written to the .CarMaker.tcl file):

```
package require dde  
dde servername SC
```

Find below an example script for a DDE communication with CarMaker. Do not forget to close the DDE channel after the communication is done!

```
Example    Tcl_DDE_Service = 'TclEval';
            Tcl_DDE_Topic = 'SC';

            disp('Initialize communication channel')
            DDE_CH = ddeinit (Tcl_DDE_Service,Tcl_DDE_Topic);

            disp('Get data from ScriptControl')
            SC_SimIsIdle = ddereq(DDE_CH,'HIL(SimIsIdle)');

            disp('Start customized simulation')
            if (SC_SimIsIdle)
                er = ddeexec(DDE_CH,'LoadTestRun Examples/CarMakerFunctions/IPGRoad/
                                Hockenheim');
                er = ddeexec(DDE_CH,'StartSim');
            end

            disp('Close communication channel')
            rc = ddeterm(DDE_CH);
```

18.4 CarMaker for Simulink

The CarMaker for Simulink library offers a special command called *cmguicmd* which can be used to execute any Tcl statement including all ScriptControl commands. The statement is placed in single quotes:

```
cmguicmd('COMMAND',TIMEOUT);
```

Example cmguicmd('LoadTestRun "Examples/VehicleDynamics/Braking"');

Optionally, a timeout may be specified to determine how long to wait for completion of a command. Possible values for TIMEOUT are:

- -1: Wait until completion of the command.
This is also the default value if no timeout is specified.
- 0: Do not wait, return immediately an empty string.
- > 0: Wait at most the specified number of milliseconds for completion of the command. If the command does not finish in time, return 'timeout' as result.



Always define a timeout of 0 ms in combination with the *StartSim* command to prevent the single task program Matlab from seizing: *cmguicmd('StartSim', 0)*.

Chapter 19

FailSafeTester

It is necessary to add some function calls to the CarMaker's source code to allow the FailSafeTester to be initialized, and also to establish the communication between the FailSafeTester and the real time computer. The communication must use the CAN bus.

19.1 FST Global Variables

The following table shows the global (integer) variables used by the CarMaker.

The content of these variables is read from the ECUParameters file at startup and should not be changed manually.



FST_CAN_Slot	The Slot of the M51 module, which is connected to the FST(s).
FST_CAN_Ch	The CAN channel connected to the FST(s)
FST_CAN_Id	The CAN id of the first FailSafeTester. FST_CAN_Id and FST_CAN_Id+1 are used for communication. If more than one FST is used the second FST has the ID FST_CAN_Id+2, the third one uses FST_CAN_Id+4 and so on.

19.2 Configuring the Communication via CAN

Simply call `FST_ConfigureCAN()`.

19.3 FailSafeTester C-Functions

Inside the User.c and IO.c source file, include the following functions in the appropriate place. See the `Car_Generic/src` directory for example function placement.

FST_Init()

```
void FST_Init (tInfos *inf);
```

Description

This function initializes the FailSafeTester variables. It must be called once if the application is started. When using the Car_Generic example or creating a new project by File=>Project=>Create, this function is implemented in the function App_Init () within CM_Main.c.

FST_ConfigureCAN()

```
void FST_Init (tInfos *inf);
```

Description

This function configures the CAN-Module (M51 or M410), enables the configured CAN-IDs and sets the communication parameters for the communication with all the configured Fail-SafeTesters. The configuration is taken from the current ECUParameters file.

FST_Cleanup()

```
void FST_Cleanup (void);
```

Description

This function must be called once before exiting the application. When using the Car_Generic example or creating a new project by File=>Project=>Create, this function is implemented in the function App_Cleanup () within CM_Main.c.

FST_New()

```
int FST_New (struct tInfos *Inf);
```

Description

This function reads the configuration from the file ECUParameters. It must be called once at the start of each TestRun. Additionally it checks, if the FST's configuration matches the configuration data and decides if the user is allowed to use the FST.
This function is called from within User_TestRun_Start_atBegin() in User.c.

Return Value

0 - No error. Either everything is ok and the FST can be used or the FailSafeTester is disabled by the user by setting FST_UseIt=0.

-1 - Any error happened. See the Session Log for more details.

FST_IsActive()

```
int FST_IsActive (void);
```

Description

This function returns if all connected FSTs should and can be used.

To use the FST(s) the following conditions must be fulfilled:

- The FST.CANSlot must be configured and must be greater than -1
- The FST.CANChannel must be configured and must be greater than -1
- The FST.CANID must be configured correctly

Return Value

0 - FailSafeTester is disabled.

1 - FailSafeTester is enabled and can be used.

FST_MsgIn ()

```
int FST_MsgIn (unsigned CycleNo, struct CAN_Msg *Msg);
```

Description

FST_MsgIn reads and analyzes the data which is sent by the FST. It must be called each TestRun Cycle, e.g. within IO_In () .

Return Value

0 - No error.

<0 - Any error happened. See the Session Log for more details.

FST_MsgOut ()

```
int FST_MsgOut (unsigned CycleNo);
```

Description

FST_MsgOut must be called each main cycle, e.g. IO_Out () . This function sends the commands to the FST.

Return Value

0 - No error.

<0 - Any error happened. See the Session Log for more details.

FST_ApoMsgEval (ch, msg, len)

int FST_ApoMsgEval (int Ch, char *MsgBuf, int len)

Description

This function evaluates FST control messages from the FST-GUI. It usually is called within User_ApoMsgEval()

Return Value

0 - ok, message handled, everything is ok

>0 Warning

1 - Unknown FST message, message not handled.

<0 - Error, Msg for FST

-2 - message ignored, because of FailSafeTester not accessible

Appendix A

Replacing the Vehicle Model

A CarMaker user might want to stay with the CarMaker environment, but simulate with his or her own vehicle model. Thanks to the modular architecture of CarMaker the integration of a so called *user vehicle* can be done with relatively little effort.

The model can be integrated using either CarMaker's C-code interface, CarMaker's model plug-in mechanism for Real-Time Workshop, or in CarMaker for Simulink. Multiple vehicle models can coexist inside a single CarMaker simulation executable. Switching to different vehicle is actually a matter of switching to a different vehicle data set, so e.g. within a single test series vehicle models can be selected as needed.

It is important to understand, that internally CarMaker differentiates three different vehicle "sources". Which one is used during a testrun can be determined by inspecting global C-code variable *Vehicle.Model.Source*, its possible values are given in the list below:

- *VehicleSource_Builtin*
The built-in default vehicle IPGCar, always available.
- *VehicleSource_ModelMgr*
A vehicle model implemented in C-code and registered in the Model Manager. This may also be a model implemented in Simulink plug-in model and built with Real-Time Workshop.
- *VehicleSource_CM4SL*
A CarMaker for Simulink user vehicle, implemented and run in Simulink.

The corresponding signal in CarMaker for Simulink is called *Vhcl.Model.Source*.

A.1 User Vehicle as a C-Code extension

When using the C-code interface, the user vehicle is registered just like any other model for a particular model class in CarMaker's Model Manager.

A.1.1 Tires

In case of a vehicle model implemented as a C-code extension, the vehicle model can either implement its own tire model or use one of the tire models that the Model Manager provides.

If the vehicle model provides its own tire model, the tire data set configured in the testrun or vehicle data set is ignored during the simulation. A tire data set may be configured, so that the tire geometry for IPGMovie is taken from it, but it is also valid to have all tires deselected.

If one of the Model Manager's tire models is to be used, the vehicle model must provide all relevant tire interface inputs, call function `VhclModel_Tire_Calc()` and fetch the tire model's outputs afterwards.

A.1.2 Brake

The vehicle model may provide its own brake implementation. In the vehicle data editor, on the *Brake* tab select *Model Manager off* in this case, otherwise the vehicle model's brake output signals will be ignored and the configured brake model will be used instead.

A.1.3 Powertrain

The vehicle model may provide its own powertrain implementation. In the CarMaker GUI's vehicle data editor, on the *Powertrain* tab select *Model Manager off* in this case, otherwise the vehicle model's powertrain output signals will be ignored and the configured powertrain model will be used instead.

A.1.4 The MyCar Example

The *MyCar* example is a part of the *Sources: Extra Models*, that you can optionally install into your project directory. It consists of the following files:

- The vehicle model source code, `src/ExtraModels/MyCar.c`
- A vehicle data set, `Examples/ExtraModels/UserVehicle_MyCar`

The model provides its own tire model and also brake and powertrain.

Simulating with the model

After the simulation program is built and the new executable is selected and started, everything is ready to run a simulation.

In principle, any testrun can be used as long as the right vehicle dataset is selected. To activate the user vehicle, the vehicle dataset `Examples/ExtraModels/UserVehicle_MyCar` needs to be selected. This data set comes with some special settings:

- At the beginning of the Infofile, the example user vehicle model is specified in line 2 by referring to the model name *MyCar* (please find more information on the model activation by Infofiles in [section 5.2.2 'Alternative database concept' on page 70](#)).
- When starting the CarMaker GUI's vehicle data editor under *Parameters > Vehicle*, most of the parametrization tabs of the submodels are deactivated. The model specific parameters can be found under *Misc. > Additional Parameters*.

The simulation can now be started.

A.2 User Vehicle as a Plug-in Model in Simulink

In order to integrate a vehicle model implemented in Simulink into CarMaker, it is best to start with an empty model created with CarMaker's *cmPluginModel* command in Matlab. The model class to be used in this case is *Vehicle_Car*. Both the *UserVehicle_RTW* and *SingleTrack_RTW* example models explained below were created this way. A detailed signal list of the vehicle interface can be found at the end of this chapter.

When generating C-code from the model with Real-Time Workshop, compiling and linking it into CarMaker, the vehicle model is actually integrated using the C-code vehicle model interface, so all explanations from the previous chapter apply.

Please find further information on the model plug-in mechanism in [section 'Simulink Coder Interface' on page 159](#).

A.2.1 Tires

In case of a vehicle model implemented as a Simulink plug-in model using Real-Time Workshop, the vehicle model can either implement its own tire model or use one of the tire models that the Model Manager provides.

If the vehicle model provides its own tire model, the tire data set configured in the testrun or vehicle data set is ignored during the simulation. A tire data set may be configured, so that the tire geometry for IPGMovie is taken from it, but it is also valid to have all tires deselected.

If one of the Model Manager's tire models is to be used, *Tire* or *Tire STI* blocks from the CarMaker blockset must be placed inside the vehicle's Simulink model. If no *Tire* or *Tire STI* block is present for a certain wheel position, the configured tire data set is ignored and no tire calculation with a built-in model takes place for this wheel.

A.2.2 Preprocessing

Simulation preprocessing may be skipped by adding the following parameter as an *Additional Parameter* in the CarMaker GUI's vehicle data editor:

```
Vehicle.SkipPreprocessing = 1
```

A.2.3 The UserVehicle_RTW Example

The *UserVehicle_RTW* example is a part of the *Real-Time Workshop Examples*, that you can optionally install into your project directory. It consists of the following files:

- A Simulink model, *src/UserVehicle_RTW.mdl*
- A parameter file for the Simulink model, *src/UserVehicle_RTW_params.m*
- An S-function, *src/tire_lin.c*
- A pregenerated wrapper module for the RTW-generated model C-code, *src/UserVehicle_RTW_CarMaker_rtw/UserVehicle_RTW_wrap.c* and *.h*
- A vehicle data set, *Examples/UserVehicle_RTW*
- A testrun, *Examples/Simulink/Hockenheim_UserVehicle_RTW*
- A short README file, *src/README-UserVehicle_RTW.txt*

This simple vehicle model provides its own tire model, but no brake or powertrain.

A.2.4 The SingleTrack_RTW Example

The *SingleTrack_RTW* example is a part of the *Real-Time Workshop Examples*, that you can optionally install into your project directory. It consists of the following files:

- A Simulink model, *src/SingleTrack_RTW.mdl*
- A parameter file for the Simulink model, *src/SingleTrack_RTW_params.m*
- A pregenerated wrapper module for the RTW-generated model C-code, *src/SingleTrack_RTW_CarMaker_rtw/SingleTrack_RTW_wrap.c* and *.h*
- A vehicle data set, *Examples/SingleTrack_RTW*
- A testrun, *Examples/Simulink/Hockenheim_SingleTrack_RTW*
- A short README file, *src/README-SingleTrack_RTW.txt*

The core model is identical to the *SingleTrack* example of the *CarMaker for Simulink Extras*, with only some extra wiring to match the slightly different model interface. It provides its own tire model and also brake and powertrain.

A.3 User Vehicle in CarMaker for Simulink



With CarMaker for Simulink, the *src_cm4sl/generic_uservehicle.mdl* should always serve as starting point when integrating an alternative vehicle model, as all required interface signals are directly linked.

The CarMaker for Simulink blockset also offers some special blocks to be used in combination with a user vehicle. Among others the built-in tire models and IPGRoad functionality to determine the current vehicle position on the road can be accessed. Please find further information in [section 6.2.8 'Special purpose blocks' on page 113](#).

Please find further information on the CarMaker for Simulink interface in [section 'CarMaker for Simulink' on page 92](#).

A.3.1 Tires

The vehicle model can either implement its own tire model or use one of the tire models that the Model Manager provides.

If the vehicle model provides its own tire model, the tire data set configured in the testrun or vehicle data set is ignored during the simulation. A tire data set may be configured, so that the tire geometry for IPGMovie is taken from it, but it is also valid to have all tires deselected.

If one of the Model Manager's tire models is to be used for calculation, *Tire* blocks (but currently not *Tire STI* blocks) from the CarMaker blockset must be placed inside the vehicle's Simulink model.

A.3.2 The SingleTrack Example

The *SingleTrack* example is a part of the *CarMaker for Simulink Extras*, that you can optionally install into your project directory. It consists of the following files:

- A Simulink model, *src_cm4sl/SingleTrack.mdl*
- A parameter file for the Simulink model, *src_cm4sl/SingleTrack_params.m*
- A vehicle data set, *Examples/Demo_CM4SL_SingleTrack*
- A testrun, *Examples/Simulink/Hockenheim_SingleTrack*

The model provides its own tire model and also brake and powertrain.

Simulating with the model

It is not necessary to build a new simulation program. To activate the user vehicle you only have to select the right vehicle data set. This has to be an Infofile with the model class *CM4SL*. As an example please open the data set *Examples/Demo_CM4SL_SingleTrack*. Special items in this vehicle data set parametrization are:

- In line 2 of the Infofile, the example CarMaker for Simulink user vehicle model is called by the identifier *CM4SL* (please find more information on the model activation by Infofiles in [section 5.2.2 'Alternative database concept' on page 70](#)).
- When starting the CarMaker GUI's vehicle data editor under *Parameters > Vehicle*, most of the parametrization tabs of the submodels are deactivated. The model specific parameters can be found under *Misc. > Additional Parameters*.

The simulation can now be started.

A.4 Vehicle Interface Signals

To successfully replace the whole vehicle model in CarMaker, the user model has to provide several signals required by further CarMaker modules as IPGRoad, IPGDriver etc. The interface signals are listed below. Please find more detailed information about the signals (unit, reference frame, etc.) in the Reference Manual, section "User Accessible Quantities".

Table 19.1: Interface signals for replacement of the vehicle model

Signal in C-Code	Signal in Simulink	Description	Mandatory
Vehicle.Steering.Ang	Steering.Ang	Steering wheel angle	Yes, for steering by angle
Vehicle.Steering.AngVel	Steering.AngVel	Steering wheel angle velocity	Yes, for steering by angle
Vehicle.Steering.AngAcc	Steering.AngAcc	Steering wheel angle acceleration	Yes, for steering by angle
Vehicle.Steering.Trq	Steering.Trq	Steering wheel torque	Yes, for steering by torque
Vehicle.v	Motion.v	Vehicle velocity in global frame	Yes
Vehicle.Distance	Motion.Distance	Road distance covered by vehicle	Yes
Vehicle.sRoad	Motion.sRoad	Vehicle position in road coordinate s (along centerline)	Yes
Vehicle.Side	Motion.Side	Lateral distance to road centerline	Yes

Table 19.1: Interface signals for replacement of the vehicle model

Signal in C-Code	Signal in Simulink	Description	Mandatory
Vehicle.Hitch_Pos[0,1,2]	Motion.Hitch_Pos[0,1,2]	Hitch position in global frame (Fr0)	No, only for trailer
Vehicle.Roll	Motion.Roll	Vehicle roll angle	Yes
Vehicle.RollVel	Motion.RollVel	Vehicle roll velocity	Yes
Vehicle.RollAcc	Motion.RollAcc	Vehicle roll acceleration	Yes
Vehicle.Pitch	Motion.Pitch	Vehicle pitch angle	Yes
Vehicle.PitchVel	Motion.PitchVel	Vehicle pitch velocity	Yes
Vehicle.PitchAcc	Motion.PitchAcc	Vehicle pitch acceleration	Yes
Vehicle.Yaw	Motion.Yaw	Vehicle yaw angle	Yes
Vehicle.YawRate	Motion.YawRate	Vehicle yaw velocity	Yes
Vehicle.YawAcc	Motion.YawAcc	Vehicle yaw acceleration	Yes
Vehicle.Pol_Pos[0,1,2]	Pol.Pos[0,1,2]	Position of Point of Interest in global frame (Fr0)	Yes
Vehicle.Pol_Vel[0,1,2]	Pol.Vel[0,1,2]	Velocity of Point of Interest in global frame (Fr0)	Yes
Vehicle.Pol_Acc[0,1,2]	Pol.Acc[0,1,2]	Acceleration of Point of Interest in global frame (Fr0)	Yes
Vehicle.Pol_Vel_1[0,1,2]	Pol.Vel_1[0,1,2]	Velocity of Point of Interest in vehicle frame (Fr1)	Yes
Vehicle.Pol_Acc_1[0,1,2]	Pol.Acc_1[0,1,2]	Acceleration of Point of Interest in vehicle frame (Fr1)	Yes
Vehicle.Pol_GCS.Long	Pol.GCS.Long	Global position for Pol, expressed in GCS coordinates	Optional
Vehicle.Pol_GCS.Lat	Pol.GCS.Lat		Optional
Vehicle.Pol_GCS.Elev	Pol.GCS.Elev		Optional
Vehicle.<pos>.t[*]	Wheel_<pos>.t	Translation of wheel carrier <pos>	Yes
Vehicle.<pos>.r_zxy[0]	Wheel_<pos>.r_zxy[0]	Rotation angles of wheel carrier <pos>	Yes
Vehicle.<pos>.F[x,y,z]	Wheel_<pos>.F[x,y,z]	Longitudinal, lateral and vertical ground reaction force at wheel/road contact point <pos>	Yes
Vehicle.<pos>.vBelt	Wheel_<pos>.vBelt	Wheel <pos> velocity	Yes
Vehicle.<pos>.LongSlip	Wheel_<pos>.LongSlip	Longitudinal slip at wheel <pos>	Yes
Vehicle.<pos>.SideSlip	Wheel_<pos>.SideSlip	Sideslip angle at wheel <pos>	Yes
Vehicle.<pos>.Trq_T2W	Wheel_<pos>.Trq_T2W	Tire torque around wheel spin axle of wheel <pos>	Yes
Vehicle.<pos>.Trq_WhlBearing	Wheel_<pos>.WhlBearing	Wheel bearing friction torque around wheel spin axle of wheel <pos>	Optional
Vehicle.Fr1A.t_0[0,1,2]	Fr1A.t_0	Position of vehicle frame origin (Fr1A) in global frame (Fr0) for sensors and animation	Yes

Table 19.1: Interface signals for replacement of the vehicle model

Signal in C-Code	Signal in Simulink	Description	Mandatory
Vehicle.Fr1A.v_0[0,1,2]	Fr1A.v_0	Velocity of vehicle frame origin (Fr1) in global frame (Fr0) for sensors and animation	Yes
Vehicle.Fr1A.a_0[0,1,2]	Fr1A.a_0	Acceleration of vehicle frame origin (Fr1) in global frame (Fr0) for sensors and animation	Yes
Vehicle.Fr1.omega_0[0,1,2]	Fr1A.omega_0	Rotational velocity of vehicle frame origin (Fr1) in global frame (Fr0) for sensors and animation	Yes
Vehicle.Fr1.alpha_0[0,1,2]	Fr1A.alpha_0	Rotational acceleration of vehicle frame origin (Fr1) in global frame (Fr0) for sensors and animation	Yes
Vehicle.Fr1.Tr2Fr0[0,1,2][0,1,2]	Fr1A.Tr2Fr0	Transformation matrix from Fr1A to Fr0	Yes
Brake.IF.Trq_WB[0,1,2,3]	Brake.Wheel_<pos>.Trq_WB	Brake torque at wheel <pos>	Yes
Brake.IF.Trq_PB[0,1,2,3]	Brake.Wheel_<pos>.Trq_PB	Park brake torque at wheel	Yes
PowerTrain.IF.Trq_Supp2Bdy1[0,1,2]	PT.Trq_Supp2Bdy1	Support torque to vehicle frame Fr1/Fr1A	Optional
PowerTrain.IF.Trq_Supp2BdyEng[0,1]	PT.Trq_Supp2BdyEng	Support torque to engine frame FrEng	Optional
PowerTrain.IF.Ignition	PT.Ignition	Flag: Ignition On	Yes
PowerTrain.IF.Operation-State	PT.OperationState	Operation state for the vehicle operator	Yes
PowerTrain.IF.OperationError	PT.OperationError	Actual vehicle operation state error from powertrain	No
PowerTrain.IF.Engine_rotv	PT.Engine_rotv	Rotational speed of engine output shaft	Yes
PowerTrain.IF.GearNo	PT.GearBox_GearNo	Current gear	Yes
PowerTrain.IF.DL_iDiff_mean	PT.DL_iDiff_mean	Mean differential ratio from driveline	Yes, for manual gearbox
PowerTrain.IF.WheelOut <pos>.rot	PT.Wheel.<pos>.rot	Rotation angle of wheel <pos>	Yes
PowerTrain.IF.WheelOut [<pos>].rotv	PT.Wheel.<pos>.rotv	Rotation speed of wheel <pos>	Yes
PowerTrain.IF.WheelOut [<pos>].Trq_B2W	PT.Wheel.<pos>.Trq_B2W	Torque acting from brake to wheel <pos>	Yes
PowerTrain.IF.WheelOut [<pos>].Trq_Supp2WC	PT.Wheel.<pos>.Trq_Supp2WC	Drive torque of wheel <pos>	Yes
PowerTrain.IF.WheelOut [<pos>].Trq_BrakeReg	PT.Wheel.<pos>.Trq_BrakeReg	Regenerative braking torque at wheel <pos>	No

Table 19.1: Interface signals for replacement of the vehicle model

Signal in C-Code	Signal in Simulink	Description	Mandatory
PowerTrain.IF.WheelOut [<pos>].Trq_BrakeReg_max	PT.Wheel.<pos>.Trq_BrakeReg_max	Max. regenerative braking torque at wheel <pos>	No

Appendix B5.1.3

Generic Plug-in Models

B.1 Introduction

The main idea of the Generic model class is to give the user a convenient way to include his own models which do not match any of the given model classes. This generic model class can be used for C-Code models as well as FMUs and Simulink Coder Plugin models (formerly Realtime Workshop). There can be various Generic models used in one simulation.

In the same manner as most sub modules (Brake, PowerTrain, etc), a generic model can be registered using the Model Manager mechanism. The calculation of the generic model, however, orientates at the DVA interface points of the main simulation cycle (see [section 1.6 'The main cycle explained'](#)).

The hook point determines when the model will be called in the main cycle. It can be specified relatively to any DVA access point. The call order of the models can be specified by parameters in the Infofile. If the same hook up point is defined for two models, the models will be called in alphabetical order.

B.2 Handling of a Generic Plugin model

As the generic model class can be used as integration platform for any auxiliary, it is treated as an additional module which has no pre-defined place in the CarMaker GUI. It needs be activated in one of the following Infofiles: SimParameters, TestRun or Vehicle/Trailer. For the latter, the keys can be specified under *Additional Parameters*. The required parameter to activate the model is the same one, which specifies the DVA access point:

```
GenericPlugin.<NameOfFMU>.AbsPlace = <AbsPlace>
```

Please find further information later in this chapter ([section B.3 'General Parameters'](#)).

B.3 General Parameters

Following parameters can be specified in the SimParameter, the TestRun or in the vehicle Infofile (including trailer files).



All related parameters, including FMU parameters, of a Generic Plugin have to be included in the same infofile. A model with the same name cannot be specified nor parameterized in more than one file.

GenericPlugin.<modelName>.AbsPlace = DVAStr

The parameter *AbsPlace* specifies the DVA access point, at which the model will be called. Possible values are:

- *IO_In:* DVA IO_In
- *DM:* DVA DrivMan
- *VC:* DVA VehicleControl
- *PTC:* DVA PowerTrainControl
- *IO_Out:* DVA IO_Out

GenericPlugin.<modelName>.LocalPos = value

Optional. The parameter *LocalPos* specifies the call order of the models at the DVA access point given by *AbsPlace*. Any model with a negative value will be called directly before the access point. Models with a positive value or zero will be called directly after the access point. If there are calls for multiple models at the same DVA access point, the models call order will be determined by value, lowest to highest. If multiple models use the same DVA access point and have the same value for *LocalPos*, the call order will be determinate by the alphabetical order of their model names.

Default: 0.

GenericPlugin.<modelName>.Reload = ModeStr

Optional. Specifies if the model should be loaded and terminated after every simulation. The default behavior is that all models will be terminated and reloaded with the start of the next simulation run. This functionality is not supported with CarMaker for Simulink. Following modes are possible:

- *Always:* The model is terminated and reloaded before each simulation start (default).
- *Auto:* The model is terminated and reloaded, in case the Infofile which is used to specify the model was modified.
- *Skip:* The model will not be terminated or reloaded as long as it is part of the simulation.



The automatic reloading mode *Auto* does not consider changes by *Named* or *Key Values* (TestManager) in the Infofile.

Index

.CANiogen.tcl 607

A

Ambient 19
animation 20
Apo 20
Apo communication 39
Apo library 16
Apo message 40
Architectural Overview 521
ASAP2 515
ASAP2 file 518

B

Blockset
 Block properties 98
 CarMaker Model Configuration ..
 100
 CarMaker utility blocks 99
 DrivMan 123, 124
 External Suspension Forces. 133
 Modeling principles 99
 Open GUI 99
 Powertrain 127
 Steering 129
 Sync_In 99
 Sync_Out 99
 Tire 131
BodyCtrl 139
BodyCtrl.mdl 139
Brake.IF.In 126
Brake.IF.Out 126, 127

C

CAL 517
Calculations 39
Calibration 514
Calibration commands 517
CAN 515, 517, 522
CAN Calibration Protocol 515
CANiogen 594
Car.Aero 130
Car.Fr1 135
Car.Hitch.FrcTrq 136, 137
Car.Load 130
Car.Trq_T2W 135
Car.Virtual 129
CarMaker/HIL 15
CarMaker GUI 14, 15
 CM_Simulink 95
 Start button 95
 Starting the CarMaker GUI ... 95
 Stop button 95
CarMaker library 18
CarMaker Model Configuration
 CarMaker Model Configuration
 Block 100
C code modules
 user accessible 18
CCP 514, 515
CCP command 518
CCP protocol 521
CM_Main.c 19, 38
CM_Main.cffd 37
CM_Simulink 95
CM_Vehicle.c 19
cmenv.m 93, 159
Upgrading 96

cminstdir.....	97	events	39
cmread.....	156	Examples	
Example usage	156	BodyCtrl	139
cmstartcond	96	TractCtrl	147
Command line arguments	100	UserBrake	147
communication model.....	515	UserPowerTrain	148
CRO.....	516	UserSteer.....	149
CTO.....	517	UserTire	152
cycle	39		

D

DAQ.....	515, 517
DAQ commands	516, 518
DAQ list	516
DAQ Processor	516
Data acquisition commands	517
data aquisition	515
DataDict.....	20, 79
General information.....	79
data dictionary	20, 79
Data file	
organisation	15
data measurement.....	518
Data subdirectory	15
DemoCar_UserPowerTrain	148, 149, 154
Dictionary	
Initialization.....	104
-disablebrake	148
-disablepowertrain	149
-disablesteer.....	149, 150
DNLOAD	518
DOWNLOAD	519
DrivMan	19, 20
DrivMan.Brake ...	123, 124, 125, 126
DrivMan.PT.....	126
DrivMan.Steering	125
DrivMan module	39
dSPACE	520
DT_CM4SL_UserTire	152
DTO.....	516, 517

E

EC_General	58
EC_Init.....	58
EC_Sim	58
ECU.....	514
Environment.h	62
erg files.....	156
Errors	155
EtherCAT	573
Ethernet.....	515, 517, 522
event driven	38
event loop	39

F

FailSafeTester.....	766
FDamp	133
FlexRay.....	430, 515, 517, 522
FSpring	133
FStabi	134

G

generic.mdl	94
geometry configuration	16
GET_SEED	518, 519
GetInfoErrorCount()	62
Global variables	19
graphical user interface	14
GUI	
Starting the CarMaker GUI ...	95

H

Hardware input/output	39
-----------------------------	----

I

Idle	40
iGetTxtOpt().....	61
infoc.h	62
Infofile	20, 60
Access functions	61
Error handling	61
Example code	62
Infofile format	60
infofile handle	61
key	61
key-value pair	61
value	61
infofile handle	61
infofiles	60
InfoUtils.h	62
initialization	40, 60
Instruments	14
Integration	523
IO.c	19
IPG-CONTROL	14, 15, 100
IPG-MOVIE	14, 15

L

libcar.a	18
libcarmaker	18
libcarmaker.a	20, 60
libcarmaker4sl	26
libraries	
special purpose.....	18
Limitations.....	520
Log.....	20, 58
Errors	57
General information.....	57
Informational messages	58
List of functions.....	58
message categories	57
Recommended use	58
Warnings	58
Log.h.....	58, 59
LogErrF	58
LogErrStr.....	58
log file	15, 20, 57
Logging Module.....	57
LogWarnF	58
LogWarnStr.....	58
ISpring	132
ISabi	133

M

make	26
mandatory.....	518
master.....	515
Matlab	
Search path	93, 159
Starting.....	93, 159
Starting Matlab under Unix ...	93
Starting Matlab under Windows .	94
Workspace variable	156
Measurement.....	514
measurement.....	516, 518
message	16
Model	
Creating a new model.....	94
Running a simulation.....	95
Switching between models ...	95
Model library	
Identification.....	26
Movie subdirectory.....	15

O

Object Descriptor Tables	516
Open GUI	
Open GUI block.....	99
optional commands	518

Overview.....	515
---------------	-----

P

PAG	517
Page switching commands.....	517
parameters.....	518
PGM.....	517
PID	516
post processing	514
post-processing	518
PowerTrain.Misc.....	128
preparations	40
Process Identifier.....	516
Program interaction	16
Programming commands	517
Program setup	39
project directory	15

Q

quantities	16
quantity	
address.....	79
monotone increasing	79
name	79
unit	79

R

real-time conditions	57
real-time performance	80
Road data.....	19

S

sample group	518
scheme	521, 522
Search path.....	93, 159
Server application name.....	100
service tool	515
Shortcut	94
Signals	
Brake.IF.In	126
Brake.IF.Out	126, 127
Car.Aero.....	130
Car.Fr1	135
Car.Hitch.FrcTrq.....	136, 137
Car.Load.....	130
Car.Trq_T2W.....	135
Car.Virtual	129
DrivMan.Brake	123, 124, 125, 126
DrivMan.PT.....	126
DrivMan.Steering	125
FDamp	133

FSpring 133
FStabi 134
ISpring 132
IStabi 133
PowerTrain.Misc 128
Steer.IF 129
TireXX_In 131
TireXX_Out 131
vDamp 132, 133
WheelCarrier.Misc 135
SimEnv 19, 61
SimInput subdirectory 15
SimOutput subdirectory 15
SimStart 40
SimStop 40
Simulate 40
Simulating 95
Simulation
 Results files 156
Simulation Parameters 95
simulation program 14, 15
Simulation results 156
simulation results 15, 20
SL_BodyCtrl 140
SL_HockenheimUserSteerTorque ..
150
SL_HockenheimUserTire 152
SL_TractCtrl 147
slave 515
SoftABS_params.m 154
Solver 98
Solvers 94
Solver step size 155
SOME/IP 489
Standard commands 517
Start button 95
startup.dict 104
Start values 96
states 39
static equilibrium position 40
STD 517
Steer.IF 129
Stop button 95
Storage of results 39
Storage of simulation results 79
struct
 SimEnv 61
 tInfos 61
SxI 517
Sync_In 99
Sync_Out 99

T

temperature 19
TireXX_In 131
TireXX_Out 131

TractCtrl 147
TractCtrl.mdl 147
Troubleshooting 155

U

Universal Measurement and Calibra-
tion Protocol 515
UNLOCK 518, 519
UPLOAD 518, 519
USB 517
User.c 19
UserBrake 147
UserBrake.mdl 147
user interface 14
user interface programs 14
user interface tools 16
UserPowerTrain 148
UserPowerTrain.mdl 148
UserSteer 149
UserSteer.mdl 149
UserSteerTorque 150
UserSteerTorque.mdl 150
UserTire 152
UserTire.mdl 152, 153, 154

V

variable
 Ambient 19
 DrivMan 19
 SimEnv 19
vDamp 132, 133
vehicle module 18
Vhcl_NewInit() 61
virtual driver 20

W

WheelCarrier.Misc 135
wind 19
Windows Explorer 94
Workspace variable 156

X

XCP 514, 515
XCP commands 519
XCP on CAN 517
XCP protocol 522
XCP specification 519