TargetLink style guide

Modular distributed development with TargetLink

*August 2018*

TargetLink Product Management
dSPACE GmbH
Rathenastraße 26
33102 Paderborn
Germany
How to Contact dSPACE

Mail: dSPACE GmbH
    Rathenaustraße 26
    33102 Paderborn
    Germany
Tel.: +49 5251 1638-0
Fax: +49 5251 16198-0
E-mail: info@dspace.de
Web: http://www.dspace.com

How to Contact dSPACE Support

If you encounter a problem when using dSPACE products, contact your local dSPACE representative:

- Local dSPACE companies and distributors: http://www.dspace.com/go/locations
- For countries not listed, contact dSPACE GmbH in Paderborn, Germany. Tel.: +49 5251 1638-941 or e-mail: support@dspace.de

You can also use the support request form: http://www.dspace.com/go/supportrequest. If you are logged on to mydSPACE, you are automatically identified and do not need to add your contact details manually.

If possible, always provide the relevant dSPACE License ID or the serial number of the CmContainer in your support request.

Software Updates and Patches

dSPACE strongly recommends that you download and install the most recent patches for your current dSPACE installation. Visit http://www.dspace.com/go/support for software updates and patches.

Important Notice

This document contains proprietary information that is protected by copyright. All rights are reserved. The document may be printed for personal or internal use provided all the proprietary markings are retained on all printed copies. In all other cases, the document must not be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form, in whole or in part, without the prior written consent of dSPACE GmbH.

© 2013 - 2018 by:
dSPACE GmbH
Rathenaustraße 26
33102 Paderborn
Germany

This publication and the contents hereof are subject to change without notice.

ConfigurationDesk, ControlDesk, MicroAutoBox, MicroLabBox, SCALEXIO, SYNECT, SystemDesk, TargetLink and VEOS are registered trademarks of dSPACE GmbH in the United States or other countries, or both. Other brand names or product names are trademarks or registered trademarks of their respective companies or organizations.
# Table of Contents

**Modular distributed development with TargetLink** ................................................................. 1

**Table of Contents** .................................................................................................................. 3

1 **Introduction and Definitions** ................................................................................................. 4

   1.1 Introduction .......................................................................................................................... 4
   1.2 How to Work with This Guide ............................................................................................. 7
   1.3 Definitions ............................................................................................................................ 7
       1.3.1 Atomic Component and Runnable Functions .......................................................... 8
       1.3.2 Compositions and Composition Functions .................................................................. 9

2 **Modeling Atomic Components and Runnable Functions** ................................................. 11

   2.1 Modeling Atomic Components [TL_MOD_ATOMCOMP_GENERAL] .................................. 11
   2.2 Modeling Runnable Functions – Block Use [TL_MOD_RUN_BLOCKS] .............................. 14
   2.3 Runnable Functions – Block Settings [TL_MOD_RUN_SETTINGS] .................................... 16

3 **Modeling Compositions and Composition Functions** ....................................................... 19

   3.1 Modeling Compositions [TL_MOD_COMPOS_GENERAL] .................................................. 19
   3.2 Modeling Composition Functions – Block Use [TL_MOD_COMPFUN_BLOCKS] ................ 20
   3.3 Composition Functions – Block Settings [TL_MOD_COMPFUN_SETTINGS] .................... 22

4 **General TargetLink Data Dictionary Organization** ......................................................... 24

   4.1 General Data Dictionary Setup [TL_MOD_DD_SETUP] ...................................................... 24
   4.2 General TargetLink Data Dictionary Structure ................................................................. 25
   4.3 General Data Dictionary Object Specifications [TL_MOD_DD_OBJGEN] ......................... 26
   4.4 Data Dictionary Settings for Specific Objects [TL_MOD_DD_OBJSPEC] .......................... 27
   4.5 Data Dictionary Module and Ownership Handling [TL_MOD_DD_MODULES] .................. 28
   4.6 Data Dictionary setting for distributed artefacts [TargetLink_MOD_DD_ARTEFACTS] ....... 29

5 **Bibliography** ....................................................................................................................... 32
1 Introduction and Definitions

1.1 Introduction

The main objective of this guide is to define TargetLink architectural modeling patterns and guidelines for applying modular, distributed development when working with TargetLink. Modular, distributed development is a development approach where (a potentially large number of) software components is implemented in the form of modular units/components developed by different developers of potentially large teams that can also be distributed across different organizations. The key point for implementing this development approach is to correctly split potentially large models into smaller components. These components can then be developed independently up to a level where what is known as an atomic component is designed by a single developer. In later model-based software integration stages, the different atomic components are then simply integrated to form a hierarchy of model-based software components (known as compositions). Following this approach, the entire application software can be gradually built. The concept is shown in Figure 1.

This document is structured as follows: The first chapter consists of a general introduction and definitions. The second chapter emphasizes how to model and structure components. The third chapter elaborates on how to model and structure integration models. The last chapter provides information on how to organize and structure the TargetLink Data Dictionary.

This guide explains with the development of non-AUTOSAR software although our terminology and the concept closely resembles AUTOSAR terminology. For modeling AUTOAR software with TargetLink, consult the TargetLink AUTOSAR Modeling Guide.

The guidelines in this guide generally support modular, distributed, model-based development. Moreover, compliance with the guidelines enables a smooth transition if you combine TargetLink with SYNECT, dSPACE's data management and collaboration platform, to boost process efficiency.
Figure 1: Modular, distributed development for TargetLink models requires the partitioning of larger integration/composition models (top) into smaller atomic components (bottom). Each of these components is designed by one TargetLink user.

```c
void fuelratecontroller(void) {
    /* call of function: fuelratecontroller/Controller */
    Controller();

    /* call of function: fuelratecontroller/r_SensorCorrection */
    SensorCorrection();

    /* call of function: fuelratecontroller/r_AirflowCalculation */
    IntakeAirflowEstimation();

    /* call of function: fuelratecontroller/FuelCalculation */
    FuelRateCalculation();
}
```

Figure 2: Integration code, i.e., glue code that is generated for an integration/composition model (refer to the top part of Figure 1). The generated function (subsequently called composition function) simply calls previously developed functions called runnable functions (refer to Figure 3). The composition function does not contain any algorithmic code but might contain code for copying from one interface variable to another (not required in this example).
Figure 3: Algorithmic code is exclusively generated from atomic components in the form of **runnable functions** (refer to the bottom part of Figure 1).
1.2 How to Work with This Guide

The main objective of this guide is to define TargetLink architectural modeling patterns for the modeling elements described and defined in section 1.3. These elements include atomic components, runnable functions, compositions, and composition functions. The objective is to reap the benefits of modular, distributed development by using the architectural style guidelines described in this guide. Since this style guide just covers architectural patterns, it will not describe how to use TargetLink to model the interior of runnable functions, for example. For more information on TargetLink modeling, consult the TargetLink Modeling Guidelines from dSPACE (Eisemann, 2016). This guide describes only the architectural aspects of TargetLink models.

Each guideline includes the following elements:

- A **headline**, including a **unique ID in capital letters** such as [TL_MOD_COMP_BLOCKS]. The headline outlines the topic and the ID is kept unique and consistent in future versions of this document.
- A **description text** that specifies the rule with which the guideline complies. The description usually contains an enumeration with which the guideline complies that has multiple rule parts.
- A **rationale** that describes the justification and motivation for the individual parts of the guideline. If the description text contains an enumeration, so does the rationale. The separate items justify the individual rule parts of the description with identical numbers.
- An **optional example** that provides additional explanations and clarifications.

The guidelines in this guide will typically be applied in two different scenarios:

1. **Partitioning an existing large model** into smaller (atomic) components
   In this scenario, the size of a model has grown to a level that makes it increasingly hard to extend and refine the model any further in its current form. In this case, the guide can be applied to partition the large model into smaller **atomic components**, which are developed by individual developers and then later integrated into a large integration/composition model that is easier to handle.

2. **Setting-up a model-based software integration process from scratch**
   In this scenario, you want to set up a model-based development process that applies not only to **atomic components**. You also want perform at least some part of the **software integration** in a model-based fashion using integration/composition models.

For both scenarios, the rules in this guide will be equally helpful.

1.3 Definitions

In this guide, we will use the following terms, which are related to AUTOSAR terminology, to describe the relevant architectural modeling elements:

1. Atomic components and runnable functions
2. Compositions and composition function
Atomic components and runnable functions deal with the atomic component level of software development (bottom part of Figure 1), whereas compositions and composition functions deal with the integration level (top part of Figure 1).

The following definitions for atomic components, runnable functions, compositions, and composition functions are important when using TargetLink, because you have to decide how to split the entire software functionality into atomic components, runnable functions, compositions, and composition functions and therefore you must have a proper understanding of the underlying principles. The following chapters of this guide contain style guidelines for atomic components, runnable functions, compositions, and composition functions.

### 1.3.1 Atomic Component and Runnable Functions

An **atomic component** is characterized by the following properties:

- An atomic component can be designed, implemented, and tested separately and independently of the remaining software.
- The atomic component implements a well-defined set of functional and non-functional requirements. In particular, this means that requirements exist against which the atomic component can be tested.
- Each atomic component is an identifiable part of the software architecture. Specifically, each atomic component displays as an item in the software architecture (which is usually defined in a separate tool).
- Each atomic component is considered atomic, because it is not divided into smaller atomic components.
- Because of the atomic nature and the independence of the remaining software, it is possible that the design of the component is carried out by only one developer. Consequently, the size and complexity of the atomic component must support this.
- The partitioning of the entire software into atomic components is also guided by the principle that atomic components are easily reusable across multiple projects.
- The partitioning of the software into atomic components must also be guided by the principle that the software within the atomic components has a high degree of cohesion, whereas the individual atomic components have low cohesion with each other.
- An atomic component communicates with the rest of the software exclusively through well-defined interfaces, which are also a part of the software architecture (usually defined in a separate tool).
- Atomic components contain a number of runnable functions (often exactly one) that are called from outside of the atomic component.
- The software generated for atomic components consists of the runnable functions only (including their interface variables). There is no additional code that belongs to an atomic component.

The above-listed properties for **atomic components** are identical to those of an **Atomic Software Component** in an AUTOSAR use case. Although, this guide is about non-AUTOSAR software, it uses similar or even identical concepts.
A **runnable function** is characterized by the following properties (refer to Figure 3 and Figure 1 bottom part):

- A runnable function is always part of an atomic component and will be developed along with the atomic component to which it belongs. The development of an atomic component is essentially the development of its runnable functions.
- All algorithmic code of the application software is partitioned into multiple runnable functions that belong to multiple atomic components.
- The generated code for a runnable function is a C code function and thus production code (as opposed to mere stub code).
- The runnable function is externally visible, which means it will be called from outside the atomic component, e.g., from an operating system task or from a composition function, refer to 1.3.2.
- A runnable function can have ordinary C code subfunctions according to the requirements of the designer. However, these subfunctions are not externally visible and they are called exclusively from runnable functions.
- Partitioning the software into runnable functions must be carried out based on the real-time properties of the different functionalities of the software. More specifically, the execution of a runnable function is tied to specific real-time events. Very often, this is a timer event, i.e., the runnable function must be executed at a fixed sample rate. However, a runnable function can also be called because of an asynchronous event.
- The interfaces/interface variable of a runnable function has to be properly defined including their data types, scalings, etc. (For more information, refer to the chapter on General TargetLink Data Dictionary Organization)

The above listed properties for a **runnable function** are identical to those of a **Runnable (Entity)** in an AUTOSAR use case. Although, this guide is about non-AUTOSAR software, it describes similar or even identical concepts.

Guidelines regarding the modeling of an **atomic component with its runnable functions** in a TargetLink model are described in chapter 2.

### 1.3.2 Compositions and Composition Functions

A **composition** deals with software integration and is characterized by the following properties:

- A Composition is the aggregate of atomic components and/or smaller compositions. Consequently, compositions are not atomic but consist of other components, both atomic and compositions. Consequently, there can be multiple hierarchies for compositions.
- Compositions do not provide any new functionality nor do they implement any new requirements. They just aggregate/integrate previously developed functionality.
- Compositions are an identifiable part of the software architecture.
- Modeling compositions can have the following objectives:
1. Generating software integration code (glue code), i.e., the generation of composition functions to integrate previously developed functions (runnable functions and/or other composition functions).

2. Simulating the interactions of multiple (atomic) components (in MIL/SIL/PIL simulation mode), i.e., integration testing/software integration testing.

Typically, both the generation of composition functions for software integration as well as performing software integration tests is desired.

The above-listed properties for a composition are similar to those of a composition in AUTOSAR. Although, this guide is about non-AUTOSAR software, it uses similar concepts.

Modeling compositions in TargetLink is typically limited to application software. The integration with RTOS code, I/O driver code, communication stacks, etc. is typically done outside the Simulink/TargetLink environment.

A composition function deals with software integration and is characterized by the following properties (see also Figure 2 and Figure 1 top):

- A composition function is always part of a composition and will be developed along with the composition itself. In many cases, the code for a composition is only one composition function.

- A composition function is a C code function, which belongs to the production code of the software (as opposed to mere stub code).

- A composition function exclusively constitutes integration code (glue code), i.e., it serves to integrate previously developed functions, for example, runnable functions or previously developed composition functions.

- A composition function can integrate only those runnable functions/composition functions with identical or compatible real-time behavior, e.g., different runnable functions with identical sample time or an integer multiple. It is obviously not possible to combine a runnable function with a fixed sample rate with an asynchronously called runnable function in one composition function, for example.

- A composition function can theoretically be a task function of an RTOS. However, in the more common scenario, a composition function will be called either from an RTOS task or from a composition function of a higher integration level.

- The interfaces/interface variables of a composition function have to be properly defined along with their data types, scalings, etc.

Since AUTOSAR Runnables are integrated directly into operating system tasks, the only form of a Composition Function in AUTOSAR is a task of the AUTOSAR OS. There are no intermediate software integration steps (glue code generation) between the AUTOSAR Runnable level on one hand and the entire RTE for an ECU on the other.

Guidelines regarding the modeling of compositions and their respective composition functions are described in chapter 3.
2 Modeling Atomic Components and Runnable Functions

This chapter provides guidelines and advice on how to model atomic components and runnable functions as defined in subsection 1.3.1. Both are closely related because developing an atomic component means developing its runnable functions. Consequently, the same TargetLink root model/Data Dictionary is used for the development.

2.1 Modeling Atomic Components

[TL_MOD_ATOMCOMP_GENERAL]

Description

The use of Simulink/TargetLink blocks and the setup of the Data Dictionary to model atomic components must be guided by the following principles:

1. An atomic component must be developed in one TargetLink model, usually in a TargetLink Subsystem that is specific to the atomic component.

2. The model for the atomic components is associated with a TargetLink Data Dictionary project, which is also specific to the atomic component. The TargetLink Data Dictionary settings must comply with all the guidelines in chapter 4.

3. In the TargetLink subsystem of the atomic component, the runnable functions of the atomic component are modeled. Ideally, the TargetLink subsystem of the atomic component contains only
   a. the runnable functions of the atomic component.
   b. the basic wiring to connect the TargetLink subsystem top level ports with the runnable functions and potential runnable functions among each other.

4. To schedule the execution of the runnable functions, a basic mechanism, preferably one that uses Simulink function calls, must be applied. In this case, runnable functions are activated by a Simulink function call generator block or a Stateflow scheduler in the TargetLink subsystem or externally, refer to Figure 5.

5. Outside the atomic component's TargetLink subsystem, you can model anything you require, e.g., a test harness for simulating/testing the behavior of the atomic component, refer to Figure 4.

6. All code generated for the TargetLink subsystem of the atomic component that does not belong to one of the runnable functions must be labeled stub code, i.e., code that is used only for simulation purposes but is not part of the production code of the atomic component. This can be done by assigning the respective model parts with the TargetLink function block to one of the Data Dictionary’s stub code module objects, refer to chapter 4.

Rationale

1. The atomic component is the level where actual development with a full TargetLink model/TargetLink Data Dictionary takes place. The model for the atomic component itself is not reused, only the individual model parts that include the runnable functions.
2. The TargetLink Data Dictionary has to be setup in a way to support later reuse of some of its branches in TargetLink Data Dictionaries for compositions/composition functions.

3. The runnable functions essentially form the content of the atomic component and are therefore modeled in the TargetLink subsystem. The atomic component itself does not necessarily have a direct representation, e.g., in the form of a subsystem. The best way to think of the atomic component’s representation in the model is as an aggregation of systems modelling runnable functions. Keeping the model structure in the TargetLink subsystem very simple is helpful for automatic frame model generation/updating via scripts.

4. A function call activation of the runnable functions allows for great flexibility to specify arbitrary execution orders while subsequently developing models for compositions/composition functions.

5. Modeling a test harness in an open or a closed loop is helpful to perform simulations with proper stimuli to verify/validate the behavior of the atomic component.

6. Depending on the modeling style used for the interior of the atomic component’s TargetLink subsystem, the generated code might contain not only the production code for the runnable functions but also some type of stub code, e.g., for testing purposes. This code must not be used with the real production code generated for runnable functions.
Example:

Figure 4 and Figure 5 demonstrate the preferable modeling approach for an atomic component with one runnable function.

Figure 4: The preferred modeling approach for an atomic component is simple. The component is modeled in a TargetLink subsystem whose contents are shown in Figure 5. All interface signals of the atomic component cross the border of the TargetLink subsystem. Moreover, a function-call-triggering mechanism (e.g., using a function call generator block) must be used to trigger the runnable functions in the TargetLink subsystem, refer to Figure 5. A test harness that consists of simple source and sink blocks can be used to simulate and test the component.
Figure 5: The preferred modeling approach for the contents of the TargetLink subsystem of the atomic component in Figure 4: The contents of the TargetLink subsystem consists of only one runnable function whose interface ports are directly connected to the ports of the TargetLink subsystem. The runnable function is triggered by a function call, which is emitted from the function call generator block shown in Figure 4. For more information, refer to the TargetLink user documentation: Details on Implementing Externally Function-Call-Triggered Subsystems (dSPACE GmbH, 2018).

2.2 Modeling Runnable Functions – Block Use
[TL_MOD_RUN_BLOCKS]

Description
Runnable functions must be modeled according to the following design guidelines with regard to the used blocks:

1. A runnable function must be modeled either as a library subsystem or as a referenced model.
2. The runnable function is always developed in combination with the atomic component to which the runnable function belongs. Therefore, the library subsystem/referenced model for the runnable function is developed in the TargetLink subsystem for the atomic component as described in 2.1.
3. The runnable function has to be atomic, i.e., if it is modeled as a subsystem, the subsystem has to be made atomic (a referenced model is always atomic).
4. It is recommended to make the runnable function not only atomic but to model it as a function-call-triggered system, either as function-call-triggered subsystem or as function-call-triggered referenced model.
5. The runnable function must contain a TargetLink Function block.
6. All inputs and outputs of the runnable function must be modeled as TargetLink inports/outports for normal signals or as TargetLink BusInports/BusOutports for busses.
input/output signals of a runnable function must be read/written only conditionally, Simulink ports can also be used at the top layer of the runnable function. However, the first use of a signal in the runnable function requires a TargetLink port.

7. The runnable function must be self-contained so that its reuse in other models does not rely on the existence of additional blocks outside the runnable function, e.g., a Data Store Memory block.

8. In the runnable function system, the actual algorithmic functionality of the runnable function is modeled with TargetLink blocks. Different subsystem hierarchies can be introduced whenever necessary, including TargetLink function blocks for subfunctions of the runnable function (the subfunctions are not runnable functions).

Rationale

1. Library subsystems and model referencing are proper methods to establish a single-source principle for models. This is important, because runnable functions will be used in more than one TargetLink model, e.g., in models for compositions/composition functions.

2. Runnable functions are always developed in the context of atomic components to which they belong. Hence, the runnable function is included in the TargetLink subsystem for the atomic component.

3. A C code function for a runnable function is inherently atomic. To ensure that MIL simulations yield identical simulation results as SIL simulations, the modeling construct for a runnable function must also be atomic.

4. Triggering a runnable function via function call simplifies subsequent integration steps, because function calls support flexibly controlling the execution order of runnable functions in integration/composition models.

5. The TargetLink function block is necessary for TargetLink to generate a separate C function for the runnable function.

6. The runnable function must have properly defined interfaces with which it communicates with external components. This is why TargetLink ports are required at the runnable function’s interface or at least the first time, a signal is used in the runnable function.

7. For later reuse of the runnable function in a composition/composition function or in other projects in general, it is important that the runnable function can be easily extracted and integrated. For this purpose, the library subsystem/referenced model of the runnable function must be self-contained.

8. In the interior of the runnable function, you can model to your requirements as long as the interface of the runnable function system is kept consistent with the above guidelines. Subfunctions themselves are not runnable functions, because they are called from within the runnable function and not from outside.

Example:

Figure 6 shows the modeling of a runnable function.
Figure 6: A runnable function is modeled as a function-call-triggered library subsystem or referenced model. All ports entering or leaving the runnable function have to be TargetLink ports or TargetLink Bus ports.

2.3 Runnable Functions – Block Settings
[TL_MOD_RUN_SETTINGS]

Description

Settings for the used blocks of the runnable functions must comply with the following guidelines:

1. The TargetLink Function block of the runnable function must reference a TargetLink Data Dictionary Module object. The specified module must contain only code for the runnable function or (although not preferred) other runnable functions of the same atomic component.

2. The TargetLink Function block of the runnable function preferably has the ‘incremental code generation’ property checked. If you use referenced models (as opposed to subsystems), this is mandatory.

3. Preferably, the TargetLink function block of the runnable function references a TargetLink Data Dictionary Signature object (available since TargetLink 4.0), refer to Figure 8.

4. All block variables contained in the runnable function that have cross-project relevance and have to be accessed by other developers (the inports, outports, calibration parameters, and measurable signals of the runnable function) must be managed in the TargetLink Data Dictionary and referenced from the TargetLink block dialogs. These block variables constitute the Interface Variables of the runnable function and must be represented either as TargetLink Data Dictionary variable objects or as Data Dictionary ReplaceableDataItems (RDI). To establish a connection between the TargetLink Data Dictionary RDI object and a variable to which the
RDI is mapped at a later point, you use TargetLink Data Dictionary DataItemMapping objects. For the naming and organization of these TargetLink Data Dictionary objects, refer to the guidelines in chapter 4.

5. If interface variables require initialization values (e.g., if they are calibration parameters), they can be specified either together with the objects in the TargetLink Data Dictionary or in M files.

**Rationale**

1. Referencing a TargetLink Data Dictionary Module object ensures that the code for the runnable function is generated to the desired file. Moreover, avoid mixing potential TargetLink-generated stub code for the atomic component and the actual production code for the runnable function.

2. If you use incremental code generation, this simplifies the reuse of a fully tested and verified runnable function code in composition/composition functions (as well as in future projects). Incremental code generation ensures that the code is not generated again when the runnable function is integrated in another composition model and not modified.

3. Referencing DD Signature objects enables interface checks between TargetLink Data Dictionary Signature objects on the one hand and the inports/outports in the model on the other.

4. If you use TargetLink Data Dictionary objects for interface specifications and links between TargetLink Data Dictionary variable objects and TargetLink Data Dictionary requirement objects, this simplify the exchange with other tools, in particular architecture tools/data dictionaries. Moreover, the organization in the TargetLink Data Dictionary provides a proper view of the architectural aspects of the runnable function and simplifies tracing/navigation between architecture and TargetLink model. Data Dictionary variables as well as replaceable data items both describe a software interface and can therefore both be used.

5. It is preferable to organize initialization values/calibration values in a centralized manner, i.e., either in the Data Dictionary or in M files.

**Example**

Figure 7 shows the use of Data Dictionary variable objects for interface specifications of the runnable function. Figure 8 shows the use of Data Dictionary Signature objects, which is a preferred option.
Figure 7: All signals/variables with cross-project relevance, such as the interface signals of a runnable function, must be specified in the Data Dictionary, i.e., either as Data Dictionary variables or as replaceable data items.

Figure 8: Left: Specification of function signatures in the Data Dictionary supported as of TargetLink 4.0. Right: Referencing a function signature in the TargetLink function block. As a result, the interface ports of the subsystem can be generated, updated, or checked against the TargetLink Data Dictionary Signature object automatically. Moreover, the Module property of the TargetLink Function block must reference a DD Module object to properly define the C code module to which the code is generated.
3 Modeling Compositions and Composition Functions

This chapter focuses on the modeling of compositions and composition functions. Both compositions and composition functions focus on the integration of previously developed atomic components and their runnable function. This level structures the model and hence the software.

3.1 Modeling Compositions

[TL_MOD_COMPOS_GENERAL]

Description

Guidelines for composition models are almost identical to those for atomic components. The only difference is that instead of runnable functions they consider composition functions. Consequently, Simulink/TargetLink blocks and the setup of the TargetLink Data Dictionary to model compositions must be guided by the following principles:

1. A composition must be developed in one TargetLink model, usually in one TargetLink subsystem which is specific to the composition.
2. The model for the composition is associated with a Data Dictionary project, which is then also specific to the composition. The Data Dictionary settings must comply with all the guidelines in chapter 4.
3. In the TargetLink subsystem of the composition, composition functions are modeled. Ideally, the TargetLink subsystem of the composition contains only
   a. the composition functions of the composition
   b. the basic wiring for connecting the TargetLink subsystem with the composition functions and potentially composition functions with each other
4. The preferred modeling approach is a composition that consists of only one composition function, which is directly connected to the ports of the TargetLink subsystem.
5. To schedule the execution of the composition functions, a basic mechanism, preferably using Simulink function calls, must be applied. In this case, composition functions are activated by a Simulink function call generator block or a Stateflow scheduler in or outside of the TargetLink subsystem.
6. Outside the composition's TargetLink subsystem, you can model as required, e.g., a test harness for simulating/testing the behavior of the composition.
7. All code generated for the TargetLink subsystem of the composition that does not belong to one of the composition functions, must be labeled as stub code, i.e., code that is used for simulation purposes only but does not belong to the production code of the composition. This can be done by assigning the respective model parts with the TargetLink Function block to one of the Data Dictionary's stub code Module objects, refer to chapter 4.

Rationale

1. Composition is the level where actual development with a full TargetLink model/TargetLink Data Dictionary takes place. This model for the composition is not reused, only the individual model parts with the composition functions.
2. The TargetLink Data Dictionary has to be setup in a way to support later reuse of some of its branches in TargetLink Data Dictionaries for compositions/composition functions of a different hierarchy level.

3. The composition functions essentially form the content of the composition and therefore they are modeled in the TargetLink subsystem. Keeping the model structure in the TargetLink subsystem simple is helpful for automatic frame model generation/updating via scripts.

4. A simple modeling style is beneficial for tool automation, e.g., for automatic interface checks or automatic wiring.

5. A function call activation of the composition functions allows for great flexibility to specify arbitrary execution orders and subsequently developing models for compositions/composition functions of a higher integration level.

6. Modeling a test harness in an open or closed loop is helpful for performing simulations with proper stimuli to verify/validate the composition.

7. Depending on the modeling style used within the composition’s TargetLink subsystem, the generated code might contain not only the production code for the composition functions but also some sort of stub code, e.g. for testing purposes. This code should not be included with the real production code generated for composition functions.

3.2 Modeling Composition Functions – Block Use

[TU_MOD_COMPFUN_BLOCKS]

Description

Guidelines for composition functions are very similar to those for runnable functions. The difference is that the content of composition functions integrates and schedules previously developed functions but no additional algorithmic functionality. Consequently, composition functions must be modeled according to the following design guidelines with regard to the blocks used:

1. A composition function must be modeled either as a library subsystem or as a referenced model.

2. The composition function is always developed in combination with the composition to which the composition function belongs. Therefore, the library subsystem/referenced model for the composition function is developed in the TargetLink subsystem for the composition as described in 3.1.

3. The composition function has to be atomic, i.e., if it is modeled as a subsystem, the subsystem has to be made atomic (a referenced model is always atomic).

4. It is recommended that you make the composition function not only atomic but you model it as a function-call-triggered system, either as function-call-triggered subsystem or as function-call-triggered referenced model.

5. The composition function must contain a TargetLink function block.

6. All inputs and outputs to the composition function must be modeled as TargetLink inports/outports for normal signals or as TargetLink BusInports/BusOutports for busses.
7. Inside the composition function, only the following functionality must be modeled, refer to Figure 9:
   a. Previously developed runnable/composition functions that are to be integrated in the composition function. Only these runnable/composition functions can be integrated in the same composition function that have compatible real-time properties, e.g., are executed at the same sample rate.
   b. A basic scheduler to trigger the execution of the functions that you want to integrate in the desired order.
   c. The basic connection between the different functions that you want to integrate and the connections to the TargetLink subsystem.

Rationale

1. Library subsystems and model referencing are proper ways to establish a single-source principle for models. This is important, because composition functions might be reused in more than one TargetLink model, e.g., in models for compositions of a higher integration level.
2. Composition functions are always developed in the context of the composition to which they belong. Hence, they are included in the TargetLink subsystem for the composition.
3. A C code function for a composition function is inherently atomic. To ensure that MIL simulations yield identical simulation results compared to SIL simulations, the modeling structure for a composition function must be atomic also.
4. Triggering a composition function through a function call simplifies subsequent integration steps, because function calls support very flexible ways to control the execution order of composition functions in integration/composition models of a higher hierarchy level.
5. The TargetLink Function block is required to generate a C function for the composition function.
6. The composition function has to have properly defined interfaces with which it communicates externally. Therefore, TargetLink ports are required at the composition function's interface.
7. The sole purpose of the composition function is to integrate previously developed functions with compatible real-time properties to form a new atomic unit.
Figure 9: The purpose of modeling a composition function is to integrate existing runnable/composition functions into a larger integration model for glue code generation and testing. Consequently, there is no wiring between the different runnable functions. A scheduler triggers the execution of the different runnable/composition functions and TargetLink ports at the border of the composition function system to generate a new function.

3.3 Composition Functions – Block Settings
[TL_MOD_COMPFUN_SETTINGS]

Description

Guidelines for the settings of the blocks of a composition functions are very similar to those of runnable functions. Consequently, composition functions must be modeled according to the following design guidelines with regard to the settings in the blocks used:

1. The TargetLink Function block of the composition function must reference a TargetLink Data Dictionary Module Object. The specified module must contain only code for the composition function itself or (although not preferable) other composition functions of the same composition.

2. Preferably, the TargetLink Function block of the composition function has the “incremental code generation” property checked. When you use referenced models (as opposed to subsystems), this is a mandatory requirement.

3. Preferably, the TargetLink Function block of the composition function must reference a TargetLink Data Dictionary Signature object (available as of TargetLink 4.0).

4. The interface variables of the composition functions, i.e., the inports/outports of the composition function must be managed in the Data Dictionary and referenced from the
Modular distributed development with TargetLink

TargetLink block dialogs. The interface variables can be implemented as Data Dictionary Variable objects or as Data Dictionary replaceable data items. For the naming and organization of these Data Dictionary objects, refer to the guidelines in chapter 4. Moreover, these TargetLink Data Dictionary objects must be referenced from the TargetLink Data Dictionary Signature object.

Rationale

1. Referencing a Target Link Data Dictionary Module object ensures that the code for the composition function is generated to the desired file. Moreover, avoid mixing potential TargetLink generated stub code for the Composition and the actual production code.

2. The use of incremental code generation simplifies a potential later reuse of a fully tested and verified composition function code in composition functions of a higher integration level.

3. Referencing TargetLink Data Dictionary Signature objects enables interface checks between TargetLink Data Dictionary Signature objects and the Inports/Outports in the model.

4. Using TargetLink Data Dictionary objects for interface specifications simplifies the exchange with other tools, in particular architecture tools/Data Dictionaries. Moreover, the organization in the TargetLink Data Dictionary provides a proper view of the architectural aspects of the composition function and simplifies tracing/navigation between architecture and TargetLink model. TargetLink Data Dictionary variables as well as replaceable data items both describe a software interface and can therefore both be used.
4 General TargetLink Data Dictionary Organization

This chapter describes guidelines for organizing of TargetLink Data Dictionaries for atomic components as well as compositions that have to be observed. Several of these guidelines were already referenced in previous chapters.

4.1 General Data Dictionary Setup

[TL_MOD_DD_SETUP]

**Description**

The general organization of Data Dictionaries for atomic components as well as composition functions must be based on the following principles:

1. Since Data Dictionary objects referenced from blocks within runnable functions are reused in composition models later during software integration, there has to be a mechanism to share these objects between data dictionaries for atomic components on the one hand and data dictionaries for compositions on the other. To establish a single source principle for this data, the objects must either be managed in a central database, such as dSPACE SYNECT, or shared using the Data Dictionary include file mechanism.

2. For each project, there are a number of Data Dictionary objects that are must be identical for all Data Dictionaries in the project, both for atomic components or compositions. These TargetLink Data Dictionary objects are typically TargetLink Data Dictionary Variable Classes, TargetLink Data Dictionary Function Classes, TargetLink Data Dictionary Units, TargetLink Data Dictionary Templates, and TargetLink Data Dictionary Code Generation options. In order to make sure that these are kept identical in all Data Dictionaries of the project, a mechanism for sharing them must be in place, e.g., using the Data Dictionary include file mechanism, refer to Figure 10.

3. Rather than using the predefined variable classes and function classes that are provided directly in the TargetLink Data Dictionary template files, it is beneficial to use these variable and function classes as starting point to create user-specific variable/function classes and apply them to models rather than using the TargetLink-provided templates, refer to Figure 10. The only exceptions are pre-defined variables, variable classes, and function classes for AUTOSAR code generation.

**Rationale**

1. Reusing the same Data Dictionary object across different Data Dictionaries requires a single-source principle for data administration. This helps to make sure that the data integrity of Data Dictionary objects is maintained.

2. To keep general information (e.g., Data Dictionary variable classes, etc.) on a project consistent throughout the project, this information must be reused in all TargetLink Data Dictionaries due to the TargetLink Data Dictionary include mechanism.

3. User-specific variables and function classes provide more flexibility if they have to be changed within the same or across different projects.

**Example**
Figure 10 shows the use of the TargetLink Data Dictionary include file mechanism to share user-specific variable classes and function classes.

Figure 10: It is generally recommended to use user-specific variable/function classes in models instead of those directly provided by TargetLink. This ensures more flexibility to simply adapt the settings for different projects. There must be a mechanism in place to share the specifications among all Data Dictionaries in the project, e.g., by using the Data Dictionary include file mechanism.

4.2 General TargetLink Data Dictionary Structure

Description

Generally, there are different approaches to handle and structure data in the TargetLink Data Dictionary. If this structure is tried and proven, consider changes only if required. A possible starting point to organize the TargetLink Data Dictionary is the following structure (for examples, refer to Figure 11):

1. Generally, variables must be organized in TargetLink Data Dictionary VariableGroups named after the component in which they are used.

2. Interface variables must be defined in the TargetLink Data Dictionary from which the variable originates, i.e., where the data flow leaves the component. Components consuming this variable should include the respective variable from the originating TargetLink Data Dictionary.

3. Variables used as a parameter must be defined in the TargetLink Data Dictionary of the component in which they are used.
4. The TargetLink Data Dictionary objects Typedefs and Scalings must be structured in groups regarding the project in which they are used.

**Rationale**

1. The structure in component groups enables you to link the variables directly to the specific component.
2. Structuring the interface variables in outport groups helps you identify from which component the signal originates and if the specific variable is used as an interface variable.
3. Structuring the parameter variables in parameter groups helps you distinguish interface variables from parameter variables.
4. These objects are mostly defined across projects and must be distinguished from generic TargetLink Data Dictionary typedefs and scalings.

![Data Dictionary Navigator](image)

**Figure 11**: Example structure for variables, typedefs and scalings into various groups

### 4.3 General Data Dictionary Object Specifications

**[TL_MOD_DD_OBJGEN]**

**Description**

The organization of Data Dictionaries for atomic components as well as compositions functions must be based on the following principles:

1. The names of all Data Dictionary objects must be valid C identifiers. This is highly important for TargetLink Data Dictionary variable objects, TargetLink Data Dictionary Typedef objects and replaceable data item objects but must be observed in general.
2. If TargetLink Data Dictionary objects of the same object kind (such as TargetLink Data Dictionary variable objects, TargetLink Data Dictionary typedef objects, TargetLink Data Dictionary scaling objects) have different properties, then they must always have a different object name (not only a different TargetLink Data Dictionary path). The object name must be unique across all Data Dictionaries of the project.
TargetLink Data Dictionary object properties are shown in the Property Value List of the Data Dictionary Manager. TargetLink Data Dictionary properties are distinct from TargetLink Data Dictionary object attributes, such as TargetLink Data Dictionary object name, TargetLink Data Dictionary object path, TargetLink Data Dictionary access rights, etc.

3. Ideally, the same Data Dictionary object (meaning same object name and same object properties) also has the same Data Dictionary path across all Data Dictionaries of the project. This means, situations must be avoided where the same Data Dictionary object is stored at different Data Dictionary paths in different Data Dictionary projects. If you deviate from this guideline, use a central data management in order to manage the different TargetLink Data Dictionary path with the respective TargetLink Data Dictionary object such as TargetLink Data Dictionary variables or replaceable data items.

Rationale

1. Valid C identifiers for object names make it possible to use the names of TargetLink Data Dictionary objects directly in the generated code without TargetLink having to adapt the name. This simplifies traceability.

2. To avoid confusion, all Data Dictionary objects must be distinguishable by their Data Dictionary object names, not only by their Data Dictionary paths.

3. Avoiding different Data Dictionary paths across Data Dictionary for the same source object helps to avoid confusion and helps to maintain the single-source principle.

4.4 Data Dictionary Settings for Specific Objects

For Data Dictionary Variable objects, Data Dictionary replaceable data items and Data Dictionary typedef objects, the following settings should be observed:

1. All Data Dictionary variables, replaceable data items (whether for runnable functions or composition functions) as well as typedef objects must have their respective Module property set, refer to Figure 12. The Module property assigns these symbols to a particular C module where they are defined.

2. For Data Dictionary variable objects, the Storage property of their respective variable classes must not be set to “external”. Instead, external variables should be either assigned to external TargetLink Data Dictionary Modules or Data Dictionary stub code Modules.

3. Interface Variables of runnable functions and composition functions, i.e., TargetLink Data Dictionary Variable objects and replaceable data items must be referenced from TargetLink Data Dictionary Signature objects (available since TargetLink 4.0).

Rationale

1. Assigning all types of code variables to the proper C modules is important for distributed development to ensure that the variables are instantiated exactly once, although they might be used in multiple TargetLink models.
2. TargetLink's mechanism for stub code generation or the inclusion of external modules eliminates the need for specifying individual variables as "external". Using external modules allows performing SIL simulation with real legacy code whereas TargetLink's stub code generation mechanisms can be used to generate a stub version of an external file.

3. Using signature objects allows you to easily check and update Data Dictionary interface specifications with the inports/outports of the runnable function in the model.

![Figure 12](image)

**Figure 12:** Variable objects for runnable functions/composition functions must have the `ModuleRef` property set to assign them to the C code module to which they belong.

### 4.5 Data Dictionary Module and Ownership Handling

[TL_MOD_DD_MODULES]

**Description**

Distributed function development is possible by partitioning models. TargetLink provides different ways to do this, i.e., it enables you to partition your models using model referencing, let other function developers work on subsystems and perform code generation for them incrementally, or generate component-based production code straight from the TargetLink Data Dictionary. In order to partition the project modules and hence their owner must be specified in the TargetLink Data Dictionary. Therefore, the following guidelines must be applied when using Data Dictionary Modules and Module Ownership objects:

1. Each Data Dictionary project must contain one of the following, regardless of whether it is intended for atomic components or compositions:
   a. A stub code ModuleOwnership object, i.e., a Data Dictionary ModuleOwnership object that has its subsystem property set to a unique name, such as xxx_stubcode_xxx
   b. A production code ModuleOwnership object for each production code generation unit, i.e., an incrementally generated subsystem, a referenced model, or a TargetLink subsystem. The subsystem property must have the same name as the referenced model/incrementally generated subsystem/TargetLink subsystem.

2. For each C code file to be generated, there must be a Data Dictionary Module object.
3. In order to properly assign all model parts to a C code module, all TargetLink Function blocks and AddFile Blocks must reference a Data Dictionary Module object.

4. Modules which are just stub code need to be referenced from the stub code ModuleOwnership objects. Each production code Modules need to be referenced from exactly one of the production code ModuleOwnership objects.

5. Parameters being shared with multiple software components are owned by a single TargetLink Data Dictionary ModuleOwner. Hence the module owner must be a separate module exclusively for parameters.

6. Interface variables can be owned by the module from which the signal originates or by multiple module. For the latter case the interface variable must be specified as a mergeable variable.

Rationale

1. All C code modules belong to either production code or stub code. TargetLink uses ModuleOwnership objects to distinguish between them. Stub code modules are generated by TargetLink for simulation/testing purposes only and they are stored in folders separate from the production code.

2. If you use Data Dictionary Module objects for all C code modules provides transparency as to which files have to be generated.

3. The assignment of different parts in a model to specific C modules hast to be controlled appropriately.

4. If you use the assignment of all C modules to module ownership objects allows you to correctly specify, which TargetLink model part generates the real production code of a file as opposed to mere stub code.

5. The specification of a single module owner for a parameters separates the parameter definition from the implementation and hence sharing of the parameter files becomes feasible.

6. If the interfaces are specified for the component from which they originate allows for a direct view on the interfaces and avoids naming conflicts etc.

4.6 Data Dictionary setting for distributed artefacts

[TargetLink_MOD_DD_ARTEFACTS]

Description

Generally, there is a folders and file structure of a software project in a certain way in order to simplify version control, reusability and general organization of a project. Therefore, following guidelines may be applied to the projects. There is no need to change a folder and file structure when proven in practice. Here, the structure of the file structure should be one of the two alternatives that were already proven in various software projects:

1. Hierarchically folder structure representing the model structure with atomic software components and compositions.
   a. An appropriate naming convention must be introduced in order to differentiate atomic software component from compositions.
2. Plain folder list – both atomic components and compositions are stored parallel in plain folder structure in a project folder.
   
a. An appropriate naming convention must be introduced in order to differentiate atomic software component from compositions.

Rational

1. This representation form is identical to the structure of the model. In a folder group of a composition the composition model and also the used atomic components are stored. This is true for every integration level.
   
a. Easy to navigate and understand the structure on the hard drive as it represents the model structure
   
b. Difficult to master reuse use cases as sharing mechanisms of a version control system must be considered. In a reuse use case the atomic software components must remain in the same place, even though used in a different composition model which by definition must invoke sharing (for both version control system and files on the hard drive) of the atomic software component in the corresponding folder of the composition.

2. Direct plain view on all models of a software with both compositions and atomic components for every integration level.
   
a. Easy to reuse models without sharing mechanisms in version control system. The atomic software component naturally remain in the same storage place on the hard drive only links in the composition model must be properly set.
   
b. Difficult to navigate through the software project as now model structure is represented by the folder structure.

Figure 13: TargetLink Data Dictionary feature for file and folder location specification allows for distributed modular development of atomic software component and their integration in compositions with location on the file system different from the TargetLink default settings.
Example

**Figure 14:** Folder and file structure correspond to model structure. Here, the composition folder consists of all model files and other related artefacts of the composition (marked in dark blue) and also all folders of the atomic software components (marked in light blue). All project related artefacts are stored in a project specific folder that remains on root level (marked in red).

**Figure 15:** All models – both composition (marked in dark blue) and atomic software components (marked in light blue) are parallel stored in one project related folder. Project related artefacts are stored in a project specific folder (marked in red).
5 Bibliography
