ASCET-SE V6.2
User's Guide

ETAS

Copyright

Contents

1   Introduction

ASCET Software Engineering (ASCET-SE) is a tool for:

- generating target-specific C code for selected microcontrollers;
- integrating the code with a target operating system or run-time environment; and
- (optionally) invoking the target-specific compiler and linker to generate an executable application and calibration configuration file (e.g. for use with ETAS' INCA tool).

In this user guide you will learn how to:

- take models developed in ASCET-MD and define the attributes required by ASCET-SE to convert those models to C code.
- define the real-time requirements of your system and how those requirements are realized on the target microcontroller
- integrate 3rd party C code with ASCET generated code
- understand the code ASCET generates
- build models in an efficient way

1.1   About this Document

1.1.1   Target Audience

This ASCET-SE User's Guide is a supplement to the ASCET documentation (Getting Started and online help). You should be familiar with the basic features and operation of ASCET before attempting to understand code generation.

This guide assumes you have:

1. a basic understanding of the C programming language
2. experience of compiling and linking C programs for embedded microcontrollers
3. knowledge of the target microcontroller.

1.1.2   Document Structure

The remainder of this manual is structured as follows:

Chapter	Contents
Chapter 2	Safety hints regarding the use of ASCET-SE
Chapter 3	An overview of how to get started with ASCET-SE and a description of the contents of the installation
Chapter 4	Explains how to configure the implementation of model elements so that code can be generated.
Chapter 5	Explains how to configure ASCET-SE for C code generation, how the compilation and build process is controlled and how it can be customized.
Chapter 6	Describes how to provide the service routines required by ASCET-SE to do interpolation in characteristic tables

Chapter 7	Explains how ASCET-SE configured to generate code to integrate with an operating system to provide real-time scheduling of the application.
Chapter 8	Shows how to generate an ASAM-MCD-2MC A2L file for use in ECU calibration.
Chapter 9	Explains how to integrate hand-written C code with ASCET-SE, to either call or be called by ASCET-SE at runtime, and how to integrate that code with the ASCET build process.
Chapter 10	Provides some modelling hints that help ASCET-SE generate optimal code.
Chapter 11	Describes how to migrate a project from an existing target to an new target.
Chapter 12	Explains the design choices and issues involved when using quantized (fixed point) arithmetic.
Chapter 13	Explains the principles by which ASCET-SE generates code, the structure of the generated source code and provides a reference to how each part of a model is converted to C code.
Chapter 14	Provides a technical overview of how ASCET-SE works.
Chapter 15	Describes the restrictions of ASCET-SE code generation.
Chapter 16	Explains how to contact ETAS for technical support.

1.1.3   Conventions

The following typographic conventions are used:

Select **File → Open**.	Menu commands are shown in **blue boldface**.
Click **OK**.	Buttons are shown in **blue boldface**.
Press <ENTER>.	Keyboard commands are shown in angled brackets and capitals.
The "Open File" dialog window opens.	Names of program windows, dialog windows, fields, etc. are shown in quotation marks.
Select the file `setup.exe`.	Text in drop-down lists on the screen, path- and file names, program code, C type names and C functions and ASCET-SE API call names all appear in an `monospaced typeface` (`Courier`)
A *distribution* is always a one-dimensional table of sample points.	General emphasis and new terms are set in *italics.*
The OSEK group (see http://www.osek-vdx.org/) has developed certain standards.	Links to internet documents are set in blue, underlined font.

Important notes for the users are presented as follows:

> **Note**
>
> *Notes like this contain important instructions that you must follow carefully in order for things to work correctly.*

1.2   Installation

The installation of ASCET-SE is described in the ASCET installation guide.

Like all ETAS products, ASCET-SE requires a valid license file. The entitlement letter provides an URL from where a license file can be obtained. Licenses are installed and managed using the ETAS License Manager.

You can choose to install ASCET-SE in the *Silent mode*; see the ASCET installation guide, chapter "Command Line Installation". To select the target(s) to be installed, you can either define environment variables or edit the `[SilentInstallation]` section of the `install.ini` file.

If you want to use environment variables, you must define them in your environment before running the ASCET-SE installation program. The easiest way to do this is to write a batch file like this:

```
setlocal
set TRG_ANSI=true
set TRG_C16X_CLASSIC=false
set TRG_C16X_VX=false
set TRG_XCV2_VX=false
set TRG_TRICORE=false
set TRG_FFMC16LX=true
set TRG_HC12M=false
set TRG_HCS12XM=false
set TRG_HCS12XC=false
set TRG_MPC55XX=true
set TRG_MPC56X=false
set TRG_NEC850=false
set TRG_SH2A=false
set TRG_TMS470=false
set TRG_EHOOKS=false
set TRG_SELF_CONTAINED_MODE=true
ASCET-SE.exe /S
endlocal
```

Each variable denotes an ASCET-SE target. If set to `true` the target will be installed. If set to `false` the target will **not** be installed. If a target is not specified then `true` is assumed by default.

`TRG_SELF_CONTAINED_MODE` controls whether or not targets share common files. If set to `true`, each installed target directory (`trg_*`) will include a copy of all the common target files. You should choose this option if you plan to make target-specific changes to the common files.

If set to `false`, the common target files are installed in a shared common directory called `common-se`. You should choose this option if you want any changes in the common files to apply for all installed targets.

Instead of setting environment variables, you can configure installation parameters in the `install.ini` file. To do so, define the following entries in the `[SilentInstallation]` section:

```
[SilentInstallation]
TRG_ANSI=true
TRG_C16X_CLASSIC=false
TRG_C16X_VX=false
TRG_XCV2_VX=false
TRG_TRICORE=false
TRG_FFMC16LX=true
TRG_HC12M=false
TRG_HCS12XM=false
TRG_HCS12XC=false
TRG_MPC55XX=true
TRG_MPC56X=false
TRG_NEC850=false
TRG_SH2A=false
TRG_TMS470=false
TRG_EHOOKS=false
TRG_SELF_CONTAINED_MODE=true
```

Values set in `install.ini` override environment variables.

1.3 Abbreviations and Definitions

ASAM-MCD

Association for **S**tandardisation of **A**utomation- and **M**easuring Systems, with the working groups **M**easuring, **C**alibration, **D**iagnosis

ASAM-MCD-2MC file

Standard exchange format for program descriptions for calibration purposes.

ASCET

Development tool for control unit software

ASCET-MD

ASCET **M**odeling and **D**esign

ASCET-SE

ASCET **S**oftware **E**ngineering – integration package for microcontroller targets; allows the generation of an executable application for the target (control unit) with ASCET.

AUTOSAR

Automotive **O**pen **S**ystem **Ar**chitecture; see http://www.autosar.org/

BD

Block **D**iagram

BDE

Block **D**iagram **E**ditor

BLOB

> **B**inary **l**arge **ob**ject, interface-specific description data provided in ASAM-MCD-2MC files.

Class

> A class is one of the component types in ASCET. Classes in ASCET are comparable to object-oriented classes. The functionality of a class is described by methods.

Code Generation

> Code generation is the first step in the conversion of a physical model to executable code. The physical model is transformed into ANSI C code. Since the C code is partly compiler (and therefore target) dependent, different code for each target is produced.

Component

> A component is the basic unit of reusable functionality in ASCET. Components can be specified as classes, modules, or state machines. Each component is built up of elements which are combined with operators to build up the functionality.

CPR

> **C**ode **P**roduction **R**ules

ECCO

> **E**mbedded **C**ode **C**reator and **O**ptimizer

ECU

> **E**lectronic **C**ontrol **U**nit

ESDL

> **E**mbedded **S**oftware **D**escription **L**anguage

ETK

> Emulator test probe (German: **E**mulator-**T**est**k**opf)

Implementation

> An implementation describes the transformation of the physical specification (model) to executable fixed point code. An implementation consists of a (linear) transformation formula, a limiting interval for the model values, and further information (as memory assignment) where necessary.

Implementation Cast

> Element that provides the users the possibility to control the implementations of intermediate results in arithmetic chains without changing the physical representation of the elements in question.

Implementation Data Types

> Implementation data types are the data types of the underlying C programming language, e.g. `unsigned byte` (`uint8`), `signed word` (`sint16`), `float`.

Implementation Types

> Implementation types offer the user the possibility to define implementation once at the center of the project, and assign them as often as needed.

INCA

INtegrated **C**alibration and **A**cquisition Systems

Literal

A literal is used in the descriptions of components. A literal contains a string that is interpreted as a value, e.g. as a continuous or logical variable.

Memory class

A memory class is the name of the abstract memory area where a quantity is placed later in the electronic control unit.

Message

A message is a real-time language construct in ASCET for protected data exchange between concurrent processes.

Method

A method is part of the description of the functionality of a class in terms of object-oriented programming. A method has arguments and one return value.

Module

A module is one of the component types in ASCET. It describes a number of processes that can be activated by the operating system. A module cannot be used as a subcomponent within other components.

OIL

OSEK **I**mplementation **L**anguage

OS

Operating **S**ystem

OSEK

Working group "open systems for electronics in automobiles" (German: Arbeitskreis **O**ffene **S**ysteme für die **E**lektronik im **K**raftfahrzeug)

OSEK operating system

Operating system conforming to the OSEK standard.

Parameter

A parameter (characteristic value, curve, or map) is an element whose value cannot be changed by the calculations executed in an ASCET model. It can, however, be calibrated during an experiment.

Priority

Each OS task has a priority, represented by a number. The higher the number, the higher the priority. The priority determines the order in which the tasks are scheduled.

Process

A process is program function called from an operating system task. Processes are specified in ASCET modules and do not have any arguments or return values. Inputs to and outputs from a process are handled by messages.

Project

A project describes an entire embedded software system. It contains components which define the functionality, an operating system specification, and a binding system which defines the communication.

RAM

Random **A**ccess **M**emory

RE

Runnable **E**ntity; a piece of code in an SWC that is triggered by the RTE at runtime. It corresponds to the process concept in ASCET.

Resource

A resource is used to model parts of an embedded system that can be used only mutually exclusively, e.g. timers. When such a part is accessed, it has to be reserved; after executing its task, it is released again. These reservations and releases are done using resources.

ROM

Read **O**nly **M**emory

RTA-OSEK

ETAS' OSEK-compatible Real-Time Operating System.

RTA-RTE

ETAS' implementation of the AUTOSAR Run-Time Environment.

RTE

AUTOSAR Run-Time Environment which provides the interface between software components, basic software, and operating systems.

Scheduling

Scheduling is the assigning of processes to tasks, and the definition of task activation by the operating system.

Scope

An element has one of two scopes: local (only visible inside a component) or global (defined inside a project).

SM

State **M**achine

SWC

Atomic AUTOSAR **s**oft**w**are **c**omponent; the smallest non-dividable software unit in AUTOSAR.

Target

The hardware a program or an experiment runs on. In ASCET-SE, a target is specific to a combination of a microcontroller and compiler.

Task

A task is the entry point for functionality that is scheduled by an OS. Attributes of a task are its priority, its mode of scheduling and its operating mode. The functionality of a task in ASCET-SE is defined by a collection of processes. When a task runs the processes of a task are executed in the specified order.

Trigger

> A trigger activates the execution of a task (in the scope of the operating system) or a state machine action.

Type

> In an ASCET model, variables and parameters can have various types: `cont` (continuous), `udisc` (unsigned discrete), `sdisc` (signed discrete) or `log` (logic). `Cont` is used for physical quantities that can have any value; `udisc` for positive integer values, `sdisc` for negative integer values; and `log` is used for Boolean values (`true` or `false`). These types are not the same as the data types generated in the code.

Variable

> A variable is an element that can be read and written during the execution of an ASCET model. The value of a variable can also be measured with the calibration system.

2   Safety Hints for Application Software Design

ASCET and ASCET-SE provide numerous mechanisms to ensure safe and consistent microcontroller code. Some details, however, cannot be checked by the code generator. This may be the case due to technical reasons or because the correctness of an implementation cannot be clearly determined in certain cases (e.g. because the correctness is related to the usage of a model).

This chapter describes some general points that should be paid attention to when designing application software in ASCET.

Please adhere to the Product Liability Disclaimer (ETAS Safety Advice) and to the following safety instructions to avoid injury to yourself and others as well as damage to your property.

2.1   Labeling of Safety Instructions

The safety instructions contained in this manual are shown with the standard danger symbol shown below:

The following safety instructions are used. They provide extremely important information. Read this information carefully.

WARNING!

Indicates a possible medium-risk danger which could lead to serious or even fatal injuries if not avoided.

CAUTION!

Indicates a low-risk danger which could result in minor or less serious injury or damage if not avoided.

NOTICE

Indicates behavior which could result in damage to property.

2.1.1   Demands on the Technical State of the Product

The following special requirements are made to ensure safe operation:

Take all information on environmental conditions into consideration before setup and operation (see the documentation of your computer, hardware, etc.).

> ⚠ **WARNING!**
>
> ***Wrongly initialized NVRAM variables can lead to unpredictable behavior of a vehicle or a test bench, and thus to safety-critical situations.***
>
> *ASCET projects that use the NVRAM possibilities of AUTOSAR expect a **user-defined** initialization that checks whether all NV variables are valid for the current project, both individually and in combination with other NV variables. If this is not the case, all NV variables have to be initialized with their (reasonable) default values.*
>
> *Due to the NVRAM saving concept, this is **absolutely necessary** when projects are used in environments where any harm to people and equipment can happen when unsuitable initialization values are used (e.g. in-vehicle-use or at test benches).*

> ⚠ **CAUTION!**
>
> ***Wrong word size and/or compiler division lead to wrong compilable code. Wrong compilable code may lead to unpredictable behavior of a vehicle or test bench, and thus to safety-critical situations.***
>
> *When working with the EHOOKS target, , users must ensure that word size and compiler division match the selected EHOOKS-DEV backendto avoid wrong compilable code.*
>
> *See also the ASCET-SE V6.2 EHOOKS User's Guide.*

Further safety advice is given in the ASCET V6.2 safety manual (`ASCET Safety Manual.pdf`) available on your installation disk, in the `ETASManuals\ASCET V6.2` folder on your computer or in the download center of the ETAS web site.

2.2 Interpolation Routines

Each ASCET-SE target is supplied with a pre-compiled interpolation routine library.

The interpolation routine library is provided for example only. It is not permitted to use the library in production code or within ECUs running in vehicles. The libraries are signed. Any use of them in a project will give the following warning:

```
WARNING(): Disclaimer for interpolation rou-
tines.txt(1): Invalid interpolation library linked. THE
ETAS GROUP OF COMPANIES AND THEIR REPRESENTATIVES,
AGENTS AND AFFILIATED COMPANIES SHALL NOT BE LIABLE FOR
ANY DAMAGE OR INJURY CAUSED BY USE OF THIS ROUTINES
```

ASCET-SE is also supplied with the source code and scripts required to re-build the library. By re-building the library you take full responsibility for ensuring the correctness of the source code, the build process and the interpolation routines in the library.

> **Note**
>
> *The ETAS group of companies and their representatives, agents and affiliated companies shall not be liable for any damage or injury caused by use of these routines.*

2.3   FPU Usage

ASCET-SE supports floating point code generation. This is especially advantageous for microcontrollers with an on-chip floating point unit (FPU).

However, if an application does not use floating-point, run time and stack consumption can be saved by not saving and restoring the FPU's floating point registers over task context switches. RTA-OSEK provides this type of optimization and ASCET-SE will automatically enable the optimization in the OS configuration if all processes and methods in a task do not use the FPU.

The information about whether or not a process or method uses the FPU is provided by a flag in the implementation information. By default, this flag is enabled, indicating the FPU is used. If the process or method does not use the FPU then the flag can be disabled.

It is the user's responsibility to ensure the FPU flag is only disabled when they are certain that no floating-point code is used in the process or method.

If the flag is disabled and the process or method uses the FPU then the floating-point context will not be saved and may be corrupted over a context switch, resulting in unpredictable application behavior.

If in doubt, leave the FPU flag enabled.

2.4   Non-Volatile Elements

ASCET-SE supports the handling of different memory classes, as described in chapter 5.3 "The `memorySections.xml` File". Each memory area can either be volatile or non-volatile. For this reason, ASCET-SE checks the uniform usage of each memory class either for volatile elements or for non-volatile elements. If both properties are mixed within one memory class, an error message is generated.

Non-volatile variables are intended to remain in the ECU memory persistently, also after a re-boot of the ECU. For this reason, variables specified as non-volatile are not initialized, even if an initialization value can be entered in the respective data editor.

It is the user's responsibility to care for a correct explicit initialization of non-volatile variables as a part of the function specification.

2.5   Provision of Customized Data Types

If customized data types are used then it is important to ensure that the types declared in `a_user_def.h` are sufficiently wide to hold values of the associated ASCET data type. For example, a customized data type which replaces sint8 must be wide enough to hold the value range `-128..127`.

ASCET cannot check for correct customized data type width, so it is essential that declarations are checked during other stages of the development process (for example by code review).

3   Getting Started

ASCET-SE is a tool for generating software for embedded microcontrollers from an ASCET-MD model. ASCET-SE uses the project to hold configuration inform-stion.

Each ASCET project includes target-neutral code generation settings, an integration of ASCET modules and configuration settings for one or more targets as shown below:
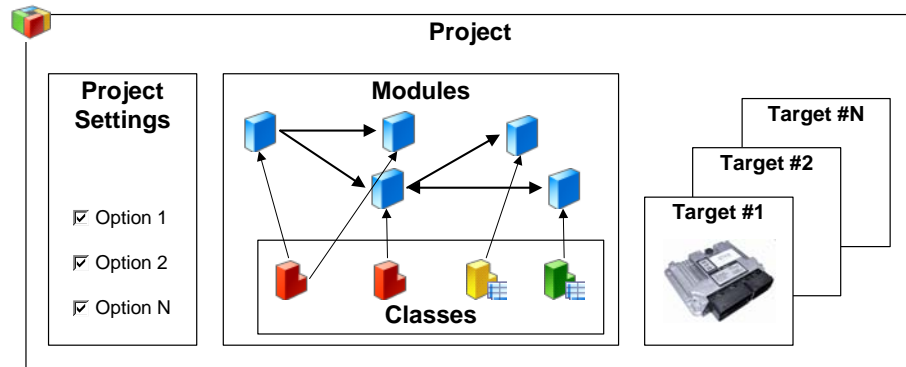
Fig. 3-1   ASCET project

The ASCET online help provides more information about how to create ASCET projects.

To generate code using ASCET-SE you will need to configure a target. In ASCET-SE a target is a specific combination of a microcontroller, a computing platform and a compiler.

Code generation produces C source code files that implement your ASCET project and also produces configuration files for an underlying operating system (OS) or run-time environment (RTE) that capture the real-time requirements of the model, such as sampling rates and communication between models. These configuration files define what ASCET requires from the OS or RTE.

ASCET-SE supports code generation for:

1. OSEK Operating Systems (OSEK OS).
2. AUTOSAR Run-Time Environments (AUTOSAR RTE)

ASCET-SE provides dedicated OSEK OS support for ETAS' RTA-OSEK, however, code can be generated for use with any OSEK operating system and optionally for any OS with a similar scheduling model to OSEK OS.

3.1   Components of ASCET-SE

The ASCET-SE delivery includes:

• The ASCET-SE code generator tools.

• A set of configuration files for each supported target.

• A hex file reader.

These components have the following functions:

- The ASCET-SE code generator tools extend the ASCET system with target neutral C code generation, OS/RTE configuration file generation and optional invocation of the compiler toolchain to build the ECU executable. All targets use the same core code generator.

  > **Note**
  >
  > *The modeling capabilities of ASCET are **not** included in the ASCET-SE shipment. They are subject to separate orders.*

- The configuration files hold all the target-specific information needed by the ASCET-SE code generator to produce code for a particular embedded microcontroller that interfaces with a specific OS/RTE. In addition, the configuration files contain information on how to build the complete system with a supported compiler to produce an executable to run on an ECU.

  > **Note**
  >
  > *The RTA-OSEK operating system configuration tools and target plug-ins are **not** included in the ASCET-SE shipment.*
  > *Please contact your local ETAS sales office for a quotation*

  > **Note**
  >
  > *Target compilers and linkers are **not** included in the ASCET-SE shipment. They are subject to separate orders from the compiler vendor. The **release notes** included in the ASCET-SE installation describe the compiler and linker versions that are supported.*

- The *Hex file reader* extracts address information from the executable so that ASCET-SE can generate an ASAM-MCD-2MC file for measurement and calibration.

  > **Note**
  >
  > *This applies **only** to the addresses of elements declared as ASCET elements.*

3.2 Basic Stages from Model to Executable

The main stages in ASCET-SE code generation are:

1. Generation of C code by the code generator
2. Invocation of the compiler toolchain to compile and link the code to create an executable ready for the ECU
3. Generation of an ASAM-MCD-2MC file for measurement and calibration

The following figure shows these stages in outline:
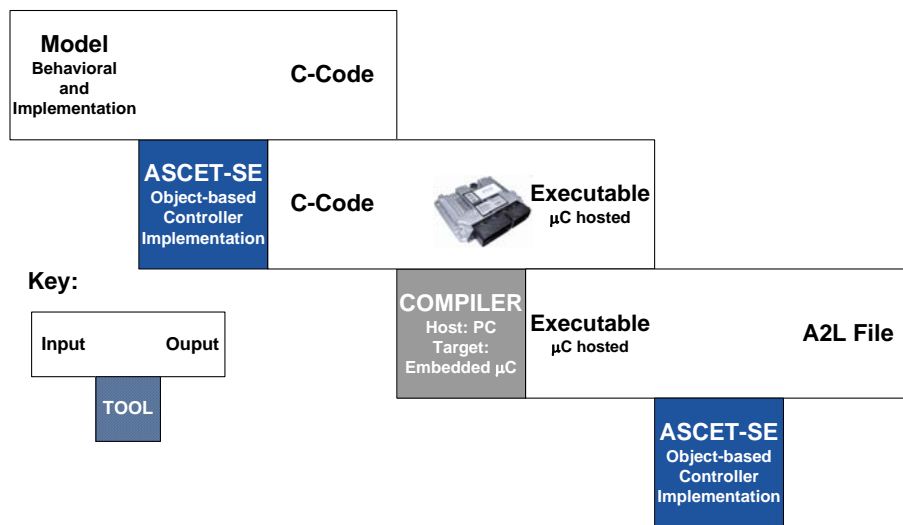
Fig. 3-2   Main stages of ASCET-SE code generation

A more detailed view of what happens is shown in Fig. 3-3.The next three sections explain what happens in each stage
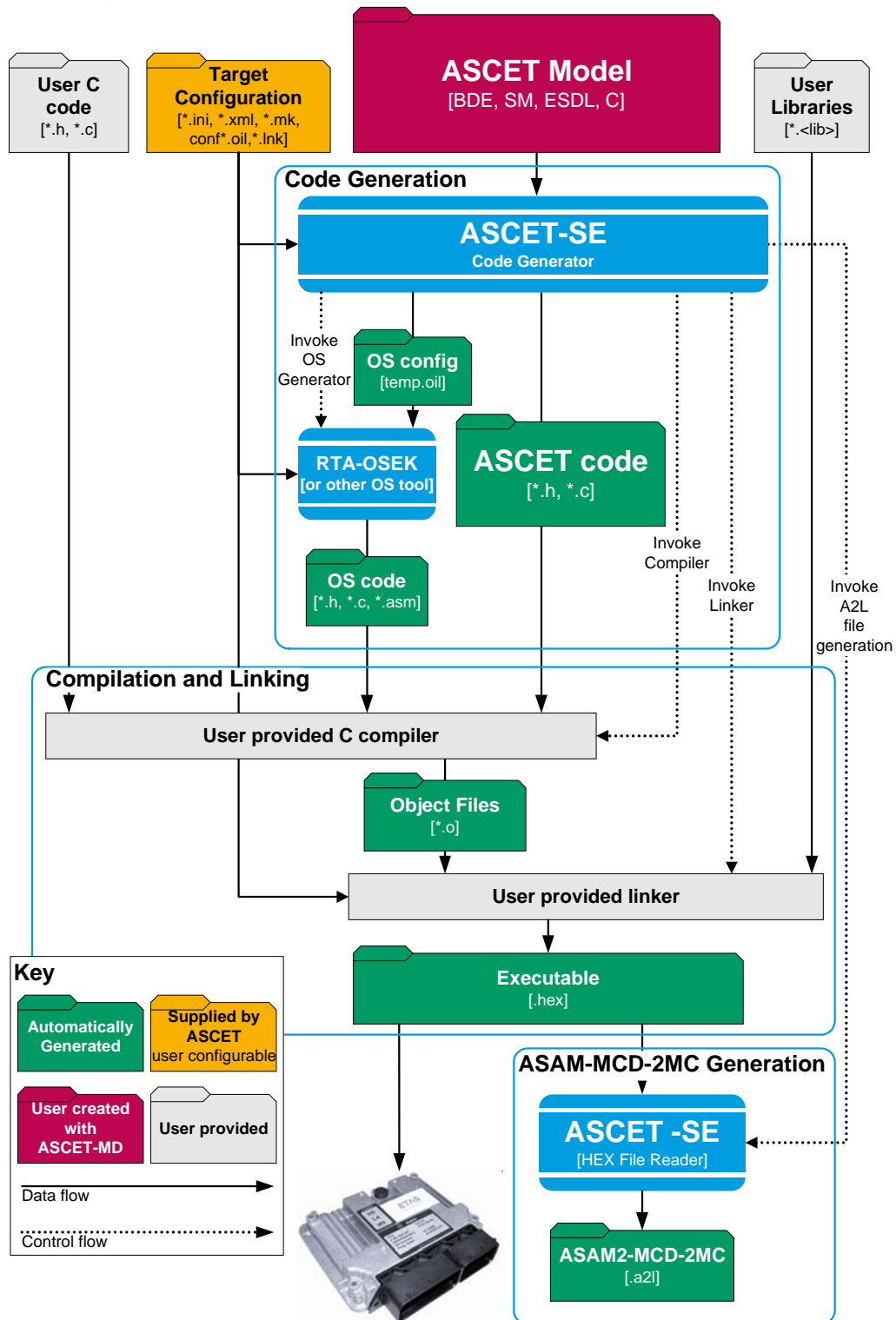
Fig. 3-3   Basic stages in ASCET-SE code generation

3.2.1 Code Generation

The main function of ASCET-SE is the conversion of the ASCET model into C code. Code generation in ASCET-SE always uses a complete model, i.e. a project in ASCET, for the chosen target. C source code files are generated for

- the project itself,
- each module,
- each class,
- each OS task body.

The software architecture, or mapping of model structures into code, is identical-for all ASCET-SE targets. However, the code generator uses target-specific information provided by target configuration files to optimize code generation or customize the code where necessary. For example, the target configuration files can be used to tell ASCET-SE to generate compiler-specific pragmas to place code or data into specific memory sections, whether the hardware provides bit-addressable memory that can be used to optimize bit-fields for space etc.

ASCET-SE also generates an OS configuration file that defines all the OS objects required by the ASCET configuration and then runs the OS generator tools to generate the data structures required by the operating system.

The combination of the ACSET and OS code includes all variable and data definitions required to make the ASCET system work.

Code generated in this way will need to be built to produce a final executable. ASCET-SE supports two use cases for this process:

1. *additional programmer*, where the generated C code is exported to external files and can be used in an external (to ASCET) build process.

2. *integration platform*, where ASCET-SE uses your compiler toolchain to build the executable. This is described in the next section.

More detailed information about how the ASCET-SE code generator works can be found in Chapter 14.

3.2.2 Compilation and Linking

In the *integration platform* use case the target toolchain, comprising compiler, linker and locator, is driven from ASCET, so that the complete project can be built in a similar way to developing software with an Integrated Development Environment (IDE). The integration platform capabilities of ASCET-SE allow you to include non-ASCET C source code and/or libraries in the build process.

ASCET uses a "make"-based system to control the build process, but interaction is similar to the build for experimental targets: on selecting a menu option, the build is started, and when it completes without error, a complete executable program for the project that can be flashed to the ECU.

3.2.3 ASAM-MCD-2MC Generation

At the end of the build process, ASCET-SE uses the hex file reader to extract the addresses of all variables and parameters declared in the ASCET model from the generated hex file.

An ASAM-MCD-2MC description (commonly called an *A2L file*) can be generated, using a separate menu item, to supply information about the system to calibration systems like ETAS' INCA.

3.3   Configuring ASCET-SE for Code Generation

The following sections explain how to configure ASCET-SE for target code generation.

3.3.1   Target Selection

During installation, the user chooses the target(s) to install. ASCET-SE can generate code for any installed target.

Each target is installed in a directory named by the target microcontroller family `<install_dir>\target\trg_<targetname>`, for example:

```
<install_dir>\target\trg_c16x

<install_dir>\target\trg_mpc55xx
```

A special microcontroller independent target, called the ANSI-C target, is also provided that generates portable ANSI-C code. This is installed in:

```
<install_dir>\target\trg_ansi
```

Unlike embedded targets, the generated code does not include any compiler-specific intrinsics for memory mapping and data access on segmented or paged hardware architectures.

ANSI-C code can be used as a basis for supporting targets not supported by ASCET-SE.

In some cases, the supplied target will need to be customized for your specific microcontroller and/or operating system. Please observe the hints provided in this manual at the appropriate places. You are referred to the following sections in particular:

- section 3.3.5 "Memory Class Configuration"
- section 5.2 "The target.ini File"
- section 5.3 "The memorySections.xml File"
- section 7.6 "Interfacing with an Unknown Operating System"

3.3.2   Path Settings for External Tools

ASCET needs to know where the compiler and OS tool chains are installed before it can use them to build ASCET applications. The paths for compiler and operating system must therefore be set in ASCET. If these tools have been installed before ASCET, then the ASCET installation process may be able to find them if they have been installed on the same host PC.

Note

It is recommended that automatically identified toolchain paths are checked for correctness before building an ASCET project. In particular, check that the versions of the tools are compatible with the versions expected by ASCET.

To set Compiler and OS toolchain paths:

- In the ASCET Component Manager, select **Tools →
  Options**.

  The "Options" dialog window opens.
- Go to the "External Tools\Compiler" node.

- Go to the subnode of your compiler, e.g., "Tasking Vx V2.x for C16x".
- Click on the button next to the "Tool Root Path" field.
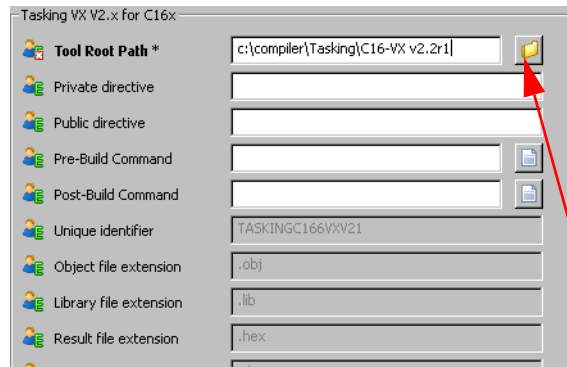
- In the "Path Selection" window, select the path for the compiler/linker and close the window.
- In the "Options" dialog window, go to the "Operating System" node.
- Go to the subnode of the OS you want to use and select the OS Installation Path.
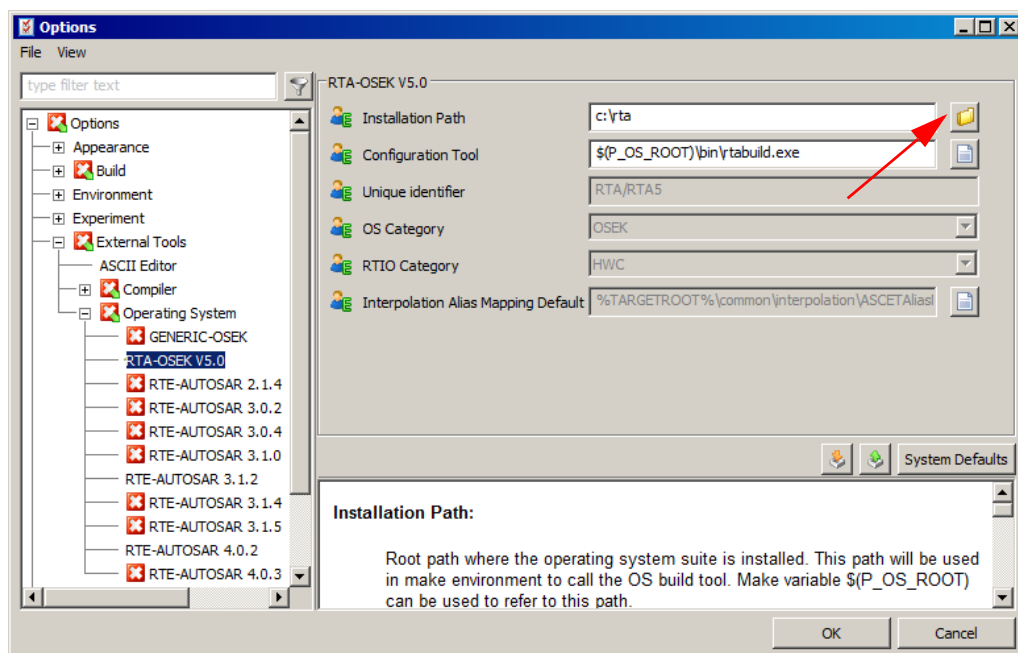
- Click **OK** to accept the changes.

3.3.3 Code Generation Settings

Code generation settings are specified on a per-project basis in ASCET's Project Editor. The settings control which compiler and OS are used for the build process.

To set the project options:

- In the project editor, click the **Project Properties** button.

  The "Project Properties" window opens in the "Build" node.

- Select the target and the corresponding compiler.

  In the "Code Generator" combo box, the entry `Object Based Controller Implementa-tion` is the only valid choice.

- Select the operating system.

  Some or all of the following operating systems are available:

`RTA-OSEK Vx.y`	Code and configuration data are generated to interface with Version x.y of ETAS' OSEK operating system.
`GENERIC-OSEK`	Code and configuration data are generated for a Generic OSEK. Additional vendor-specific configuration may be required outside of ASCET.
`RTE-AUTOSAR x.y`	Code and configuration data are generated to interface with Version x.y of the AUTOSAR RTE.

Note

The `RTE-AUTOSAR x.y` *operating systems are only available for the ANSI-C target.*

- Set the code generation options in the various sub-nodes.
- Click **OK** to accept the changes.

More details on code generation settings are given in the ASCET online help.

3.3.4   Operating System Configuration

Operating system configuration is used to configure how the OS is integrated with ASCET. OS integration includes mapping processes into tasks, defining task attributes settings, defining interrupt attributes, etc.

Configuration is done in the "OS" tab of the Project Editor (see the ASCET online help for additional details about the Project Editor).

ASCET assumes a priority-based pre-emptive operating system like OSEK OS. It is important to understand how the OS schedules tasks at runtime because this influences how ASCET processes (mapped into tasks) are scheduled. Some basic guidance, including the restrictions which apply to OS integration, is provided in section 7.1 "Scheduling and the Priority Scheme". Code generation errors will be issued if the restrictions mentioned there are not observed.

Note

For the `RTE-AUTOSAR` *"operating system", only ANSI-C code generation is supported and no operating system settings are required. Any settings you make in the "OS" tab for a newly created project that uses* `RTE-AUTOSAR` *are removed together with the "OS" tab itself when you close the project editor.*

3.3.5 Memory Class Configuration

Unlike a PC, embedded microcontrollers usually require that data and code is located in specific sections of memory, often at specific addresses. Program code and static data (e.g. constants) is usually located in ROM. Dynamic data (i.e. variables) must be located in RAM.

Some microcontrollers also allow memory sections that can be addressed in different ways. For example, some sections might be addressable with an 8 or 16-bit address and other sections may only be accessible with a 32-bit address.

The arrangement of elements in the controller memory is determined by the *memory classes* they are assigned to in the implementation. In the ASCET data model, memory classes are represented simply by abstract names, freely selected by the user. Example names might be:

- IRAM - Internal RAM
- IFLASH1 - First bank of internal Flash ROM memory
- IFLASH2 - Second bank of internal Flash ROM memory
- NEAR_RAM - RAM addressable with an 8-bit address
- FAR_ROM - ROM addressable with a 32-bit address

The definition of the names and the conversion to compiler-specific conventions for marking up the C code correctly is stored in a file called `memorySections.xml` in the target directory. ASCET-SE supplies a typical file for each target.

The section names defined in `memorySections.xml` are selectable in the implementation editor for each ASCET element.

During the second phase of code generation, ASCET-SE uses the conversion information in `memorySections.xml` to add the correct compiler intrinsics (usually `#pragma` statements) to the generated C code.

The use of memory classes is described in detail in section 5.3 "The memorySections.xml File".

The assignment of actual memory addresses to these locations is done in the linker control file.

3.3.6 Target Initialization Code

Each ASCET target includes an example application which provides simple target configuration. By default, ASCET-SE uses the target configuration and the main program from this example when building a project. The files used are

```
<install_dir>\target\example\target.[hc]

<install_dir>\target\example\system_counter.c
```

These files contain a main program and the code required to initialize the target hardware to provide a 1ms periodic timer interrupt used to drive task scheduling. The interrupt handler itself is provided in `system_counter.c.` This code must be reviewed for suitability in production projects.

If additional interrupts are defined in ASCET, then additional target code is required to configure the interrupt sources and (possibly) to initialize interrupt priority registers. You should consult your OS documentation for further information.

Note that ASCET assumes that memory sections have been initialized correctly for executing C programs. By default, ASCET uses the C start-up code (the code which executes before the main program is entered) provided by the compiler vendor for initializing the C environment.

3.3.7 Customizations for Compiling and Linking

The following settings are required in the linker/locator control file to customize for a specific hardware target:

- Locate the ASCET memory classes defined in `memorySections.xml` to the applicable physical memory space (see section "Linker/Locator Control" on page 72).

- Locate the memory sections for the operating system into the physical memory space. Note that it may be necessary to tell the OS the location of the stack pointer. For specific instructions, refer to the OS documentation (for RTA-OSEK this information is given in the RTA-OSEK Binding manual for the target).

Compiler and linker invocation can be customized in the `project_settings.mk` make file (see section 5.4.1). For example, special supplementary header files and pre-compiled objects can be integrated via this make file, as well as user-provided libraries (e.g. for drivers, external code, interpolation routines), compiler, assembler and linker options and some settings concerning the build process.

On some targets, additional configuration for time measurements may be required.

- Enter the *input frequency* and *timer prescale factor* in the `project_settings.mk` file (see section 5.4.1).

Modifications are also possible in the `target_settings.mk` configuration make file (see section 5.4.1), which contains compiler-specific configurations. However, changes in this file should be avoided, if possible.

3.3.8 Generating the Executable File and Running it on the Target

Before an application can be executed on the target microcontroller an executable file must be created. If a measurement and calibration tool will be used, then an ASAM-MCD-2MC file also needs to be generated. This section reviews the steps for generating source code, the executable, and the ASAM-MCD-2MC file.

Depending on the target, the following modifications may be necessary:

- Enter the memory layout into the ASAM-MCD-2MC data file `mem_lay.a2l` (see section 8.2).

- Enter global blobs for the ETK (TP and QP blobs) into the ASAM-MCD-2MC data files `aml_template.a2l` and `if_data_template.a2l` (see section 8.3).

The following sections explain each stage.

To generate the source code:

> **Note**
>
> *Code can be generated and simulation for an ASCET module without a project context when using the code generator in physical experiment mode only. Using other modes of the code generator requiire that modules are integrated into a project. A default project can be defined for each class or module for that purpose. This is the only way to access the implementation information. Without project context, the conversion formulas as well as all implementations of imported entities are missing.*

- In the project or component editor, select **Build** → **Generate Code** to generate source code.

  Code can be generated for the entire project or any component (i.e., module or class). All the necessary components are generated automatically.

- Select **File → Export → Generated Code → *** to save the source code to a file.

  Until this step is performed, the code only exists internally within the ASCET code manager.

To generate executable code for the project:

- In the project editor, select **Build** → **Build** to create an executable file.

  Code for the complete project is generated, compiled, and linked. If no errors occur, an executable file in hexadec. format, named `temp.*`, is created. The source and object code created during the code generation is stored in the ASCET database.

When generating an executable file, all files (including the source code) are created by default in the `<install_dir>\CGen` directory. When the **Keep files in Code Generation Directory** option in the "Build" node of the ASCET options is deactivated (see the ASCET online help), the content of the `<install_dir>\CGen` directory is deleted whenever you exit your ASCET session.

> **Note**
>
> *To retain any of these files, they should be copied into another directory before ASCET is closed. Retrospectively activating the option has no effect for the running session.*
> *The files generated in* `<install_dir>\CGen` *are not compilable C source files*.
> *If only the source code needs to be saved, then the code should be exported using* **File → Export → Generated Code → *.** *These menu options prompt the user for a location in which to save the generated code provided the code was previously stored in the database during the code generation process.*

ASCET's "make" mechanism does not take *all* dependencies (e.g., formula changes, etc.) into account for efficiency reasons. Some global side effects from changes in the model are therefore not recognized. After changes in the model structure, a complete regeneration should therefore be enforced via **Build → Touch → Recursive** before the generation of important code is started.

Once the executable is being generated, the ASAM-MCD-2MC data for the interface to the application system needs to be created.

To write the ASAM-MCD-2MC file:

- In the project editor, select **Tools → ASAM-2MC → Write** to generate the ASAM-MCD-2MC file.

  The "Write ASAM-2MC To:" dialog window is displayed.

- In the dialog window, enter the specific file name and select the specific storage directory.

Note

If the ASAM-MCD-2MC file is to be stored, be careful when placing in the directory `.\CGen\`*. The files in this directory may be deleted upon exiting ASCET, depending on the settings in the ASCET options (see the ASCET online help).*

At this point, the user has everything that is needed to run the program on the target. The executable program can be loaded onto the controller or evaluation board, for instance, using a debugger or calibration system. The ASAM-MCD-2MC file is used by the calibration system (e.g., INCA) for calibration and measurement.

Other tools (e.g., logic analyzer, source level debugger) can be used if necessary, based on the user's preference.

Differences for the ANSI-C Target

Linking is suppressed for the ANSI-C target due to undefined behavior for e.g. startup code, memory layout etc. This suppression is controlled by the `noLinking` option in the `target.ini` file; this option contains a list of all compilers for which linking is disabled.

If you use a compiler listed after the `noLinking` option, **Build → Build All** and **Build → Rebuild All** stop after the creation of the `*.obj` files and the following error message is shown in the monitor window:

```
Selected target "ANSI-C" / compiler "<compiler name>"
combination does not support "Link Code" --- please
refer to target description file ("c:\ETAS\ASCETx.y\
Target\trg_ansi\target.ini")
```

For compilers as Microsoft Visual C++ , the calculation of physical addresses is meaningless. To suppress map file generation for these compilers, `target.ini` offers the `noMapFileGeneration` option which contains a list of compilers for which no map files shall be generated.

Similarly, generation of an ASAM-MCD-2MC description needs access to the executable program file. As ANSI-C code generation usually does not produce an executable (because linking does not happen) the generation of an ASAM-MCD-2MC file is not possible.

It is recommended that the code generation option **Generate Map File** (see the "Project Properties" window or the ASCET online help for details) is deactivated in order to avoid the generation of the Virtual Address Table and the `etas.map` file. See also the notes in section 8.4.

The following table show which ASCET-SE features are supported by a default installation for which combinations of target and operating system.

	Target	
Operating System	**Embedded**	**ANSI-C**
RTA-OSEK	Code Generation Compile Link A2L generation	Code Generation Compile
Generic OSEK	Code Generation Compile Link A2L generation	Code Generation Compile
RTE-AUTOSAR	---	Code Generation Compile

3.4   ASCET-SE Installation Reference

This section provides a quick reference to an ASCET-SE target installation directory `<install_dir>\target\trg_<targetname>`.

3.4.1   Installation Contents

Some important ASCET-SE files are listed and shortly described below. They are located in a subdirectory of the ASCET installation, i.e., relative to the `<install_dir>\ETAS\ASCET6.2` directory. The subdirectory is called `.\target\trg_<targetname>`.

Directory `.\target\trg_<targetname>`

File	Meaning / Explanation
`.indent.pro`	Configuration file for the "Indent" code formatting utility.
`aml_template.a2l`	Template file with type descriptions of global configuration BLOBs for the ETK. This file must be customized by the user (see section 8.3 on page 107).
`build.mk`	Makefile for the linker/locator phase (see section 5.4.5).
`clean.mk`	Makefile to customize the **Build → Clean Code Generation Directory** menu option in the project editor.
`codegen.ini`	File with macro definitions for code generation. The individual entries are explained in the file itself.
`codegen_<target-name>.ini`	File with target-specific settings for code generation. The individual entries are explained in the file itself.
`codegen_ecco.ini`	File with ECCO settings for code generation. It is read by ECCO each time code generation for a specific target is started. The entries are explained in the file.
`compile.mk`	Makefile for the compiler phase.
`custom_settings.mk`	Makefile for customizing the Make process.
`depend.mk`	Makefile for generating the dependencies of the generated files.
`do_compile.mk`	Make file for actual compiler invocation.
`generate.mk`	Makefile only for code generation via ECCO. After execution of this makefile, all project modules are generated as C and H files and are written in the directory `.\CGen` of the ASCET installation (see section 5.4.3 "Code Generation – Make File generate.mk").
`global_settings.mk`	ASCET-SE internal makefile.
`if_data_template.a2l`	Template file with type descriptions of global configuration BLOBs for the ETK. This file must be customized by the user (see section 8.3 on page 107).
`mem_lay.a2l`	Example data file defining the memory layout of the controller in ASAM-MCD-2MC format. This file must be customized by the user (see section 8.2 on page 107).

File	Meaning / Explanation
`memorySections.xml`	Contains XML definitions of memory classes. See section 5.3 "The memorySections.xml File" for more information.
	Note that the ANSI-C target (`trg_ansi`) contains additional memory class definitions files `memorySections_Autosar.xml` and `memorySections_Autosar4.xml`.
`OS_<osname>_<version>.template`	OS template file for `<osname>` (and optionaly `<version>`) used by ASCET-SE to generate an OS congfiguration file.
`os_settings.mk`	Makefile for general OS settings.
`postasap.mk`	Makefile for post-processing ASAM-MCD-2MC files.
`prj_def.a2l`	Example ASAM-MCD-2MC file to define the `MOD_PAR` section (see section 8.1).
`project_settings.mk`	Contains project-specific configuration settings like included libraries or special compiler and linker settings (see section 5.4.1).
`services.ini`	File containing arithmetic services (see the "Arithmetic Services" section in the ASCET online help).
`settings_<compiler>.mk`	Defines compiler- and target-specific settings valid for all projects, such as file extensions, call conventions for precompiler, compiler, linker and other programs, as well as paths for program calls, include files and libraries (see section 5.4.4).
`smart_compile.mk`	Makefile for SmartCompile control.
`target.ini`	Target-specific settings for ASCET for the default variant of the target microcontroller; the individual entries are described in more detail in section 5.2.
`target_<variant>.ini`	Target-specific settings for ASCET for alternative variants of the target microcontroller; the individual entries are described in more detail in section 5.2.
`target_settings.mk`	Makefile to specify target specific settings (see section 5.4.1).

Directory `.\target\trg_<targetname>\cp_rules`

This subdirectory contains the Perl macros, know as the *Code Production Rules*, that are used by ECCO during C code generation.

Directory `.\target\trg_<targetname>\docco`

This subdirectory contains the stylesheets and definitions files used in by the DOCCO automatic code documemtation tool.

Directory `.\target\trg_<targetname>\example`

This directory contains files with target-specific settings for a small ASCET-SE example project.

File	Meaning / Explanation
`confV50.oil`	A template OIL file, which is the entry point for the example project. This file contains definitions of OIL objects like `CPU`, `OS`, `COUNTER` (system counter, for the time raster), an `ISR` (which drives the system counter) and `COM`.
`example_rta.exp`	ASCET export file containing the example project.
`HowTo.html`	HTML file that describes the further content of this directory and explains what the exampel application does and how to build it in ASCET.
`<targetname>_user .<lnk>`	Example linker/locator control file; see also section "Linker/Locator Control" on page 72. The `<lnk>` extension depends on the target.

Directory `.\target\trg_<targetname>\include`

This directory contains the C include files for ASCET-SE.

File	Meaning / Explanation
`a_basdef.h`	Central header file with ASCET controller definitions; the file is to be included by all ASCET projects files.
`a_limits.h`	Definitions of the upper and lower boundaries for standard ASCET types.
`a_sect.h`	Header file with memory section definitions. Not required for all targets.
`a_std_type.h`	Contains definitions of ASCET standard types, e.g., `uint16`.
`a_user_def.h`	Used to define customized data types. By default, this file contains no compilable code.
`message_scheme.h`	Header file for the selection of the message variant (for more information, see section 13.4.3 "Messages").
`os_inface.h`	Header file containing OS interface definitions; the file is included by all generated component C files.
`os_rta_inface.h`	Header file containing OS interface adaptations for RTA-OSEK.
`os_unknown_inface.h`	Template header file containing OS interface adaptations that allows customization to an OSEK-like OS.
`proj_def.h`	Header file for application-specific adaptations (see section 5.7.2 "The Include File proj_def.h").
`tipdep.h`	Header file for target-specific declarations.

Directory `.\target\trg_<targetname>\Intpol`

> **Note**
>
> *The interpolation routines provided with ASCET are examples, not intended to be used in production or in ECUs running in a vehicle. See also the safety hints in section 2.2.*

File	Meaning / Explanation
`a_intpol.h`	interface definitions of the interpolation routines
`build_cmd.bat`	Batch file used during the build process of the interpolation library. *This file **must not** be called directly. It is to be called only by* `intpol_<target>_<compiler>.bat` *files.*
`customize.pm`	Perl macro with functions that can be customized to generate desired type combinations for interpolation routines.
`HowTo.html`	Instructions on handling of interpolation routines.
`intpol_<target>_` `<compiler>.bat`	Batch file to start the build process for an interpolation library for the target `<target>` and the compiler `<compiler>`. The source files have to be located in the `.\target\trg_<targetname>\` `as\intpol\src` subdirectory.
`makeintpol.pl`	Perl script to generate the type combinations of interpolation routines.
`makeintpol_` `header.pl`	Perl script to generate a header file with prototypes of interpolation routines, used by ASCET-SE for characteristic tables.
`path_settings.bat`	Batch file to set compiler paths for all targets. Called by `intpol_<target>_<compiler>.bat`.
`settings_` `<compiler>.mk`	Make file for compiler-specific settings.

Directory `.\target\trg_<targetname>\Intpol\lib`

File	Meaning / Explanation
`Disclaimer for interpolation routines.txt`	Important information regarding the provided interpolation routines. Read carefully!
`intpol_<target>_ <compiler>.<lib>`	Library of interpolation routines, which is linked to the project in `project_settings.mk` (included in `build.mk`, see section 5.4.5). The library does not contain all possible interpolation routines. Further routines can be generated automatically on demand via the `customized.pm` file. The extension `<lib>` is the target-specific extension for libraries defined by the target compiler. Typical examples are `*.lib`, `*.h12`, `*.a`

For further details see chapter 6 "Interpolation Routines"; if in doubt, please contact ETAS.

Directory `.\target\trg_<targetname>\Intpol\Src`

This directory contains all source code templates for interpolation routines.

Directory `.\target\trg_<targetname>\scripts`

This directory contains several Perl scripts. The table lists the most important ones.

File	Meaning / Explanation
`convert_hip_db.bat`	Batch file for migration of memory class definitions from the old format (`hip.db/target.ini`) to the current format (`memoryScections.xml`).
`convert_hip_db.pl`	Perl script used by `convert_hip_db.bat`.
`cctolog.pl`	Perl script that transforms error/warning messages generated by a compiler into a format readable by ASCET. Thus, errors/warnings can be automatically displayed in the ASCET monitor window.
`lltolog.pl`	Perl script that transforms error/warning messages generated by a linker into a format readable by ASCET. Thus, errors/warnings can be automatically displayed in the ASCET monitor window.
`ostolog.pl`	Perl script that transforms error/warning messages generated by an OS configuration tool (like `rta-build.exe`) into a format readable by ASCET. Thus, errors/warnings can be automatically displayed in the ASCET monitor window.

Directory `.\target\trg_<targetname>\source`

File	**Meaning / Explanation**
`blkcopy.c`	Block Copy routines for initializing the arrays in the controller code.
`msgcopy.c`	Contains methods for copying non-atomic messages (i.e., messages larger than one machine word).
`upmsgcp.c`	*unprotected message copy* - used to allow communication between two processes via messages.

4   Implementation Configuration

When modelling with ASCET, the physical model's functional behavior can be tested. Then, the embedded control software can be refined gradually up to the production stage of development. This is done by specifying the implementation information in conjunction with the code generation.

The task of the implementation consists of mapping the physical model, represented by continuous, discrete and logical entities, to the implementation layer in a semantically correct way. A major part of this task is to decide how to map continuous real arithmetic of the model into the discrete integer (fixed-point) arithmetic supported by embedded target microcontrollers. The transformation requires a quantized representation of all entities. Quantization introduces numerical error that cannot be avoided. The behavior of the generated code will always differ slightly from the physical specification.

> **Note**
>
> *In ASCET, "Implementation code generators" serves as a generic term for the code generators used for the "implementation experiment" and "controller implementation" (or "object-based controller implementation", respectively). They resemble each other closely in terms of structure and mode of operation.*

In the context of the user's specifications, the implementation code generators create a compromise between numerical precision, RAM and stack requirement, code size, and code performance.

Implementations are a refinement (the addition of detail) of the physical model and are necessary to create embedded control software in ASCET. They determine how the physical functionality is mapped to an implementation in an ECU. The separation of the physical model and its corresponding implementation in ASCET helps to support a structured development process.

4.1   Implementations for Basic Model Types

To edit an element implementation:

- Right-click the element you want to implement, e.g. the parameter `P_Gain` in the following example.

- Choose **Edit Implementation**.

The implementation editor shown below opens.
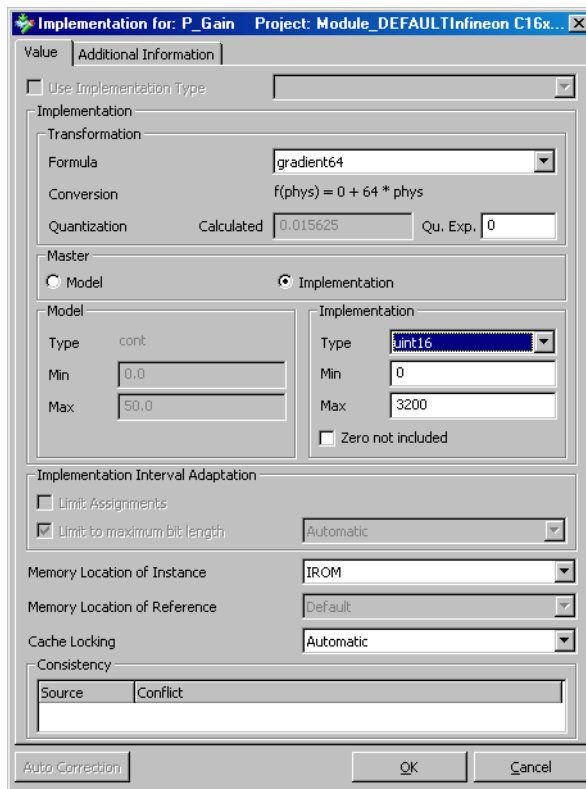
In this example, `P_Gain` is the proportional gain for a PID controller. It has a physical range of 0.0 to 50.0 and a quantization of 0.015625, i.e.

```
Ximpl = 0 + 64*xphys
```

The implementation of the variable has type `uint16` with a range of 0 to 3200. The following table shows how physical values are mapped onto implementation values:

xphys	Ximpl	
	Integer	**Binary**
0.000000	0	00000000_00000000
0.015625	1	00000000_00000001
0.031250	2	00000000_00000010
...	...	...
0.984375	63	00000000_00111111
1.000000	64	00000000_01000000
1.015625	65	00000000_01000001

xphys		Ximpl
	Integer	**Binary**
...	...	...
49.968750	3198	00001100_01111110
49.984375	3199	00001100_01111111
50.000000	3200	00001100_10000000

Since this is a calibration parameter (the parameters are typically located in a ROM memory area), the memory class IROM is selected.

The following sections describe the various aspects of element implementation.

4.1.1   Implementation Data Types

Unlike the abstract data types used for quantities in the physical model (i.e., continuous, discrete, logical), a concrete data type is used in the implementation. ASCET uses the following implementation data types:

Type	Contents	Comment
sint8	8-bit signed integer	-128 to +127
uint8	8-bit unsigned integer	0 to +255
sint16	16-bit signed integer	-32768 to +32767
uint16	16-bit unsigned integer	0 to +65536
sint32	32-bit signed integer	-2147483648 to +2147483647
uint32	32-bit unsigned integer	0 to +4294967296
real32	32-bit IEEE Floating-Point	not available for all targets
real64	64-bit IEEE Floating-Point	not available for all targets
bit	directly addressable single bit	not available for all targets

> **Note**
>
> *On certain processors, the floating-point implementation is only possible with software libraries that are capable of emulating floating-point arithmetic. In such cases, it is not recommended for typical applications in electronic control units because it requires considerable more execution time and memory.*

The following special cases apply:

- When a variable of model data type udisc is mapped to an implementation data type of sint*, the lower limit of the implementation interval is *not* set to the corresponding negative value, but to zero.

- When a variable of model data type sdisc is mapped to an implementation data type uint*, the upper limit of the model interval is *not* set to 2147483647, but to the maximum value of the implementation data type. This is valid even for the uint32 implementation data type.

- When you edit a variable of model data type cont or sdisc and implementation data type uint*, the lower limit of the model interval is *not* set to the corresponding negative value, but to zero.

The code generation allows a combination of floating-point and integer arithmetic in the software for assignment only:

- The assignment of non-quantized floating-point to quantized integer quantities and *vice versa* is valid.
- The code generator creates the necessary code for the conversion and automatic limits.
- The same holds true regarding method calls for the implicit mapping between formal and actual arguments.

Note

The combination of floating-point and integer implementations in mathematical operations or comparisons is invalid and results in an error message.

4.1.2   Conversion Formula

A conversion formula transforms the physical value of a model quantity into its implementation value in the software. This transformation must be invertible in the valid interval (i.e. value range) for the quantity. In ASCET, the conversion formula is always specified from physical model to implementation, i.e.

```
Ximpl = f(xphys)
```

Conversion formulas are required:

- for physical quantities of type *cont* that are to be mapped to *integer* in the generated code.

The identity conversion formula (`Ximpl = xphys`) must be used in the following cases:

- for logical (Boolean) quantities, there is no possibility to specify conversion formulas.
- for discrete physical quantities, those of type *udisc* or *sdisc*, the *identity* conversion formula is mandatory.
- for physical quantities of type *cont* with floating-point implementation, the *identity* conversion formula is mandatory.

In the following discussion, physical quantities are generally represented in lower-case characters. The corresponding implementation values are written in upper-case characters.

Conversion formulas can be defined globally for an entire project in the "Formulas" tab of the Project Editor. There, choose **Global Formulas → Add** in order to define a new formula. Afterwards, you can use the defined conversion formulas in the implementation editors.

ASCET knows different types of conversion formulas (i.e., linear, linear rational, square rational, tabular and verbal formulas). However, the code generation supports only simple linear formulas of the following form:

```
X = ax+b
```

Here, `a` and `b` are called the scale value and offset, respectively. The quantization of a value is the reciprocal of the scale value:

```
q = 1/a
```

In the following, it is assumed that scale values and offsets are rational numbers. This is not a substantial restriction because real values can be approximated with a given level of precision using rational numbers. Note also that only rational numbers can be used in for integer arithmetic anyway.

Non-linear conversion formulas can be used in the specification. However, an automatic conversion between non-linear formulas in the code generation is not supported.

> **Note**
>
> *The code generation treats non-linear conversion formulas internally like identity so that no automatic conversions are performed.*

Arithmetic with non-linear quantizations is not possible. They can only be used for inputs of characteristics and methods, e.g., as a time constant of an integrator. The user is responsible for ensuring that non-linearly quantized quantities are used only in such a way. There is no further tool support of this, including the code generation.

4.1.3 Value Range (Only for Numerical Quantities)

The range of values for a quantity is simply its valid numerical interval. The specified value ranges are then used by the code generator to calculate the intervals of intermediate results. In doing so, the occurrence of overflows can be detected. The code generator decides through this how to generate intermediate results and calculations in the software. If necessary, the use of limiters must be enabled.

Both the physical and implementation value ranges can be specified. Then, the linear, invertible conversion formula updates the other value range. Therefore, the user can choose which environment (physical or implementation environment) to work in.

In the following cases, however, the specification of a value range is not possible or will be ignored:

- For logical (Boolean) quantities and enumerations, there is no possibility to specify a value range.
- Continuous physical quantities with floating-point implementation are mapped without limits to the specified implementation data type. Though you can enter a value range in the ASCET editors, it will be ignored. A pseudo-infinite interval is used instead.

4.1.4 Implementation Master

Either the physical model specification or the implementation specification can be chosen as implementation master. The values entered by the user for the implementation master will be used to adapt the opposite, non-master side according to the master specification and the formula.

After the global change of a formula in the project editor, all affected implementations can be updated automatically by means of the **Extras → Update Implementations** option in the project editor. In this context, the "Master" options in the implementation editor can be used to specify whether to preserve the value range on the model side or the implementation side. If the model side is selected

as the master, the settings of the model side will remain unchanged and the implementation side will be updated. If the implementation side is the master, the model side will be updated.

4.1.5   Implementation Types

To be able to edit the implementations of individual variables more easily and to be able to easily assign the same implementations to elements with comparable physical significance, you can define what are referred to as *implementation types* in the project context. This is also true of the default project of a class or a module. These implementation types contain the implementation parts described in chapters 4.1.1 to 4.1.4; they can be assigned to individual elements in their implementation editors.

How to create and set up implementation types is described in the ASCET online help, section "Implementation Types". How these are used during implementation is described in the instruction "Using Implementation Types" of the ASCET online help.

4.1.6   Value Range Limitation

The **Limit Assignments** option can be used to specify for each element individually if its value range shall be limited to the defined range. Calculated values which are less than the lowest permitted value are set to the lowest value. Similarly, calculated values that are higher than the highest permitted value are set to the highest value. This is called saturated arithmetic – the highest (lowest) value in the type range is "saturated" with all higher (lower) values. Saturated arithmetic prevents underflow and overflow at runtime.

If the option is activated, additional code is generated for each assignment operation to check and ensure that the specified range is kept. If the option is deactivated, it is the user's responsibility to keep the value range. Continuos physical quantities with floating-point implementation are generated with the selected implementation data type and without limitation.

> **Note**
>
> *In previous ASCET versions, the "Integer Arithmetic" node of the Project Properties dialog window contained an option **Generate Limiters**, which had to be activated for the element-specific limiter configuration to become active. In ASCET V6.2, this option is always* `true`*. It can no longer be edited and has been removed from the Project Properties dialog window.*

By means of the option **Limit to maximum bit length** the user can specify individually for each element, whether and how ASCET checks and avoids potential overflows during assignments. In addition, the user can define the way by which overflow is avoided.

- *Reduce Resolution*: potential overflows are avoided by a suitable re-quantization. This results in a loss of precision.

- *Keep Resolution*: potential overflows are avoided by means of limitation. The resolution remains unchanged. This option can only be used in connection with arithmetic services.

- *Automatic*: ASCET treats potential overflows according to the option *Keep Resolution* if the usage of arithmetic services is active, and according to the option *Reduce Resolution* otherwise.

4.1.7   Zero Containedness in the Value Range

Division in ASCET can be protected against division by zero. This option intro-duces a run-time check in the generated code to ensure that such a division does not occur.

However, for a given element in the model, if zero is not in the range of possible values then the option *Zero not included* can be used to disable the check for division by zero. This options is an assertion to the code generator that the user himself will take care that the denominator does not take the value zero.

> **Note**
>
> *Activate the option **Zero not included** only if you are completely sure that the implemented value can never take the value 0. Otherwise, severe exception errors can occur at ECU run time as a consequence of divisions by zero.*

4.1.8   Memory Locations

Memory locations (selected in the "Memory Location of *" combo boxes) specify the name of the abstract memory section where a quantity (and its reference where applicable) is placed in the memory of the ECU. The code generator uses this information to generate C code data structures according to the required layout of elements in the control unit memory. Besides, the memory classes are used for the generation of corresponding compiler intrinsics, typically `#pragma` statements. The locator uses these `#pragma` statements to map the memory classes to certain address ranges in the control unit. This is done with the help of a transformation table specified by the user.

The code generation checks whether all elements in a certain memory class have the same attribute (*volatile* or *non-volatile*) assigned in the "Memory" field of element editor or not. In the latter case, an error message is generated because one memory class cannot refer to both volatile and non-volatile memory at the same time.

Depending on the "Memory" attribute, variables are treated differently by the code generation: only volatile elements are automatically initialized.

For databases, ASCET provides an easy way to get rid of the error message: the Component Manager menu functions **Tools → Database → Convert → Vari-ables to Volatile** and **Tools → Database → Convert → Parameters to Non-volatile**. The former function assigns the attribute *volatile* to all variables in the database, while the latter assigns the attribute *non-volatile* to all parameters.

For workspaces, there are no such global conversion functions.

4.1.9   Consistency Check

If the implementation editor contains inconsistent data, ASCET will notify the user by means of the *Consistency* check list in the implementation editor. The user can highlight single inconsistencies in the list and correct them automatically means of the **Auto Correction** button, if desired.

4.1.10   Additional Information

Further implementation information can be entered in the "Additional Informa-tion" tab, if required. This can be necessary for a specific electronic control unit. They can also be used for supporting special infrastructures (e.g., DAMOS and MSRDOC). Depending on the application, this field may contain the following:

- Code syntax, address scheme
- Bit base address and binary position for bit packets

This field is not used in the ASCET basic system. Its syntax and semantics are not defined here. The field definition is application-specific. Through the open interface it is possible to add further implementation information.

4.1.11   Sizes of Composite Model Types

The size of composite model types, i.e. arrays, matrices, distributions, characteristic curves and maps, are not part of the implementation specification. Instead, this information is part of the data sets in ASCET.

4.1.12   Summary of Element Implementation

The table below summarizes the implementation information required for each basic model type used in ASCET. Note that only logicals (`log` type) and enumerations do not require all of the implementation information, e.g., no conversion formula. The other scalar types (i.e. `continuous` and `signed/unsigned discrete`) require all of the implementation constituents. This is also true for the array, matrix, and distribution composite types.

> **Note**
>
> *For continuous model types with floating-point implementation, the Identity Conversion Formula (identity, i.e., multiplication with the factor 1.0) is required. For discrete data types, the Identity Conversion Formula is required, too.*
> *In both cases, a warning is displayed when another formula is selected.*

Characteristic lines and maps have special treatment. For these composite types, separate implementation data types, conversion formulas, and value ranges may be specified for the independent and dependent axes. Besides, the access type (linear, rounded, user-defined) can be specified in the properties editor of a characteristic.

	Scalars			Enu-mera-tions	Arrays, Matrices, Distribu-tions	Characteristics	
	logical	dis-crete	cont.			Lines	Maps
Implementation Type	+	+	+		+	2*(x,y)	3*(x,y,z)
Formula		o	+		+	2*(x,y)	3*(x,y,z)
Implementation Data Type	+	+	+		+	2*(x,y)	3*(x,y,z)
Value Range		+	+		+	2*(x,y)	3*(x,y,z)
Data Representation*	+	+			+	+	+

* for parameters only
o *identity* is mandatory
x in the properties editor

	Scalars			**Enu-mera-tions**	**Arrays, Matrices, Distribu-tions**	**Characteristics**	
	logical	**dis-crete**	**cont.**			**Lines**	**Maps**
Memory Location	+	+	+	+	+	+	+
"Additional Information" tab	+	+	+	+	+	+	+
Access Type (linear / rounded / userdef)						x	x

* for parameters only
o *identity* is mandatory
x in the properties editor

4.2   Implementations for Complex Model Types (Classes, Modules, Projects)

The implementation of a complex model type (i.e. class, module or project) involves the following steps:

- Enter the implementations for all the basic model types included in that component.
- Enter the implementations for any other complex model types (i.e., other classes, modules or projects) contained in that component.
- Only if an individual memory class or other component-specific settings (e. g. for the use of user-provided service routines, or for calling hand coded functions) are necessary for the data structures of the component: Acti-vate the respective settings in the "Settings" tab of the implementation editor for components.

The implementation of an entire project defines the implementation of all ele-ments within that project.

In ASCET, it is possible to indicate a number of different implementation alterna-tives for complex model types. For the code generation, however, only one of the indicated alternatives is activated for each instance.

Changing between the alternatives can be done in the implementation editor of the specific element (e.g., on project level). Due to the hierarchic linking of the implementations of a model, the implementations of all child elements are also adapted.

To edit a project or component implementation:

- In the project or component editor, select **Edit →
  Component → Implementation.**

  The implementation editor of the component or
  project opens.
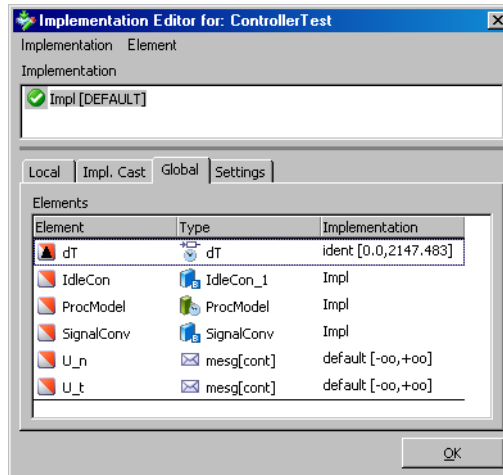
- In the "Elements" pane, double-click on one of the
  elements.

  The implementation editor for that element opens.

This process can be repeated to access the implementation editor for any ele-
ment in the project or component. The above example only allows selecting a
standard implementation. However, it is also possible to define target-specific
implementation alternatives that can be selected.

To copy and paste element implementations:

In the implementation editor of complex model elements, implementations of
basic model elements can be copied and pasted easily.

- In the component/project implementation editor,
  right-click on a basic element and select **Copy
  Implementation To Buffer**.

  The complete implementation information of the
  selected element is copied into a buffer.

- Right-click on another basic element and select
  Paste Implementation From Buffer.

  The entire implementation information from the
  buffer is assigned to the selected element.

4.2.1 Optimized Method Calls

For methods defined in classes, ASCET is able to handle multiple instances using
identical code but different data structures (see chapter 13.3.3 "Data Structures
and Initialization for Complex (User-Defined) Objects"). In these cases, a pointer
to the data structure is passed to the generated C function, the so called *self*-
pointer. As an example, a respective method declaration has the form:

```
sint16 PIDT1_IMPL_out (
      const struct PIDT1_IMPL *self,
      sint16 in);
```

For classes using only one data structure (so called single instances), ASCET auto-matically optimizes the method call and the data elements are accessed directly, e. g.

```
sint16 PIDT1_IMPL_out (sint16 in);
```

This optimization is done by default.

If a user intends to call ASCET-generated methods from code created manually, however, it is not desirable to have the self-pointer optimization done by the tool automatically, as the calling conventions for a method may change unexpectedly due to model changes. For this purpose, ASCET offers the possibility to deacti-vate the single method optimization in the "Settings" tab of the class implemen-tation editor.
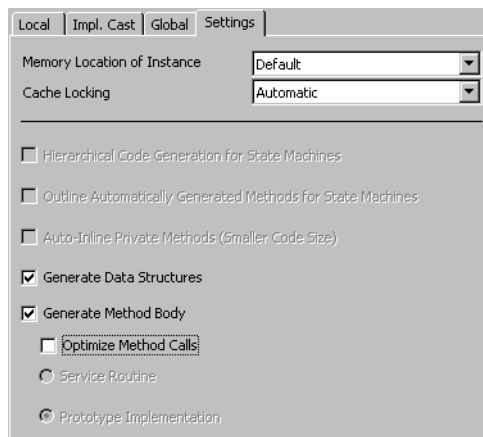
In this case, the self pointer will always be generated, no matter if the class is multiply instantiated or not.

> **Note**
>
> *When calling ASCET-generated methods or using ASCET-generated variable and parameter definitions from handcoded functions, the user must be sure to observe the data type definitions generated by ASCET carefully. It is not recom-mended to use types other than the ones generated by ASCET. This is especially emphasized for the self-pointer.*
> *The function interfaces provided by the ASCET generated code might change in successor versions of the tool.*

If a class will only be single instantiated in a model, a method interface that does not use a self-pointer can be attained by activating the **Optimize method calls** option.

4.2.2   User-Defined Service Routines

The code generator offers the possibility to implement class methods and pro-cesses as user-defined service routines. The method body is then no longer gen-erated by ASCET, but must be provided by the user, for example, by adding the

code during the link process. This makes it possible, e.g., to implement highly optimized methods in assembler code. In particular, service routines have the following properties:

- No method bodies are generated for class methods implemented as service routines. The functionality modeled in ASCET (as block diagram, ESDL or C code) will be ignored for the microcontroller code generation. The user must provide the respective code in other sources. However, ASCET still offers the possibility to specify method contents as they could be needed in simulation experiments executed in ASCET.

- Methods and method arguments specified for service routines can be used from the enclosing ASCET model. However, the generated code provides no "extern"-declarations for them. If a class has local elements, self-pointers will be used and will not be optimized (see section 4.2.1), i.e. for service routines multiple class instances are supported.

> **Note**
>
> *To avoid nested structures as argument types for service routines, it is highly recommended to assign the respective class itself as well as its local variables to the same memory class. In addition, the class using service routines should not contain any local parameters. Parameters should be specified globally, or passed as method arguments, if necessary.*

- Variables exported from the prototype class can be used from the enclosing ASCET model. The generated code provides "extern"-declarations for the prototype methods at the respective locations. The user must provide the respective definitions in his hand coded sources.

- Local instance variables and parameters are generated as a part of the local data structure and passed to the service routine by means of the self-pointer. Imported variables and method local variables are not regarded in the code generated for service routines, as they do not concern the method interfaces.

To specify service routines:

Service routines are specified as follows:

- Select **Edit** → **Component** → **Implementation** to open the implementation editor for a class or module.
- In the "Settings" tab, deactivate the **Generate method body** option.
- Activate the **Service routine** option.

The name of the service routine must follow a strict naming convention: It is comprised of the name of the class or module, the implementation name, and the name of the method or process, each name segment connected with the next by underscores. If the implementation name itself includes underscores (e.g., U8_MASSFLOW_INTEG), it is used in the name of the service routine only up to the first underscore.

> **Note**
>
> *The user must be sure to observe the naming convention.*

For example: Assume a class instance with the name `MassFlow_Integ` of type `INTEGRATORK`. The class contains a specification for a method with the name `compute`. The class was implemented as `U8_MASSFLOW_INTEG`.

The data type prefix of the implementation leads to the call

```
INTEGRATORK_U8_compute(…),
```

i.e., for the name of the implementation, only the data type prefix is taken into the generated function call. Hence there is no need to specify a service routine for every concrete implementation, but only for every data type.

To work with multiple implementations, the following naming convention is recommended (not mandatory): Choose a name after the data type prefix corresponding to the name of the class instance. If necessary, append a consecutive sequence number (e.g., `U8_MASSFLOW_INTEG1`).

These naming conventions can also be met by means of preprocessor commands (`#define`).

Service routines are called from the generated code in the same way as "normal" class methods. This means that the user must observe all conventions regarding arguments, return values, and local elements in the specification of the routine (see chapter 13.3.6 "Method Declarations and Calls").

> **Note**
>
> *When calling handcoded functions or using hand oded variable and parameter definitions from ASCET, the user must be sure to observe the data type definitions generated by ASCET carefully. It is not recommended to use types other than the ones generated by ASCET. This is especially emphasized for the self-pointer.*

The Make mechanism does not generate, compile and link any code for the corresponding class. Instead, the user must provide the respective code (function code, variable and parameter definitions) another way. Within ASCET, service routines can also be defined in the external C code.

4.2.3 Prototype Implementations

Especially for the use of hand coded functions, ASCET and ASCET-SE provide the user the possibility to declare class prototypes. Like function prototypes in the context of a programming language, class prototypes can be used in the ASCET context to declare function interfaces without defining the function contents. In particular, this has the following consequences:

- No method bodies are generated for a class implemented as prototype. The functionality modeled in ASCET (as block diagram, ESDL or C code) will be ignored for microcontroller code generation of prototype classes. The user must provide the respective method code in his hand-coded sources.
  However, ASCET still offers the possibility to specify method contents as they could be needed in simulation experiments executed in ASCET.

- Methods and method arguments specified in the ASCET prototype class can be used from the enclosing ASCET model. The code generated for the surrounding model provides "extern"-declarations of the prototype methods at the calling locations. No self-pointers will be used (see

section 4.2.1), i.e. for prototype classes no multiple instances are sup-ported. The user must provide the respective function definitions in his hand coded sources.

- Variables and parameters exported from the prototype class can be used from the enclosing ASCET model. The code generated for the surrounding model provides "extern"-declarations for the prototype methods at the respective locations. As these declarations are embraced by preprocessor commands, they can be deactivated if required. The user must provide the respective definitions in his hand coded sources.

- Local instance variables, imported variables and method local variables are not regarded in the code generated for a prototype class, as they do not concern the method interfaces. Direct access (whether optimized or not) to local elements of prototype classes is not supported.

To specify method prototypes:

Method prototypes are specified as follows:

- Select **Edit → Component → Implementation** to open the implementation editor for a class.

- In the "Settings" tab, deactivate the **Generate method body** option.

- Activate the **Prototype implementation** option.

The name of the C function must follow a strict naming convention: It is com-prised of the name of the class or module, the implementation name, and the name of the method or process, each name segment connected with the next by underscores. Unlike service routines, no special naming conventions apply for prototypes. The naming conventions can also be met by means of preprocessor commands (`#define`).

> **Note**
>
> *When calling handcoded functions or using handcoded variable and parameter definitions from ASCET, be sure to observe the data type definitions generated by ASCET carefully, especially for element types like arrays, matrices, character-istic tables and maps and classes. It is not recommended to use types other than the ones generated by ASCET.*
> *The function interfaces provided by the ASCET-generated code might change in successor versions of the tool.*

The Make mechanism does not generate, compile and link any code for the cor-responding class. Instead, the user must provide the respective code (function code, variable and parameter definitions) another way (see chapter 9 for possi-bilities).

4.2.4   Processes and Methods

Processes and methods can be implemented as well. Their implementation edi-tors provide the following options:

- The memory location of the process or method code can be defined.

- The usage of the microcontroller's floating point unit (FPU) can be speci-
fied.

  This option is used during OS configuration generation to work out
  whether or not the FPU context needs to be saved during a task context
  switch. If all the processes and methods used in an OS task have this
  option disabled, then the OS does not need to save and restore the FPU
  context as there is no code in the task than can corrupt the current FPU
  context. This optimization reduces execution time and stack RAM con-
  sumption at runtime.

  The default setting is to support FPU usage.

  If the process or method does not use the FPU and this option is enabled,
  then the FPU will not be used for calculation but the FPU context will be
  saved unnecessarily.

  Note

  If the microcontroller does not have an FPU then this option has no effect.

- For methods, the user can define whether function inlining should be
  applied to their code. This option only has an affect if the configuration of
  the compiler defines an appropriate keyword in the "Inline Directive". See
  the entries in the "External Tools\Compiler\<*compiler*>" node of the
  ASCET options dialog for the current settings for your compiler.
- The text entered in the "Symbol" field is the C function name used for the
  process or method in the currently selected implementation of the com-
  ponent.
- The cache locking settings are only relevant for experimental targets.

To open the implementation editor for processes and methods:
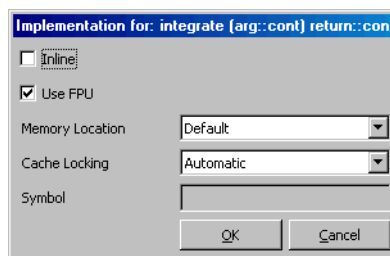
- In the "Outline" tab of the component editor,
  select the process or method.
- Select **Edit → Implementation** to open the imple-
  mentation editor.

4.3   Implementations for Temporary Variables

Temporary variables can be specified at the outputs of operators and complex
model elements. In order to do this, right-click onto the desired element and
choose **Temporary Variable** from the context menu. These temporary variables
cannot be implemented explicitly. Instead, method-local variables can be imple-
mented as described in section 4.5.

For temporary variables, the code generator determines the implementation automatically: when a temporary variable is assigned an implemented quantity for the first time, it obtains the corresponding conversion formula and value range. The implementation data type is chosen so that it is appropriate for the conversion formula and value range.

> **Note**
>
> *The insertion of temporary variable in a mathematical expression does not affect the generation of mathematical operations for this expression. Temporary variables should not be used in different branches of the control flow (e.g., in the branches of an If statement). The result and the implementation (e.g., quantization) may be different for the separate branches. This could cause serious arithmetical errors in the generated code.*

4.4   Implementations for Implementation Casts

Implementation casts (see the ASCET online help) provide the user with the ability to specify the implementation in a targeted manner at any chosen position of a calculation or a data stream. Unlike variables and parameters, implementation casts do not allocate any memory, and thus have no storing effect in the model and cannot be calibrated.

Implementation casts do not have data; they are always of the `cont` model type, always have a scalar dimension and a *local* range of validity (see section 3.3). Unlike other elements, the properties of implementation casts cannot be edited. The implementation of an implementation cast is edited the same way as implementations of basic model types (cf. Chapter 4.1).

4.5   Implementations for Method- and Process-Local Variables

For methods and processes, local variables can be created. For this purpose, double-click on the method or process name in the corresponding class or module editor and then select **Edit** from the context menu. In the "Locals" tab of the signature editor, click **Add** to create a local variable.

After creating these variables, you can provide them with an implementation as described in section "Implementations for Basic Model Types" on page 41. If you do not specify an implementation, the code generator automatically defines the conversion formula, a value range, and an implementation data type in the same way as for temporary variables.

4.6   Migration of Operator Implementations

> **Note**
>
> ASCET V5.0 and later replaced operator with *implementation options* **Limit to maximum bit length** *and* **Zero not included***, and implementation casts to insert requantizations in concatenated arithmetic operations without creating additional storage space requirements.*
> *Existing operator implementations in older projects can be viewed, replaced by implementation casts or removed, but not edited.*

You can delete operator implementations in older models (see the ASCET online help) or replace them automatically by the newly introduced implementation casts. Automatic replacing, however, applies to the entire database and not individual components.

> **Note**
>
> *Implementation Casts are described in sections "Implementation Casts", "Implementation Casts in ESDL" and "Implementation casts in Block Diagrams" in the ASCET online help.*

Rules for automatic conversion: The following conditions have to be fulfilled for an operator implementation to be converted automatically.

- The operator implementation must not contain any other quantization than **Auto** (addition, subtraction, MIN, MAX and MUX).

  > **Note**
  >
  > *This is the only condition which has to be fulfilled for automatic conversion for MIN, MAX and MUX operators. The other conditions only apply to* +, -, *, /.

- The operator output has to be connected.
- The operator output can only be connected to primitive elements.

  It cannot be connected to a component or operator or hierarchy.

- If an implementation cast is connected to the operator output, *something other* than <No implementation> has to be selected for this implementation cast in the combo box next to the **Use Implementation Type** option.
- The operator implementation must not contain any special pre-shift (multiplication and division).
- If the operator is a division operator and the **Allow zero in phys. interval** option is activated in the operator implementation, the following rules also apply for the denominator input:
  - The denominator input has to be connected.
  - The denominator input can only be connected to primitive elements.

    It cannot be connected to a component or operator or hierarchy.

– If an implementation cast is connected to the denominator input, *something other* than `<No implementation>` has to be selected for this implementation cast in the next to the **Use Implementation Type** option.

If one of these conditions is not fulfilled in any implementation of the component (see the ASCET online help), the relevant operator has to be converted manually.

To replace an operator implementation with an implementation cast:

- In the Component Manager, select **Tools → Database → Convert → Operator Implementations to Impl. Casts**.

    The operator implementations of the entire database are converted into implementation casts in accordance with the above rules.

If an operator (apart from MIN, MAX, MUX) can be converted automatically, the following occurs:

- An implementation cast is created on every connection of the operator output.

- If the operator is a division operator and the **Allow zero in phys. interval** option is activated in the operator implementation, an implementation cast is created on the connection to the denominator input.

- The implementation information of the following element is accepted for every implementation cast at the output of an implemented operator. This is not the case for the model type; this is always `cont` for implementation casts.

- The implementation information (apart from the model type) from the previous element is accepted for implementation casts which were added at the denominator input of a division operator.

> **Note**
>
> *For implementations of the component (see the ASCET online help) in which the operator has no implementation, `<No implementation>` is selected for newly created implementation casts.*

- The overflow handling is converted in accordance with the following scheme:

Implementation Cast: / Operator Implementation:	Limit to maximum bit length	Reduce Resolution	Keep Resolution
Reduce Resolution	X	X	
Keep Resolution And Limit	X		X
Keep Resolution And Don't Limit			X

Each row shows the settings set for the implementation cast to replace the corresponding setting of the operator implementation.

- The operator implementation is removed.

If a MIN, MAX or MUX operator can be converted automatically, only the operator implementation is removed. No implementation cast is added.

If an operator cannot be converted automatically, the following occurs:

- An implementation cast is created on every connection of the operator output—even with components, operators etc., `<No implementation>` is selected for these implementation casts in all implementations of the component.
  This implementation cast is given the relevant implementation information during manual conversion of the operator implementation.

  If this kind of implementation cast already exists on one of these connections, no other implementation cast is added to this connection.

- If the **Allow zero in phys. interval** option is activated in the operator implementation of a division operator, an implementation cast with `<No implementation>` is created on the connection of the denominator input.

  If this kind of implementation cast already exists, another one is not added.

- The operator implementation remains unchanged.

If it is not possible to convert all operator implementations automatically in the database, the following message is issued:

```
Not all operator implementations could be replaced
automatically. Please do the conversion manually.
```

Confirm this message with **OK**. The "Operator Implementations" window opens; it shows the components which contain the remaining operator implementations. You can now convert these manually.

5   Configuring ASCET for Code Generation

The properties of generated code are controlled in three different ways in ASCET:

1. Globally for all projects ("Build" node and subnodes in the ASCET options dialog window, opened via **Tools → Options**).

2. For a specific project ("Project Properties" dialog window, opened via **File → Properties** in the Project Editor).

3. For all projects on a specific target by configuring `*.ini`, `*.mk` and `*.xml` files in the corresponding target directory.

The first two ways are described in the ASCET online help. This chapter describes the third way.

Code generation for all projects on a specific target is controlled by three types of configuration file:

1. `codegen[_*].ini` files control the core code generator.

2. `target.ini` provides the target specific information to the Project Editor for OS configuration.

3. `memorySections.xml` defines memory class names for use in the Implementation Editors in ASCET and the mapping between these names and the target-specific compiler intrinsics to provide them.

How code is compiled by ASCET is controlled by a set of GNU makefiles (with the extension `.mk`). The make process is run by ASCET to build a project.

The following sections describe these aspects of configuration file in more detail.

5.1   The `codegen[_*].ini` Files

ASCET uses three files to control the code generator:

- `.\target\trg_<targetname>\codegen.ini`

  Contains macro definitions defining the naming conventions of objects generated by code generator and additional settings for some aspects of code generation. This file is read only by the ASCET base system.

- `.\target\trg_<targetname>\codegen_<target>.ini`

  Contains target-specific settings for code generation. This file is referenced by the `CODEGEN_INI` make file variable in `project_settings.mk`. Note that by default, ASCET-SE uses `codegen_example.ini` in preference to this file. The `EXAMPLE_MODE` make file variable in `project_settings.mk` must be set to `FALSE` to change this behavior.

- `.\target\trg_<targetname>\codegen_ecco.ini`

  Contains target-independent settings for code generation. This file is included in by `codegen_<target>.ini`. This file is read only by ECCO.

Together, these files control the following properties:

- code appearance, e.g., the naming of variables

- code generation, e.g., initialization of variables, and use of `#pragma` statements
- inclusion of operating system, e.g., selection of message semantic, creation of hook routines, and generation of the OIL description file

The first section of `codegen_<target>.ini` offers the possibility to include other `*.ini` files. `codegen_ecco.ini` is inserted automatically, other files can be added. Since `[INCLUDE]` is the first section, the settings in the included file(s) are made first, and afterwards, the settings defined in `codegen_<target>.ini` are made. Thus, `codegen_<target>.ini` can be used to make specific settings that override those in the other two files.

The options are described in detail in the `codegen[_*].ini` files themselves.

> **Note**
>
> *The configuration files are always read at the start of code generation; therefore, changes take effect immediately. However, it is usually necessary to force code generation for all components in the current project to ensure that changes are applied. For this purpose it is recommended to call* **Build →Touch → Recursive** *before code generation is started.*

Including a user-defined `*.ini` file:

In a user-defined `*.ini` file, the include mechanism can be used to set specific options without changing the original `codegen_*.ini` files. Proceed as follows:

- Create the *`<MyIniFile>`*`.ini` file and place it in the target directory.
- In the `project_settings.mk` file, include the *`<MyIniFile>`*`.ini` file.

  ```
  ###################################
  ##  CODEGEN SETTINGS (ECCO)
  ###################################
  # complete path to codegen.ini  ↵
      (ECCO options)

  CODEGEN_INI =$(P_TARGET)/      ↵
        <MyIniFile>.ini
  ```
- In the *`<MyIniFile>`*`.ini` file, add the `[INCLUDE]` section at the first place.
- Include the `codegen_<target>.ini` file to set the target-specific default options.
- If necessary, include further `*.ini` files.

  ```
  [INCLUDE]
  File1=codegen_<target>.ini
  File2=<path>\<filename>.ini
  ...
  ```
- Add the `[ECCO]` section with your individual settings.

```
[ECCO]
<option1>=<value>
<option2>=<value>
...
```
These settings override settings in the included files.

The `codegen_*.ini` file which ASCET-SE V6.2 uses during code generation is defined in `project_settings.mk`.

A default installation of ASCET-SE V6.2 is configured to build projects using the `codegen_example.ini` file provided in the examples directory. Use of `codegen_example.ini` can be disabled by defining the `EXAMPLE_MODE` make variable `EXAMPLE_MODE` as `FALSE`. The following fragment of `project_settings.mk` shows the first part that must be changed.

```
EXAMPLE_MODE=TRUE
EXAMPLE_PATH=$(P_TARGET)/example
EXAMPLE_CONF_OIL=$(EXAMPLE_PATH)/confV50.oil

####################################################
## CODEGEN SETTINGS (ECCO)
####################################################
# complete path to codegen.ini (ECCO options)
ifeq ($(strip $(EXAMPLE_MODE)),TRUE)
   CODEGEN_INI =$(EXAMPLE_PATH)/codegen_example.ini
else
   CODEGEN_INI =$(P_TARGET)/codegen_tricore.ini
endif
```

The other parts that use `EXAMPLE_MODE` require adaptation, too.

5.2   The `target.ini` File

Each ASCET-SE target is supplied with a target description file called `target.ini`. The contents of this file are used to configure the OS editor (see ASCET online help). In addition, the file contains internal configuration settings for ASCET-SE that must not be altered by the user.

The entries allowed in `target.ini` are described in this section. The file must follow the Windows `*.ini` format.

By default, `target.ini` includes definitions that match the generic or default target microcontroller variant provided with RTA-OSEK. A target directory may provide additional `target_<variant>.ini` files where `<variant>` is the name of a corresponding RTA-OSEK microcontroller target variant.

All variants of a microcontroller share the same CPU architecture but differ in peripherals. This often means that each variant of a microcontroller has a different number of interrupt vectors and/or mapping between vector addresses and peripheral interrutp sources. The correct variant is required if interrupts need to be configured in the ASCET Project Editor.

To use a different target variant:

- Rename `target.ini` as `target_default.ini`
- Choose the variant required
- Rename `target_<variant>.ini` as `target.ini`

The following tables describe the contents of a `target.ini` file.

> **Note**
>
> *Modifications to the* `target.ini` *file are effective only after restarting ASCET. This is also true for a change between different targets or target variants.*

Section `[Target]`*:*

`type=<target type>`	Unique identifier for the target. Do not change this setting.
`label=<target name>`	A label to be shown in the ASCET user interface.
`compilerTools=<compiler list>`	List of compilers available for the target. The entries are separated by blanks.
`osTools=<OS list>`	List of operating systems available for the target. The entries are separated by blanks.

Compiler settings can be made via the "External Tools\Compiler\<*compiler name*>" node in the ASCET options window.

OS settings:

`maxCoopLevels=<n>`	Max. allowed number of cooperative priority levels. For OSEK OS, `maxCoopLevels` is set to 6 by default.
`maxPreempLevels=<n>`	Max. allowed number of preemptive priority levels. Equal to `numHWlevels` + `numSWLevels` - `maxCoopLevels`.
`numHWLevels=<n>`	Number of hardware levels, equal to the number of hardware interrupt priorities on the target. (Further information about interrupt levels can be found in the RTA-OSEK User Guide or RTA-OSEK Binding Manual for the target.)
`numSWLevels=<n>`	Number of software levels, defined by the OS. For RTA-OSEK this will usually be `n=16` or `32` depending on the target.
`event:<n>=<identifier>, <x>,<y>,<address>`[a]	Description of an interrupt source, `n` is the event number, `identifier` denotes the event, `x` and `y` are min. and max. priority, `address` is the interrupt vector address.

a: These entries are usually not changed by the user.

Sections [<osname>]

The `target.ini` file contains one section [<osname>] for each operating system that can be used with the target.

> **Note**
>
> *For the purposes of* `target.ini` *files, an AUTOSAR RTE is handled in the same way as an operating system.*

The settings define the default paths, library names and options for each OS supported by the ASCET-SE target.

P_OS_INCLUDE	Comma-separated list of path names for OS header files.
P_OS_LIBRARY	Comma-separated list of path names for OS-specific libraries.
OS_LIBS	Comma-separated list of OS libraries to be linked with the project.
OS_CONFIG_TOOL_CMD	Command line options to be passed to the the OS configuration tool.
PROJ_OIL_FILE	An OIL file which is the entry point for the example project. Only required for integration with an OSEK OS. **Default:** $(EXAMPLE_CONF_OIL) which refers to the conf_<version>.oil file in <install_dir>\target\trg_<target-name>\example.

The values are automatically included in the "OS Configuration" node in the "Project Properties" dialog in ASCET's Project Editor. It is not necessary to adapt these settings in `target.ini` to suit an individual project. Instead, project-specific changes are best entered as overrides in the "OS Configuration" node by selecting **Enable OS Configuration**. The configuration options are described in the ASCET online help.

Default OS settings are specified relative to $(P_OS_ROOT) which defines the root installation directory of the OS. This is set globally in ASCET for each supported OS in the respective subnode of the "External Tools\Operating System" node in the ASCET Options window.

> **Note**
>
> *The default settings for RTA-OSEK are:*
>
> ```
> P_OS_INCLUDE = $(P_OS_ROOT)\<targetname>\inc
> P_OS_LIBRARY = $(P_OS_ROOT)\<targetname>\lib
> OS_LIBS = rtk_s.<lib>
> OS_CONFIG_TOOL_CMD = -ds
> ```
>
> *These settings use the RTA-OSEK Standard Status library (indicated by the `s` after `rtk_`) and force the RTA-OSEK configuration tool to generate Standard Status data structures regardless of the setting in the OIL file (indicated by the `-ds` command line option).*
> *If a different library and/or build level is required then both the library and the tool options must be modified. The library designator must match the `-d` parameter and can be one of `s`, `t`, `e`, `ts`, `tt`, `te`, `att`, `ate`.*
> *For example, to use Extended (debug) status use `rtk_e.<lib>` and `-de`.*

5.3   The `memorySections.xml` File

ASCET models allow data and code to be assigend to different memory classes. Memory classes are defined abstractly and given unique names, for example sections might be IROM (Internal ROM), EXT_RAM (EXTernal RAM), FLASH (FLASH memory). In addition, the ASCET code generator automatically creates certain memory class names depending on the context, e.g., for references or virtual parameters.

During the code generation process, the memory class names need to be converted into actual names, compiler-specific pragmas and type qualifiers. Both the memory class names and the conversion of memory class names are defined in an XML-based memory section defintion file called `memorySections.xml`.

A sample configuration file of that name is provided for each target, it can be found in the target directory. If you need different section names or settings then the file needs to be modfied. Details on how to write `memorySections.xml` files are provided in the file `ReadMe_memorySections.html` located in the target directory.

The ANSI C target includes three sample configuration files:

- `memorySections.xml` defines the memory sections for standard code generation. It is used when non-AUTOSAR code generation is selected.

- `memorySections_AUTOSAR.xml` defines the memory sections for AUTOSAR code generation. It is used by ASCET automatically when AUTOSAR code generation is selected. The sections are compatible with AUTOSAR's Memory Mapping (`MemMap.h`) and Compiler Abstraction (`Compiler.h`, `Compiler_Cfg.h`) concepts.

- `memorySections_AUTOSAR4.xml` defines the memory sections for AUTOSAR code generation, assuming AUTOSAR Release R4.x conventions (function parameters passed by reference use a pointer instead of a const pointer). The file can be used instead of the standard `memorySections_AUTOSAR.xml` by renaming it to `memorySections_AUTOSAR.xml`.

The definition of memory classes depends on the target and compiler. Refer to the compiler documentation when adjusting the sample file to your needs.

At the beginning of the `memorySections.xml` file, the default memory classes for the following four memory class categories are defined:

- `Code` – memory classes for code (e.g. methods, processes etc.)
- `Variable` – memory classes for variables
- `Characteristic` – memory classes for parameters
- `ConstData` – memory classes for structural data (type descriptor information for components)
- `DistSearchResult` – memory classes for distribution search results

The default memory classes for the categories depend on the target; an example for such a definition can look like this:

```
<MemClassCategories>
   <Code defaultMemClass="ICODE"/>
   <Variable defaultMemClass="IRAM"/>
   <Characteristic defaultMemClass="IFLASH"/>
   <ConstData defaultMemClass="IFLASH"/>
   <DistSearchResult defaultMemClass="DISTRRAM"/>
</MemClassCategories>
```

The definitions of individual memory classes appear in the `<MemClasses>` section. A memory class definition looks like this:

```
<MemClass>
   <name>string</name>
   <guiSelectable>Boolean</guiSelectable>
   <prePragma>string</prePragma>
   <postPragma>string</postPragma>
   <typeDef>string</typeDef>
   <typeDefRef>string</typeDefRef>
   <funcSignatureDef>string</funcSignatureDef>
   <constQualifier>Boolean</constQualifier>
   <volatileQualifier>Boolean</volatileQualifier>
   <storageQualifier>string</storageQualifier>
   <description>string</description>
   <category>[1]string</category>
</MemClass>
```

Code parts set in *italics* have to be replaced by appropriate values. The elements and their meanings are described in the `memorySections.xml` file in your target directory (e.g., `...\ETAS\ASCET6.2\target\trg_mpc55xx\memorySections.xml`).

String elements may contain line breaks, entered as `\n`. Some string elements can use macros. The macros available for template definitions are also described in the `memorySections.xml` file in your target directory.

5.3.1   Defining a Memory Class

The following steps must be performed to define a memory class and assign ASCET variables to it:

[1.] A memory class definition can contain several `<category>` parameters.

Step 1

Variables are assigned to the required memory class (in the "*Memory Location*" combo box) in the ASCET implementation editor. The class names available are those defined in the target-specific configuration file `memorySections.xml` (cf. section 5.3 on page 66).

To provide a different set of names, or to add new memory classes, you need to edit the classes in the `<MemClassCategories>` declaration of `memorySections.xml`. Each memory class category you define must have a corresponding `<MemClass>` definition.

Step 2

After compilation, the memory sections present in the object files must be located in the microcontroller's memory space. The linker control file defines the mapping of memory sections to address ranges. An example linker control files can be found in the `.\target\trg_<targetname>\example\` directory of each target. The example can be modified to the needs of your project or you can provide your own file.

If you choose to write your own linker control file, then the `MEM_LAYOUTFILE` variable in the `project_settings.mk` needs to be modified to reference the name and path of your file, e.g.:

```
MEM_LAYOUTFILE  = my_layout_file.inv
```

When you change the memory layout file or linker invocation file, make sure that the following constraints are met:

- **`VIRT_PARAM` section**

  This memory section should be placed beyond your real memory range, since virtual parameters are only important for calibration tools like INCA.

- **`VATROM` section**

  This memory section should be placed beyond your real memory range, and `VATROM` should not interfere with the placement strategy of other memory sections. This memory section is only used to collect virtual address tables used by the hex file reader to extract correct addresses of all project elements (ASAM-MCD-2MC generation). Therefore all other objects should be placed in memory independent of whether the `VATROM` section is used or not.

For MPC55xx and MPC56x targets only, the `a_sect.h` file has to be adapted, too. Details can be found in the compiler toolset manual.

5.3.2   Migration of Legacy Projects

ASCET projects developed with ASCET-SE V5.x define memory classes using `hip.db`, `target.ini` and `codegen.ini`. Such projects can be migrated to later versions of ASCET-SE by converting the older form of memory class definitions into a `memorySections.xml` file.

ASCET-SE provides a perl script, `convert_hip_db.pl`, in the `.\target\trg_<targetname>\scripts\` subdirectory for this purpose.

Migrating memory class definitions:

- Copy the `convert_hip_db.pl` file to the directory containing the old `hip.db`, `target.ini` and `codegen.ini` files.
- Run `convert_hip_db.pl` from a command line window.

> **Note**
>
> *Use the Perl version provided in the `Tools` subdirectory of ASCET V6.2. The conversion may fail with older Perl versions.*

The command line window logs the procedure, and lists relevant entries from `codegen.ini` and `target.ini`, as well as the memory classes imported from `hip.db`.
The following figure shows an example for a conversion where `codegen.ini` contained no relevant entries.

```
T:\M\test>c:\etas\ASCET6.1\tools\perl588\bin\perl.exe convert_hip_db.pl

        reading defaults for memory categories from <codegen.ini> ...
        done
        reading memory categories from <target.ini> ...
                found final default [mapMemClass_Default]    = IFLASH
                found final default [mapMemClass_Parameter]  = IFLASH
                found final default [mapMemClass_Variable]   = IRAM
                found final default [mapMemClass_Executable] = ICODE
        done
        reading memory classes from <hip.db> ...
                memory class COMPILER imported
                memory class IRAM imported
                memory class ERAM imported
                memory class NURAM imported
                memory class IFLASH imported
                memory class EFLASH imported
                memory class ICODE imported
                memory class ECODE imported
                memory class ERC_RAM imported
                memory class ERC_ROM imported
                memory class ERC_IDA imported
                memory class ERC_FAR_ROM imported
                memory class ERC_TRACE_RAM imported
                memory class _OSSTACK imported
                memory class _CUSTACK imported
                memory class DISTRRAM imported
                memory class REFRAM imported
                memory class VIRT_PARAM imported
                memory class VATROM imported
                memory class SERAP_WORK imported
                memory class SERAP_REF imported
                memory class OSEK_COM imported
        done
        writting memory classes to <memorySections.xml> ... done

T:\M\test>
```

- Check the new `memorySections.xml` file and adjust the attributes, if necessary.

5.4   Build System Control & Configuration Settings

ASCET-SE uses a "make"-based build system for running the code generator, the compiler and the linker. The basic control is shown in Fig. 5-1:
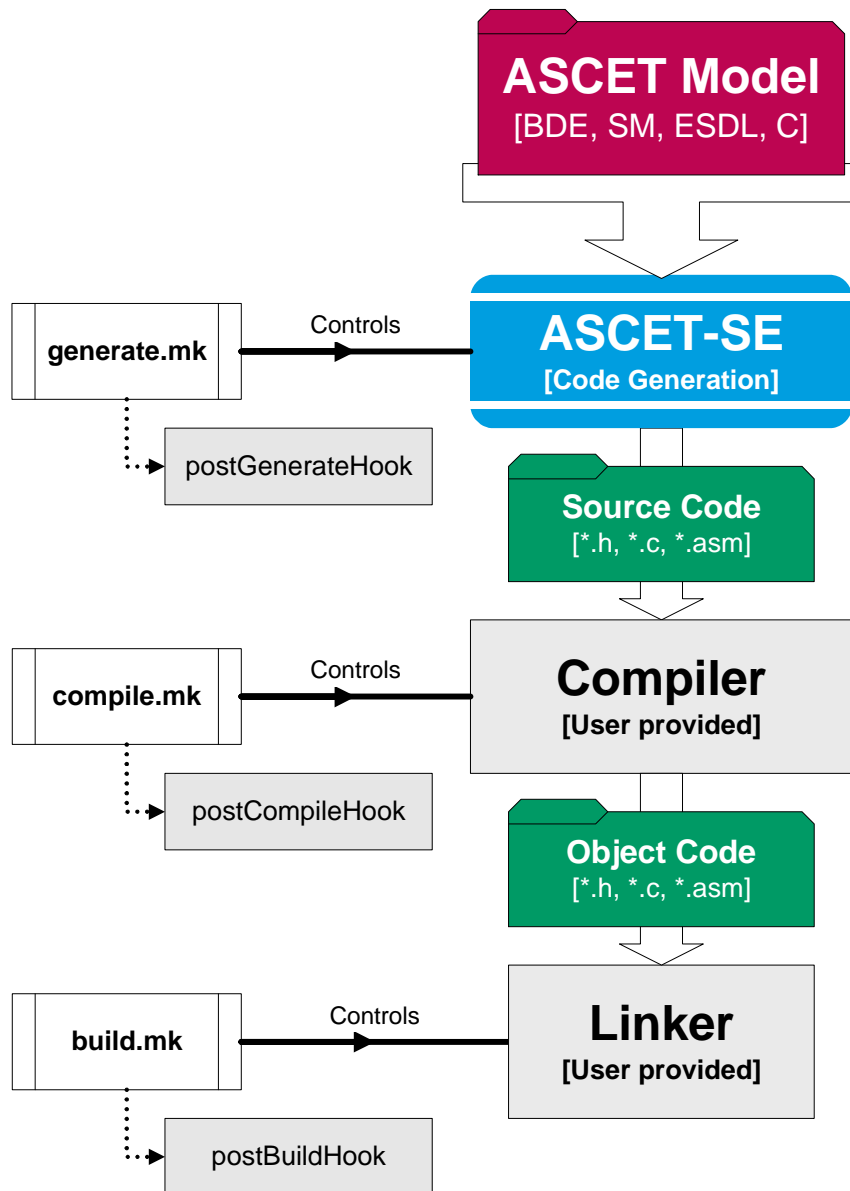
Fig. 5-1   Build system – basic control

The make process is managed using GNU Make. All make files and build scripts support paths with spaces.

- If a path containing spaces is to be used in a *makefile*, ASCET converts it to a Windows shortname format (for example, `c:\Documents and Settings` would be converted to `c:\DOCUME~1`).

- If a path containing spaces is to be used in a *batch file*, ASCET generates it encapsulated in `"`, or converts it to Windows shortname format.

The `makefile` file itself is generated and run whenever you select an option from the **Build** menu, using the information you specify in the project properties. The following is an excerpt from the `makefile` file, using the MPC56x with RTA-OSEK as example:

```
# path definitions
P_TGROOT = C:\etas\ascet6.2\target
P_TARGET = c:\etas\ascet6.2\target\trg_mpc56x
...
P_CCROOT = c:\compiler\diab\5.0.3
...
# phase definition
include $(P_TARGET)\compile.mk
```

The following sections describe how these phases are controlled and explain how each one can be customized via configuration files that are located in the *target-specific subdirectory*.

5.4.1 Project Settings - make file `project_settings.mk`

This make file defines project-wide configuration settings and can be found in in the target directory (e.g., `.\target\trg_<target>\` `project_settings.mk`).

The file `project_settings.mk` can be modified by the user and thus be adjusted to the project requirements, and it is included by the make files `compile.mk` and `build.mk`.

`project_settings.mk` is shipped with example mode switched on, i.e. the variable `EXAMPLE_MODE` is set to `TRUE`. This means that the settings given in the example files (see "Directory .\target\trg_<targetname>\example" on page 36) are used for the build process. To use your own configuration files, set `EXAMPLE_MODE=FALSE` and adapt further settings in `project_settings.mk`.

The parameter `STOPWATCH_TICK_DURATION` tells ASCET the length of a single tick of the dT time reference in nanoseconds. The value specified must match your target hardware configuration for dT timings in ASCET to be accurate.

5.4.2 Target and Compiler Settings – Make Files `target_settings.mk` and `settings_<compiler>.mk`

The make file `target_settings.mk` is included by the two make files controlling compiling and linking (`compile.mk` and `build.mk` respectively) and includes, in turn, `settings_<compiler>.mk`.

The `settings_<compiler>.mk` file defines file extensions, call conventions for precompiler, compiler, linker and other programs, as well as paths for program calls, include files and libraries. Command line parameters for compiler and linker calls are defined here, too.

You can change the values set in the `COMPILER SETTINGS` section to include another compiler than the preset one selected in the project properties. If you do so, make sure that all compiler-specific settings are correspondingly modified as well.

5.4.3 Code Generation – Make File `generate.mk`

This make file should not be modified by the user. It controls the ECCO generation process. All project and target-specific files are passed to ECCO here. For example, the Make variable `FILES_HEADER_PROJ` is defined here, which contains all generated header files of a project.

5.4.4 Compilation – Make File `compile.mk`

This make file controls the translation process. All files corresponding to the project are compiled and assembled here using the appropriate options. As a result, all object files are written into the `cgen` directory. Additionally, all compiler errors are evaluated and transferred to ASCET, if necessary. If an error occurs during compilation, the generation process is terminated and an error window is displayed.

"Smart-Compile"

ASCET-SE supports the option to re-compile only those C source files that have changed since the last build. The code is compared explicitly to find out whether a re-compilation is necessary.

Smart-Compile is controlled by two make variables:

- `COMPILE_MODE` in `compile.mk` specifies whether Smart-Compile is active or not. `COMPILE_MODE` is either `smartCompile` (smart compilation – check code explicitly for changes) or `compile` (conventional compilation behavior – only check timestamps). Smart compile is enabled by default.

- `SMART_COMPILE_COMPARE` in `smart_compile.mk` specifies the file comparison and is either `smart` (ignore only time and date of generation within comments, default), `strict` (do not ignore anything), or `relaxed` (ignore anything within arbitrary C comments).

When using Smart-Compile, several intermediate files are generated during compilation. These files are of no relevance for the user.

The "Smart-Compile" feature has led to an increased complexity and number of make files with respect to earlier versions. Not all details can be described here. To avoid problems, it is thus highly recommended to change the `project_settings.mk` file and, if necessary, the `target_settings.mk` file only.

5.4.5 Build – Make File `build.mk`

The link process is controlled by `build.mk`. The compiled object files and the required libraries are integrated into an executable program file which is written to the `CGen` directory.

The build process can be customized be editing `project_settings.mk`. Edits to `build.mk` itself should not be required.

Linker/Locator Control

The build process controlled by `build.mk` uses the Linker/Locator provided by the compiler toolchain to allocate parts of the executable program (code, static data, dynamic data etc.) to physical memory areas (RAM, ROM etc.) on the microcontroller. This process is controled by linker/locator control file. The file fomat is specifc to the compiler toolchain. The file contents are specific to your microcontroller variant (i.e. different devices with diffrenent memory layouts or sizes will need different linker/locator control files.

The linker/locator file ASCET uses is specified by the `MEM_LAYOUTFILE` variable in `project_settings.mk` file (see section 5.4.1). The variable must reference a valid linker-locator control file for your microcontroller.

A sample linker/locator file is supplied with each ASCET-SE target. and can be found in the `.\target\trg_<targetname>\example` folder.

You will need to consult both your compiler documentation and your microcontroller documentation to make changes to the file.

5.5   Customizing Code Generation

5.5.1   Banners

Banners in the generated code are described in the "Project Editor" section of the ASCET online help.

5.5.2   Formatting Generated Code – the `.indent.pro` Configuration File

The code formatting utility "Indent" can be used to re-format generated code. The properties of the code format can be widely influenced this way. The `.indent.pro` file, found in the target directory, serves for the configuration. You can find a detailed documentation of Indent's capabilities in `<install dir>\..\ETAS Manuals\ASCET V6.2\Tools\indent.html`, that is installed together with the ASCET-SE documentation. Indent is redistributed under the "GNU Public License".

5.5.3   Code Post-Processing

ASCET-SE offers the user the possibility to modify the generated code by means of Perl scripts. The called scripts must be specified in the make variable `POST_CGEN_PERL_MODS` in `project_settings.mk`, e.g.:

```
POST_CGEN_PERL_MODS= postCGenIndent postCGenSample
```

A sample file called `postCGenSample.pm` is included in the ASCET-SE delivery, in the `.\target\trg_<targetname>\scripts` directory. The calling conventions can be derived from that file easily. All scripts implemented by the user must comply with these conventions:

- Provision of a Perl macro called `process`

- Utilization of three invocation arguments. These arguments represent the path to the source code, a list of the C files and a list of H files to be processed.

Example:

```
sub process ($$$) {
   my $src_path,$c_files, $h_files) = @_;
   ...
}
```

In the delivered version, ASCET-SE uses the code formatting utility "Indent", which is called through the described mechanism as well. By specifying

```
POST_CGEN_PERL_MODS=
```

the execution of Indent can thus be suppressed. See also "Formatting Generated Code – the .indent.pro Configuration File" on page 73 for more details on "Indent".

5.6   Customizing the Build Process

5.6.1   Including Your Own Make Files

The make process in ASCET can be customized to run user-provided make rules at selected points in the overall build process. For this purpose ASCET-SE provides special *make targets*:

- `PRE_GENERATE_HOOK` is executed before code generation
- `POST_GENERATE_HOOK` is executed after code generation
- `PRE_COMPILE_HOOK` is executed before compilation
- `POST_COMPILE_HOOK` is executed after compilation
- `PRE_BUILD_HOOK` is executed before linking.
- `POST_BUILD_HOOK` is executed after linking.
- `POST_FILEOUT_HOOK` is executed after file out

The hooks can be defined in `custom_settings.mk`.

Your make file must conform to GNU make syntax. Documentation for GNU make is included in the ASCET-SE installation and can be found in `<install_dir>\ETASManuals\ASCETx.y\Tools`. Additional information can be found in the GNU-Make Manual (ISBN: 1-882114-80-9, not supplied).

5.6.2   Including User-Defined C and H Files

ASCET-SE can include additional C source files in the make process. Lists of file names can be defined in the `project_settings.mk` file. In addition, lists of path names can be indicated to specify where the compiler searches for the defined files. The following make variables can be used:

- `C_INTEGRATION` indicates, whether additional C source files are to be considered by the make process. Possible values are `FALSE` or `TRUE`.

   Note

   For RTA-OSEK integration, `C_INTEGRATION` must be set to `TRUE` because task and ISR bodies generated by ASCET-SE are placed in separate files which are compiled via the C code integration mechanism.

- `P_C_SRC_FILES` indicates a list of one or more paths for additional C source files, separated by blanks.
- `C_SRC_FILES` indicates a list of one or more additional C source file names, separated by blanks. If a file of a list of files is specified in `C_SRC_FILES`, a valid path must be provided in `P_C_SRC_FILES` and `C_INTEGRATION` must be set to `TRUE`.
- `P_H_SRC_FILES` indicates a list of one or more paths for additional H (header) files, separated by blanks.
- `LIBS_USER` contains a list of user-defined libraries. The respective path names have to be specified as parts of the file names.
- `P_ASM_SRC_FILES` indicates a list of one or more paths for additional assembler files, separated by blanks.

- `ASM_SRC_FILES` indicates a list of one or more additional assembler file names, separated by blanks.

The following example illustrates how the make file variables can be used (extract from `project_settings.mk`):

```
...
P_H_SRC_FILES = $(P_TARGET) $(P_DATABASE)/math
C_INTEGRATION = TRUE
P_C_SRC_FILES = $(P_DATABASE) $(P_DATABASE)/math
C_SRC_FILES   = mathop.c hwdriver.c errhndl.c
...
```

The files from the `C_SRC_FILES` list are compiled and linked by the ASCET make process.

5.6.3   Special Makefile variables provided by ASCET

Some special make variables can be used to access files at locations predefined by the system. These are:

- `$(P_TARGET)`, the specific path of the current target installation, e.g., `.\target\trg_mpc56x`,
- `$(P_TGROOT)`, the `.\target` path in the ASCET installation,
- `$(P_DATABASE)`, the specific path of the currently used ASCET data base,
- `$(P_CGEN)`, the `CGen` directory.

More information on the make variables is provided by the comments in `project_settings.mk`.

5.7   Controlling What is Compiled Using ASCET Header Files

The C code generated by ASCET-SE includes various C pre-processor directives that allow compile-time configuration using ASCET-SE header files. The header files are located in `.\trg_<targetname>\include` unless indicated otherwise.

5.7.1   The Include File `a_basdef.h`

The `a_basdef.h` file is included by all files generated by ASCET. It provides access, through further header files, to:

- the standard ASCET types (`a_limits.h`, `a_std_type.h`)
- target-dependent definitions (`tipdep.h`)
- the operating system interface (`os_inface.h`)
- project specific configuration (`proj_def.h`)

Project-specific configuration definitions for a project can be provided via the `proj_def.h` file. A template `proj_def.h` file can be found in th same include folder as `a_basdef.h`; the template shall be adapted by the user.

The `a_basdef.h` file itself should not be modified by the user.

5.7.2   The Include File `proj_def.h`

The supplied version of this file contains some macro definitions and an empty section that can be used for application-specific adaptations.

In particular, the file offers the possibility to include preprocessor commands that are valid throughout the complete code generated by ASCET. The switches noted below have a particular meaning in the code:

- `COMPILE_UNUSED_CODE`: This switch can be defined to compile code that is generated from the ASCET model, but not used by the model itself, e.g., a method that is never called.

  Example:

  ```
  #define COMPILE_UNUSED_CODE
  ```

- `DECLARE_PROTOTYPE_METHODS`: In ASCET, classes can be implemented as prototypes (see section 4.2.3 "Prototype Implementations"). This switch defines, whether (extern-)declarations shall be generated for the respective methods. This may become relevant, if the user intends to map method names to macros by means of pre processor commands (`#define`).

  Example:

  ```
  #define DECLARE_PROTOTYPE_METHODS
  ```

- `DECLARE_INLINE_METHODS`: For methods implemented as inline (see section 4.2.4 "Processes and Methods"), function declarations can be made visible for the compiler via this switch, if desired. Extern declarations for inline functions are usually not required, since the functions are expanded textually, so that their definitions must be known before they are used. ASCET takes care of that.

  Example:

  ```
  #define DECLARE_INLINE_METHODS
  ```

- Model-specific switches for the individual deactivation of single extern-declarations and type definitions.

- Switches for message configuration: the default optimization of message copies based on the operating system's priority scheme is not suited for all applications. The message handling can thus be configured, provided the `modularMessageUse` option is activated in the `codegen_ecco.ini` file. Four different variants exist:

  – Default message optimization:
    As a default, messages are optimized based on the operating system's priority scheme. In this case, the compiler switch

    ```
    #define __MESSAGES __OPT_COPY
    ```

    is used. It can be set by the user explicitly as well.

  – No message copies:
    Messages are used like global variables in this case. No copies are generated. This can be achieved using the compiler switch:

    ```
    #define __MESSAGES __NO_COPY
    ```

> **Note**
>
> *For methods in modules, **only** `__OPT_COPY` and `__NO_COPY` are available. Other optimizations are not supported.*

  – No message optimization (copy always):
    Messages are always copied using the compiler switch:

```
#define __MESSAGES __NON_OPT_COPY
```
In this case, no optimization takes place.

– If supported by the respective operating system, the OSEK COM message definition can be used:

```
#define __MESSAGES __OSEK_COM
```

6   Interpolation Routines

> **Note**
>
> *The interpolation routines provided with ASCET are for example only. They are not intended for use in production ECUs or development ECUs running in a vehicle. See Chapter 2.2 for further details.*

If your project uses characteristic tables then it is necessary to provide interpolation routines. Suitable interpolation routine libraries named `intpol_<target>_<compiler>.<libext>`[1] and the header file `a_intpol.h`[2] are delivered with ASCET-SE. These files contain several routines for the interpolation of characteristic curves and maps for various combinations of data types.

For characteristic curves and maps, over 500 possible combinations of input and output data types exist, each of which must have its own interpolation routine. However, since only a few of these combinations are actually used in a real project (usually less than 10), it does not make sense to deliver all 500 additional routines with ASCET-SE or to always integrate them into the code. The library, therefore, does not include the entire set of interpolation routines.

Further routines can be generated automatically at need. This is done by using the batch file `intpol_<target>_<compiler>.bat`[2] and a Perl interpreter provided with the system. The generated files are then compiled into the new library.

> **Note**
>
> *The generation of interpolation routines is described in the* `ReadMe_Interpolation.html` *file in the* `.\target\trg_<target-name>\Intpol` *interpolation routine directory.*

Interpolation routines use the following naming convetion:

- Distributions: `RoutineName_<Distribution-Type>`
- 1d Tables: `RoutineName_<X-Axis-Type><Y-Value-Type>`
- 2d Tables: `RoutineName_<X-Axis-Type><Y-Axis-Type><Z-Value-Type>`

The following type combinations are supported by these libraries for normal characteristic curves and maps as well as group characteristic curves and maps (for fixed characteristic curves and maps, interpolation is performed without calling interpolation routines).

Distributions:

All `<Distribution-Type>`s (e.g. `u8`, `s16`, `r32` etc.).

1d Table Routines:

All combinations of `<X-Axis-Type><Y-Value-Type>` for all integer types (e.g. `u8u8`, `s8s8`, `u16s32` etc.) plus `r32r32` and `r64r64` values.

---

[1] In the `.\target\trg_<targetname>\intpol\lib` directory. Possible library extensions are `*.lib`, `*.a`, `*.h12`.
[2] In the `.\target\trg_<targetname>\intpol` directory.

2d Table Routines:

All combinations of `<X-Axis-Type><Y-Axis-Type><Z-Value-Type>` for all integer types (e.g. `u8u8u8`, `s8s8s8`, `u16s32u8` etc.) plus `r32r32r32` and `r64r64r64` values.

6.1   Use of Interpolation Routines

For each target, ETAS provides some example interpolation routines in a pre-compiled library. The library is not intended for production projects without additional assessment and quality assurance. Nevertheless the routines contained in the library demonstrate how interpolation routines are generated, referenced and linked to a project and can serve as a starting base for customer specific improved routines.

After ASCET-SE has been installed, a directory `\intpol` is located in the target directory of each installed target, e.g.,

    C:\ETAS\ASCET6.2\target\trg_<targetname>\intpol

The ASCET online help descibes the callbacks to interpolation routines required by ASCET.

The following example describes how ASCET uses interpolation routines assuming an interpolation routine for `GetAt()` for characteristic curves.

For `uint8` values, the `GetAt()` call logically required by ASCET is replaced by a call to the `CharTable1_getAt_u8u8()` method. ASCET accesses the routines via the `a_intpol.h` header file. Yon need to implement a method with the same C signature in your interpolation rotuine library. The library must be linked with the application.

When using the example source code provided by ASCET, follow the instructions of the included `ReadMe_Interpolation.html` file to generate the related library and link it during the make process.

6.2   The Interpolation Procedure

The interpolation procedure for all variants consists of two steps:

1. Searching the proper interval of interpolation points and deriving the offset, i.e. the distance between the interpolation point and the x-axis value to be interpolated.
2. Calculating the linearly interpolated value at the desired position.

For group characteristic curves/maps, the search result is stored in intermediate variables to avoid multiple calculations of the values for the various characteristic curves/maps.

For characteristic curves with equidistant interpolation node distribution (fixed characteristic curves), less memory is required because an offset and a distance are stored instead of a list of interpolation points. Instead of the search procedure, the nearest fixed interpolation node to the x-axis value is used.
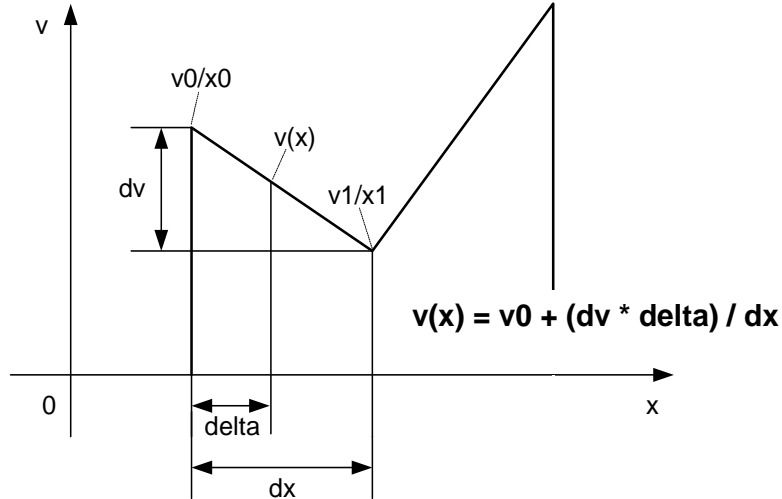
6.3   Accuracy and Allowed Range of Values

The supplied interpolation routines do calculation in the integer implementation to within ± 1.0 of the exact integer result.

The distance of interpolation nodes, and the difference between consecutive characteristic values cannot be arbitrarily large, due to a possible overflow during the interpolation.

Fig. 6-1   Interpolating a characteristic curve

The condition to avoid overflows is as follows:

$(dv * dx) < 2^{31}$ [$dv > 0$, a positive slope]

$(dv * dx) \geq -2^{31}$ [$dv < 0$, a negative slope]

For very steep characteristic curves (large differences between consecutive characteristic values), the number of interpolation nodes has therefore to be increased.

Within the current implementation, all routines are affected that use the data types `uint16`, `sint16`, `uint32` and `sint32`. To avoid wrong results in case of a possible overflow, the calculated value is checked by these routines. If the characteristic value does not fall within the value range of the two adjacent interpolation nodes, the value from the lower interpolation node is returned.

The algorithm for floating-point value interpolation differs only slightly from the one for integer value interpolation. In theory, an overflow can occur for floating-point values, too.

7   **Operating System Integration**

This chapter describes how ASCET-SE integrates with an operating system to provide real-time scheduling of ASCET processes.

The focus is primarily on integration with OSEK OS, in particular with ETAS' RTA-OSEK operating system. Integration with other OSEK-compatible operating systems is similar, but specific details will differ.

To integrate with the OS, ASCET-SE **generates**:

- an OS configuration file fragment that configures the OS to run the ASCET tasks and interrupts; and

- C code implementations of OS task and interrupt bodies that will be invoked by the OS

To integrate with the OS, ASCET-SE **requires**:

- an OS configuration file for system as a whole which must at least configure the OS objects required to schedule ASCET's tasks

- an implementation of a "main" program which configures the target hardware and starts the OS in the required application mode

- an implementation of a callback function to provide the dT model variable

7.1   Scheduling and the Priority Scheme

Tasks in OSEK OS are statically assigned a priority at configuration time. Zero represents the lowest priority task and higher numbers indicate higher priorities.

Tasks in OSEK can be scheduled *preemptive* and *non-preemptively*. These are configured by the "Scheduling" options `FULL` and `NON` respectively in ASCET task configuration (see the ASCET online help for details).

In addition to the standard OSEK OS scheduling modes, ASCET uses features of OSEK OS to support *cooperative* scheduling. This is configured by the "Scheduling" option `COOPERATIVE` in ASCET task configuration (see the ASCET online help for details).

Preemptive tasks can be preempted at any point during their execution by tasks with higher priority or any interrupt.

Non-preemptive tasks can preempt both preemptive and cooperative tasks, but once they are executing they cannot be preempted by any other task. Any higher priority task that becomes ready to run while a non-preemptive task is executing must wait until the non-preemptive task completes execution. However, non-preemptive tasks can be preempted by interrupts.

Cooperative tasks can be preempted at any point during their execution by pre-emptive and non-preemptive tasks and by interrupts. However, they can only be preempted by other cooperative tasks between processes.

To support these models, ASCET apparitions the OSEK OS task priority space into two parts:

1. Priorities used for cooperative scheduling

    The number of priority levels used for cooperative scheduling is defined by the configuration option **Coop. Levels** (in the "OS" tab of the project editor). Cooperative tasks can therefore be assigned priorities in the range `0..`**Coop. Levels**`-1`.

    The maximum value that the option can take is defined by `maxCoopLevels` in `target.ini`. The value of `maxCoopLevels` is defined to be 6 by default.

2. Priorities used for preemptive and non-preemptive scheduling

    The number of priority levels is equal to the maximum number of tasks supported by RTA-OSEK on the target minus the maximum number of cooperative levels. The value is equal to `numSWLevels - maxCoopLevels` in `target.ini`.

    Preemptive and non-preemptive tasks can therefore be assigned priorities in the range `0..numSWLevels - 1`.

The ASCET partitioning is overlaid onto the OSEK OS priority scheme when the OS configuration is generated.

For interrupts, ASCET uses the *Interrupt Priority Level* (IPL) model of RTA-OSEK. In this model, RTA-OSEK standardizes IPLs across all target microcontrollers, with IPL 0 indicating user level, where all tasks execute, and an IPL of 1 or more indicating interrupt level[1]. The maximum IPL which can be assigned is equal to the priority of the highest priority OSEK OS Category 2 ISR supported by the microcontroller. The maximum level is target dependent; it is equal to the setting of `numHWlevels` in the `target.ini` file in the target directory.

Note
Do not confuse IPLs with task priorities. An IPL of 1 is higher than the highest task priority used in your application.

Fig. 7-1 shows the relationship between task and interrupt priorities in the OS and ASCET.

---

[1] The IPL concept is explained in more detail in the RTA-OSEK User Guide. Specific details about how IPLs are mapped onto target hardware interrupt priorities are provided in the RTA-OSEK Binding Manual for the microcontroller.
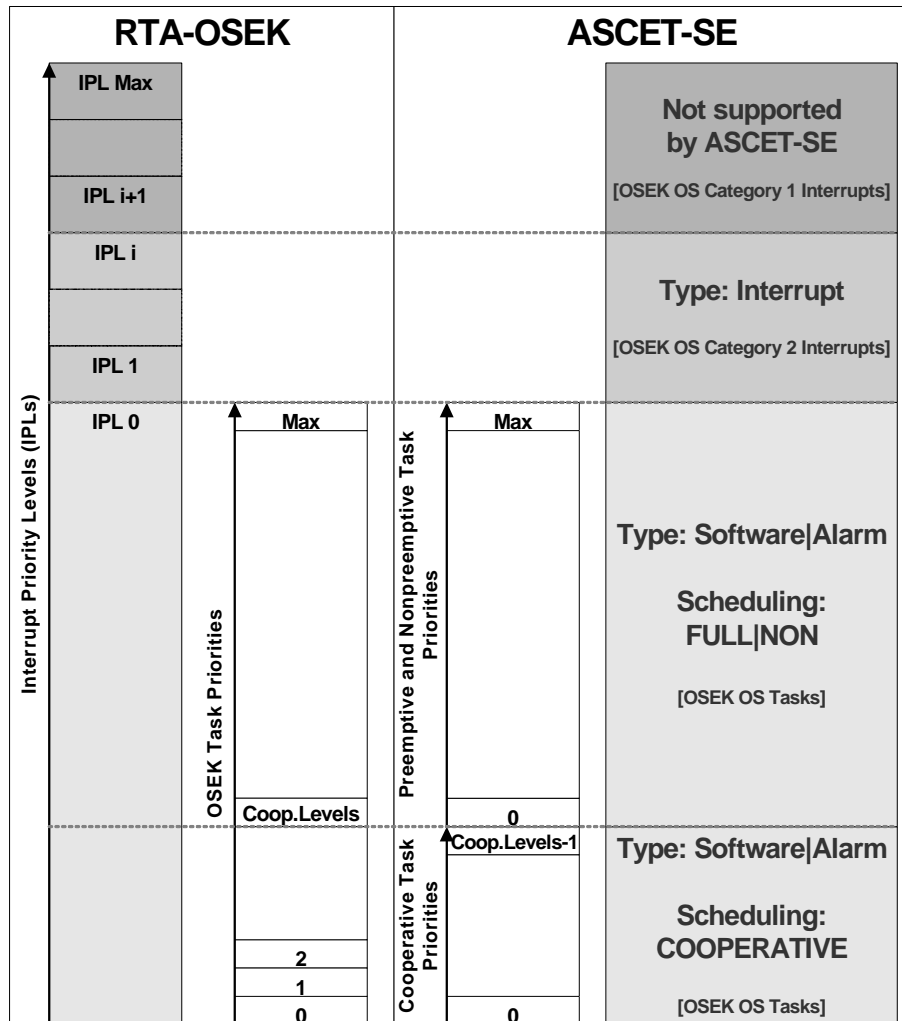
| RTA-OSEK	ASCET-SE

Fig. 7-1   Priority Levels

7.2   Setting Up the Project

7.2.1   Generating ASCET's OS Configuration File

During code generation for either RTA-OSEK or Generic OSEK, an OS configuration file called `temp.oil` is generated automatically using the configured OS template file. This file contains an OSEK Implementation Language (OIL)[1] configuration for the OS objects declared in ASCET, e.g. tasks, ISRs, resources, messages, alarms and application modes.

[1.] Details about OIL can be found on www.osek-vdx.org.
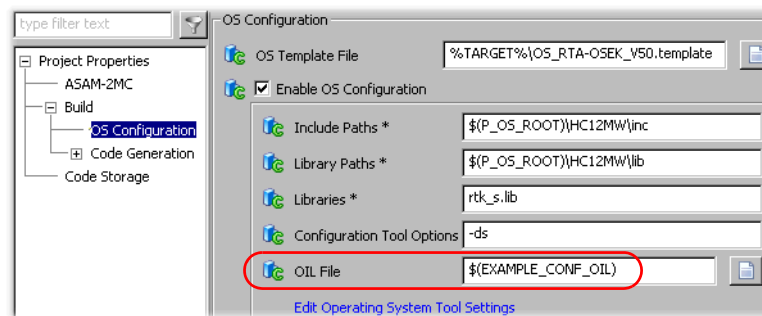
Fig. 7-2   Selecting the OS and the template on project settings

7.2.2   Providing Additional OS Configuration

The `temp.oil` file does not contain a complete OS configuration. Additional OS configuration is required to integrate ASCET with the OS. The following definitions are required:

- An OSEK OS object that defines global OS settings, including the build status, error logging modes and any hook routines required.

- An OSEK COUNTER that defines the counter used to drive the alarm tasks generated by ASCET. By default, ASCET expects the name to be `SYSTEM_COUNTER`. The name of the COUNTER is defined in the OS template file.

- An OSEK Category 2 ISR that provides the real-time "tick" for the COUNTER.

This additional configuration is provided as a framework OIL file. The framework file to be used for a project is specified in the Project Properties in the "OIL File" field of the "OS Configuration" node as shown in Fig. 7-2. Further details about configuration can be found in the ASCET online help.

An example framework OIL file for integration with RTA-OSEK is provided with the example application that can be found in `..\target\trg_<target-name>\example\conf<version>.oil`. This can be referenced using the macro `$(EXAMPLE_CONF_OIL)`.

It is recommended that you copy the example framework OIL file and adapt it according to your specific project needs.

The `conf<version>.oil` file supplied works with RTA-OSEK. RTA-OSEK uses "smart comments" (OIL comments with the form `//RTAOILCFG` or `//RTAOSEK`) to provide additional OS configuration that is required but not defined in OIL (for example, the interrupt priority level and the interrupt vector address).

The following objects are defined:

- **CPU** - The container for all other objects.

- **OS** - Defines the OS properties.

- **COUNTER** - The system counter defines the time base for the triggering of alarm tasks. By default, ASCET-SE expects this counter to be called `SYSTEM_COUNTER`.

Example:

```
COUNTER SYSTEM_COUNTER {
  MINCYCLE = 1;
  MAXALLOWEDVALUE = 4294967295;
  TICKSPERBASE = 1;
  //RTAOILCFG OS_TIMEBASE ts_SYSTEM_COUNTER;
  //RTAOILCFG OS_SYNC FALSE;
  //RTAOILCFG OS_PRIMARY_PROFILE ISR        ↵
     system_counter OS_PROFILE default_profile;
};
```

- **ISR** - The Category 2 interrupt that "ticks" the SYSTEM_COUNTER. The name of the ISR is not important, but by convention ASCET-SE uses system_counter.

  Example:

```
ISR system_counter {
  CATEGORY = 2;
  //RTAOILCFG PRIORITY = 1;
  //RTAOILCFG ADDRESS = 0x170;
  //RTAOILCFG OS_EXECUTION_BUDGET OS_UNDEFINED;
  //RTAOILCFG OS_BEHAVIOUR OS_SIMPLE;
  //RTAOILCFG OS_USES_FP FALSE;
  //RTAOILCFG OS_STACK {OS_UNDEFINED };
  //RTAOILCFG OS_PROFILE default_profile { };
  //RTAOILCFG OS_PROFILE default_profile { ↵
        OS_BASE OS_WCSU {OS_UNDEFINED }; };
  //RTAOSEK OS_TRACE 0;
};
```

- **COM** - Defines properties for message communication using OSEK COM.

  Example:

```
COM RTACOM {
  USEMESSAGERESOURCE = FALSE;
  USEMESSAGESTATUS = FALSE;
};
```

Other OIL objects can be defined here, too, as well as additional RTA-OSEK configuration information (see the RTA-OSEK User Documentation for details).

The generated temp.oil file is included using RTA-OSEK's auxiliary OIL file mechanism. The inclusion must be placed *after* the OIL CPU clause as shown below:

```
CPU rta_cpu {
  OS RTAOS {
    ...
  };
  ...
};
//RTAOILCFG OS_SETTING "AuxOIL" "1";
//RTAOILCFG OS_SETTING "AuxOIL0" "temp.oil";
```

The `system_counter` ISR must be implemented in external C code. An example is provided for each ASCET target in `..\target\trg_<target>\example\target.c`. Additional information can be found in the RTA-OSEK User Guide.

The duration of each `SYSTEM_COUNTER` counter tick in nanoseconds (which will usually equal the rate of the `system_counter` ISR) must to be entered into the "Tick Duration" field of the ASCET OS editor prior to code generation. For RTA-OSEK based systems, the value should be identical to the value of the macro `OSTICKDURATION_SYSTEM_COUNTER` in the generated `oscomn.h` file.
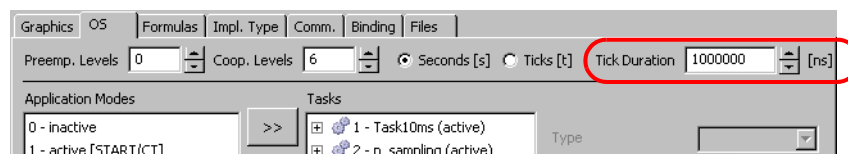
ASCET uses the value of Tick Duration for tick/time conversion for alarm tasks only. The value is unrelated to dT calculation.

7.3   Providing the Main Program

The main program, usually called `main`, is responsible for target hardware initialization and starting the OS in the required application mode.

By default, a build of an ASCET project will use an external main program provided in `..\target\trg_<targetname>\example\main.c`. The example main program for an embedded target configures the hardware to generate the `system_counter` interrupt every 1 ms and starts RTA-OSEK in the `active` application mode.

A different main program can be used by setting the `makefile variable` `EXAMPLE_MODE` in `project_settings.mk` to `FALSE` and either:

- configuring ASCET-SE to generate the main program in `conf.c` automatically (`Os-Config-C_gen_main=TRUE` in `..\target\trg_<targetname>\codegen_ecco.ini`.); or

- ensuring that ASCET-SE is configured to not generate the main program (`Os-Config-C_gen_main=FALSE`) and setting the variables `P_C_SRC_FILES` (and/or `P_ASM_SRC_FILES`) to refer to your own source code.

7.4   The `dT` Variable

ASCET provides each project with a model variable called `dT` (delta time). `dT` provides each task and interrupt with the time, in microseconds, which has elapsed since the start of the previous execution.

You can choose to scale the value of `dT` to represent a different time unit by providing an implementation formula (in the same way as for other ASCET variables). ASCET handles the scaling automatically.

In generated code, a special variable called `dT` is created globally for each project. `dT` holds the time elapsed between since the previous execution of a task/interrupt started.

dT is normally a *dynamic* value that holds the actual time that has elapsed between executions. The value of dT will change depending on how much interference (due to preemption) and blocking (due to resources being held or interrupt being disabled) a task or interrupt suffers.

To provide dT, ASCET needs to be provided with a free-running timer and must be told the duration of a tick of the timer in nanoseconds. This configuration is described in section 7.4.1.

In some use-cases, it is sufficient for dT to hold the *configured* period for alarm tasks. In ASCET this is called "static dT" and configuration is described in section 7.4.2.

The difference between dynamic and static dT (and the difference between a scaled and non-scaled dynamic dT) is shown below.
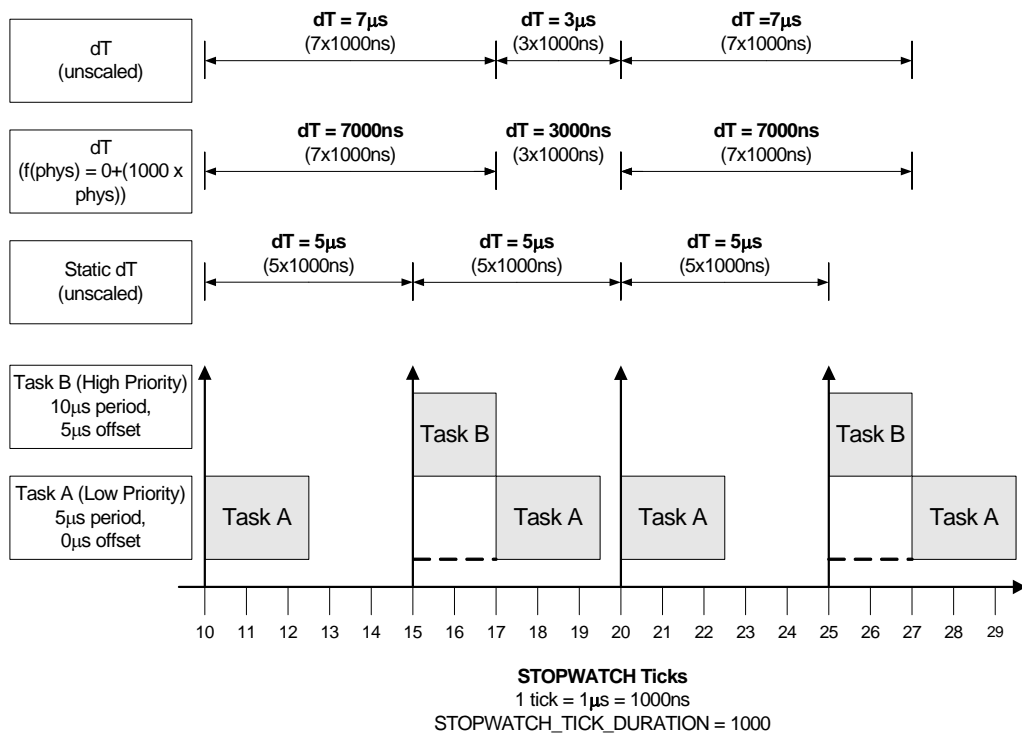
Fig. 7-3   Static and dynamic dT

7.4.1   Dynamic dT

To use dynamic dT, the option **Generate Access Methods for dT (Alternative: use OS dT directly)** must be enabled in the Project Properties. ASCET-SE will generate the code to use and calculate dT at runtime. However, to do this ASCET-SE must be given access to a free-running 32-bit timer source (see below).

ASCET generates a function called setDeltaT() that is used in each generated task body to update the ASCET model element dT (generated as dT_PROJECT_IMPL in the code). If the model element dT is scaled (i.e. it does not use the identity implementation) then ASCET-SE automatically ensures that the scaling is handled correctly. For example, if the model variable dT is implemented in milliseconds, the following code is generated:

```
void setDeltaT (void)
{
   TimeType dTMicroSeconds =
      (STOPWATCH_TICK_DURATION*dT)/(TickType)1000;
   (dT_PROJECT_IMPL = ((dTMicroSeconds/1000)));
}
```

Providing a Time Reference for Dynamic `dT` *Calculation*

ASCET uses a callback function called `GetSystemTime()` to get access the time reference for the `dT` value used by in ASCET models. The implementation of the callback must provide the current value of a free-running hardware timer on your target microcontroller.

The following steps are required to provide dynamic `dT`.

1. Enable the **Generate Access Methods for dT *** code generation option.
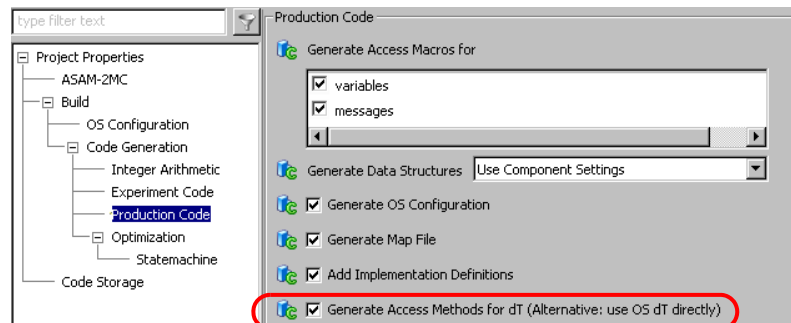
Fig. 7-4 Production Code options

2. Enable the following options in `codegen_ecco.ini`:

    ```
    Os-Config-C_gen_process_container=1
    Os-Config-C_gen_dt_calc=1
    ```

3. Ensure that the following line is **not** commented out in `.\target\ trg_<targetname>\include\os_inface.h`:

    ```
    extern TimeType GetSystemTime(void);
    ```

4. Provide an implementation of the `GetSystemTime()` callback function. The implementation of this function must return the value of a free running 32-bit hardware timer.

    When integrating ASCET-SE with RTA-OSEK, `GetSystemTime()` can be mapped onto RTA-OSEK's `GetStopwatch()` callback automatically by setting `ASD_OS_INTEGRATION` in `project_settings.mk` as follows.

    ```
    ASD_OS_INTEGRATION = ASD_OS_INTEGRATION_RTA ↵
       MAP_TO_GETSTOPWATCH
    ```

    RTA-OSEK's `GetStopwatch()` callback is required by the OS in timing or extended build. It provides the OS with access to a free-running 32-bit hardware timer for time measurement (see the RTA-OSEK documentation for details) – i.e. the RTA-OSEK callback provides identical functionality to that required by ASCET-SE for `GetSystemTime()`. Note that the implementation of `GetStopwatch()` must be provided in external C code.

An example implementation is supplied in `.\trg_<target-name>\example\target.c` in your target directory; here, the implementation from `..\example\trg_tricore\ target.c` is shown.

```
OS_NONREENTRANT(osStopwatchTickType)
GetStopwatch(void)
{
   /* Get the current value of the lowest 32 bits of
   the STM timer. */
   return (osStopwatchTickType)_STM_TIM0;
}
```

5. ASCET is told the duration of a `dT` tick in nanoseconds by the macro `STOPWATCH_TICK_DURATION` defined in `project_settings.mk` (see section 5.4.1):

```
# Free-running HW counter for GetSystemTime()
# has a tick every 50ns
STOPWATCH_TICK_DURATION = 50
```

These settings allow ASCET to calculate `dT` at runtime for use in the code generated from your ASCET model.

7.4.2   Static dT

ASCET-SE can be configured to provide alarm tasks with their configured inter-arrival time as a static `dT`.

Note

The value of static `dT` is only defined for alarm tasks. Other types of tasks and interrupts must not include processes that use `dT`.

To configure static `dT` you must

1. Disable the **Generate Access Methods for dT *** code generation option in Project Settings (see Fig. 7-4).

2. Enable the static `dT` option in `codegen_ecco.ini`:

   `Os-Config-C_gen_dt_static=`**1**

3. Enable `USE_ASD_CALC_SCALED_DT` in `project_settings.mk`

When these settings are made, ASCET generates a macro called `_ASD_TICKS_PER_TASK_PERIOD` in each task body that defines the task's configured period in ticks of the System Counter. For example:

```
TASK(task_100ms)
{
   #define _ASD_TICKS_PER_TASK_PERIOD 10
   ...
   /* Rest of task body */
   ...
   #undef _ASD_TICKS_PER_TASK_PERIOD
}
```

In this case, `SYSTEM_COUNTER` is being ticked every 10 ms, so the macro is set to 10 ticks (i.e. 10 ticks X 10 ms = 100 ms).

To convert the ticks into time for use in runtime calculations, or to handle any scaling of the model dT by an implementation formula, you must modify the macro ASD_CALC_SCALED_DT in proj_def.h. By default, the macro assumes an identify scaling and converts DT ticks into VAR time VAR assuming 1 DT tick = 1 VAR us as shown below:

```
#define ASD_CALC_SCALED_DT(VAR,DT) \
   do {\
      VAR = DT; \
   }while(0);
#endif
```

With static dT, a DT tick has the same duration in nanoseconds as a SYSTEM_COUNTER tick (i.e. it is equal to the value Tick Duration configured in the ASCET OS editor). To convert _ASD_TICKS_PER_TASK_PERIOD into microseconds, the macro would need to be modified to multiply DT by TickDuration (DT*10000000) and then divide the result by 1000 to convert from nanoseconds to microseconds (DT*10000000/1000=DT*10000), for example:

```
#define ASD_CALC_SCALED_DT(VAR,DT) \
   do {\
      VAR = DT*10000; \
   }while(0);
#endif
```

> **Note**
>
> *When doing any re-scaling you must ensure that any intermediate results do not result in overflow or underflow. It is your responsibility to ensure that this does not occur.*

7.4.3   Implementing Your Own dT Routines

If you require any special functionality from dT then you can provide your own implementation. In this case, the option **Generate Access Methods for dT (Alternative: use OS dT directly)** must be disabled (see Fig. 7-4).

ASCET-SE will not generate setDeltaT() or defined the dT variable. You must provide definitions of these externally in your own code. ASCET expects the function and the variable to correspond to the following C extern defintions:

```
extern TickType dT;
extern void setDeltaT();
```

Your implementation of TickType must be at least uint32. The unit of TickType variables is one tick (i.e. one increment) of the free-running hardware timer accessed through GetSystemTime().

```
extern TickType GetSystemTime()
```

Your implementation of setDeltaT() should be a void/void function that updates the global dT variable, taking account of any scaling defined in your model.

ASCET-generated code uses C macros to access dT functionality. Default implementations of the macros are provided in .\trg_<targetname>\include\os_inface.h. If you want to provide an alternative implementation of dT, the following macros in os_inface.h should be modified:

- `DEF_GLB_DT_MEASURE` — This macro is used in `conf.c`. It provides global variables or extern declarations necessary for the `dT` calculation.

- `DEF_TASK_DT_MEASURE` — This macro is used at the beginning of each task. It can be used to define task-local variables necessary for the `dT` calculation.

- `PRE_TASK_DT_MEASURE` — This macro is also used at the beginning of each task, after `DEF_TASK_DT_MEASURE`. Here, code can be inserted that calculates `dT` at the beginning of the task.

- `POST_TASK_DT_MEASURE` — This macro is used at the end of each task. Here, code can be inserted that restores the global `dT` variable for the other tasks.

7.5   Template-Based OS Configuration Generation

OSEK OS configuration files are generated by ASCET using a template-based mechanism. Templates (`*.template` files) are supplied for all supported Operating Systems and can be found in the `<installation directory>\target\trg_<targetname>` directories.

> **Note**
>
> *Templates are only used for generating OSEK-based Operating System configurations. The templating mechanism is not used for AUTOSAR RTE configuration.*

When an OS is selected in the "Project Properties" window, "Build" node, ASCET-SE will automatically select the default template for the chosen OS. The template in use is shown in the "Project Properties" window, "OS Configuration" node. No additional configuration is necessary.

Fig. 7-5 shows these two parts of configuration.

Tab. 7-1 shows which template is used for which OS, where `%TARGET%` is the path to the target directory.

The template for a chosen OS can be changed by entering the full path to the template file or by selecting a template file by clicking on the 📄 (**Open File**) button.

When OS configurations are changed in the "Project Properties" window, **Build** node, ASCET-SE will remember which template file is in currently in use for the selected OS.

At code generation time, ASCET-SE uses the template together with the configuration settings specified for the OS in the project editor to generate a configuration file for the chosen OS. The configuration file is always called `temp.oil.`

The template mechanism is highly flexible and OS configurations can be changed simply by modifying one of the supplied templates or by providing a customized template. This is of most use when an OS configuration that works with a specific 3rd party OSEK OS configuration tool is required.

Operating System	Default Template
RTA-OSEK 5.0	`%TARGET%\OS_RTA-OSEK_V50.template`
GENERIC-OSEK	`%TARGET%\OS_Generic-OSEK.template`
RTE-AUTOSAR Vx.y	\<empty\>

Tab. 7-1   Default templates for supported Operating Systems
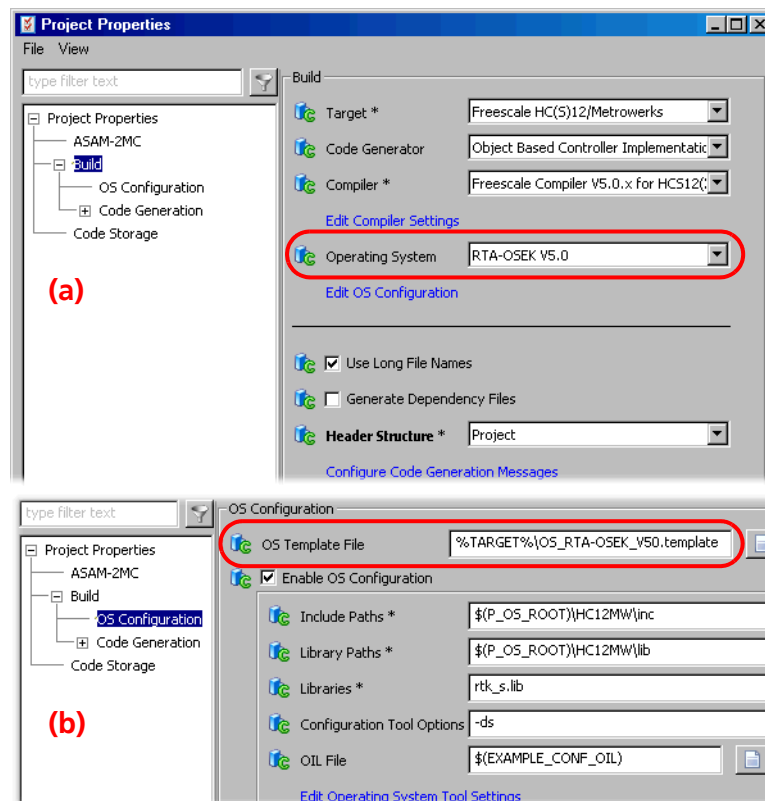
Fig. 7-5   Selecting the OS and the template in the "Project Settings" window (a: "Build" node, b: "OS Configuration" node)

> **Note**
>
> *The templating mechanism customizes the generation of OS configuration files only. It does not modify the properties of generated C code.*

7.6   Interfacing with an Unknown Operating System

ASCET-SE can be interfaced to an unknown operating system. This is particularly useful when working with the ANSI-C target. The generated code accesses the OS interface through the definitions in the `os_unknown_inface.h` file in the target directory.

7.6.1   Configuration of Tasks

ASCET generates task bodies with the following structure:

- Task definitions start with the `TASK` keyword and the task name, e.g.,

    ```
    TASK(t10ms){
    ```

- A list of processes assigned to the task in the form of function calls, e.g.,

    ```
    MODULE1_IMPL_process1();
    MODULE2_IMPL_process1();
    MODULE2_IMPL_process2();

    ...
    ```

- A function call to terminate the task:

    ```
    TerminateTask();
    }
    ```

The supplied `os_unknown_inface.h` file contains the following definitions of the `TASK` macro and `TerminateTask()`.

```
#define TASK(x) void task_ ## x (void)
#define TerminateTask()
```

These must be modified to the appropriate definitions for your OS.

The following code is obtained from the C preprocessor when using the default definitions

```
void task_t10ms (void)
{
  MODULE1_IMPL_process1();
  MODULE2_IMPL_process1();
  MODULE2_IMPL_process2();
}
```

It is recommended that the trigger mode setting for ASCET tasks is set to either *Software* or *Init* when interfacing with an unknown OS. Trigger modes *Interrupt* and *Alarm* require special OS support and should not be used unless you are confident that your OS can provide this.

7.6.2   Interfacing with the OS API

Calls to the OS use the OSEK OS naming conventions, but their implementation is not defined. All operating system calls are mapped to empty character strings using `#define` statements.

Example:

```
#define GetResource(x)
```

With this, the `GetResource` call in the generated code is removed by the pre-compiler, and ignored during compilation.

> **Note**
>
> *When the ANSI-C target is used, by default no ASCET features are supported that rely on OSEK OS functions (e.g. resources). This applies also to OSEK function calls used in the C code.*

By changing the `#define` statements, function calls can be mapped onto those provided by the your OS. e.g.:

```
#define GetResource(x) lock(x)
```

7.7   Template Language Reference

This section describes how templates can written and provides a reference to the OS objects to which ASCET-SE provides access.

7.7.1   Templating Basics

A template is an ASCII text file. When the template is processed by ASCET-SE V6.2, any content that is not enclosed by template tags [% and %] is written to the output `temp.oil` file.

> **Note**
>
> *Templates must have the extension* `.template` *to be recognized by ASCET-SE V6.2 as such.*

The template mechanism uses the "Template Toolkit" as the templating engine and any construct supported by the toolkit can be used in custom template. This section provides an overview of the template language constricts used in ASCET-SE templates. For a complete description of the capabilities of the templating engine, see http://template-toolkit.org/.

Listing 1. shows a template that contains no tags. When this is processed by ASCET-SE, the resulting `temp.oil` file contains identical content as shown in Listing 2.

1. Content of `MyFile.template`

    ```
    CPU MyCPU {
       ...
    };
    ```

2. Content of generated `temp.oil` file

    ```
    CPU MyCPU {
       ...
    };
    ```

Directives

The text between template tags is processed as a *directive* to the templating engine to do some kind of action. Directives can be placed anywhere in a line of text and can be split across several lines.

Expressions: Expression directives are replaced by the result of the evaluation in the output `temp.oil` file.

Expressions are typically used to evaluate the value of OS object properties provided by ASCET-SE. A complete list of objects and properties made available is provided in section 7.7.2.

The following example shows how to add a comment into the template that shows the number of interrupt and task priority levels by reading the `numOf-HardwareLevels` and `numOfSoftwareLevels` attributes from the OS object.

```
// There are [% OS.numOfHardwareLevels %]     ↵
    interrupt priority levels
// There are [% OS.numOfSoftwareLevels %] task ↵
    priority levels
```

Conditionals: The templating language provides a conditional construct. The following example shows how to add a comment into `temp.oil` depending on whether or not there are any OSEK COM messages defined.

```
[% IF OS.isEnabledOSEKCOM %]
// OS message objects need to appear here
[% ELSE %]
// No OS message objects need to be added
[% END %]
```

Iteration: The majority of OS configuration generation requires adding a configuration element for each object declared in the ASCET-SE V6.2 project configuration. ASCET-SE provides access to most configuration objects as a list that can be iterated over, writing out the correct configuration for each object.

The following example shows how to write out the correct configuration for an OSEK OS application mode.

```
[% FOREACH appmode IN AppModes %]
APPMODE [% appmode.name %];
[% END %]
```

Assuming that the list `AppModes` contains the items `Normal`, `Diagnostic` and `LimpHome`, the effect of processing the directive in the this example would be this OIL language fragment:

```
APPMODE Normal;
APPMODE Diagnostic;
APPMODE LimpHome;
```

Sub-Routines: Common operations can be placed in subroutines called BLOCKS. A block can contain any template text, including other directives. Each block must be uniquely named.

```
[% BLOCK Greeting %]
[% parameter %] World!
[% END %]
```

A block can be called from the main template using the `PROCESS` command. Variables that are used inside the block need to be passed in as parameters:

```
[% arg='Hello' %]
[% PROCESS Greeting parameter=arg %]
```

Blocks do not need to be defined before use, but they must be placed in the same file as the calls.

Including other files: External files can be included using the `INCLUDE` directive. The directive will add the contents of the specified file into the output.

> **Note**
>
> *The content of included files is not processed by the templating engine.*

Path can be absolute or relative. Relative paths are relative to the location of the template code generation path.

```
[% INCLUDE '..\RelativeDir\Relative.txt' %]
[% INCLUDE 'C:\MyFiles\Absolute.txt' %]
```

Note

It is recommended that path names are quoted using single quotes.

Comments: Comments in a directive are marked using the **#** symbol. Comments can span multiple lines. The following examples show single and multi-line comments respectively.

Example 1: Single line comment

```
[%# This is a single line comment %]
```

Example 2: Multi-line comment

```
[%# This
is
a
multiple
line
comment
%]
```

Chomping Whitespace: When a directive is placed on its own line and it evaluates to null, the templating engine will insert a blank line into the output. This includes any control flow directives that are placed on their own lines.

This can be avoided by "chomping" whitespace using an equals sign (=) as the first character after the open directive tag. A directive like this:

```
AAAA
[%= IF ConditionWhichIsFalse %]
BBBB
[%= END %]
CCCC
```

will result in an output like this

```
AAAA
CCCC
```

Note that blank lines have not been inserted.

7.7.2   Object Reference

The template can assess the OS configuration using pre-defined objects. The objects generally correspond to configuration items in an OSEK OS, though there are some non-OS objects provided to support legacy operating systems.

The following objects are accessible:

Object	Type	Description
OS	Structure	Contains general OS properties.
AppModes	List of AppMode objects	All application modes defined in current project.
Tasks	List of Task objects	All tasks (both software and alarm tasks) defined in current project.

Object	Type	Description
InitTasks	List of InitTask objects	All init tasks.
ISRs	List of ISR objects	All interrupt service routines.
Alarms	List of Alarm objects	All alarms used to activate tasks.
Resources	List of Resource objects	All resources used within current project.
Messages	List of Message objects	All messages used within current project.
UsedMessages	List of UsedMessage objects	All messages used by a Task or ISR.
Processes	List of Process objects	All processes used within current project.
Functions	List of Function objects	All functions used within current project.

Each object has a set of properties. Object properties are accessed using the "dot" notation, *<object_name>.<property_name>*, e.g. `task.prio`.

Note

Object and property names are case-sensitive.

The following example shows how to iterate over a list of task objects, extracting properties:

```
[% FOREACH task IN Tasks %]
  TASK [% task %] {
    PRIORITY = [% task.prio %];
    SCHEDULE = [% task.schedule %];
    ACTIVATION = [% task.activation %];
    ...
  }
[% END %]
```

The following sections describe the properties available for each object.

OS

An OS object defines the global properties of the OS. Exactly one OS object is defined.

Property	Type	Description
numOfCoopLevels	integer	Defines the number of cooperative priority levels.
numOfHardwareLevels	integer	Defines the number of hardware priority levels supported by the target.
tickDuration	integer	Defines the duration of a tick of the ASCET-SE system counter in nanoseconds.

Property	Type	Description
numOfSoftwareLevels	integer	Defines number of software priority levels supported by the target. For embedded targets, this is equal to the number of tasks the target supports (as defined in `target.ini`). For experimental targets, this value is equal to the priority of the highest priority software task plus the number of cooperative levels.

numOfPreempLevels	integer	Defines number of all preemptive levels. It is defined as
		`numOfHardwareLevels`
		`+ numOfSoftwareLevels`
		`- numOfCoopLevels`

isEnabledOSEKCOM	boolean	Defines if OSEK-COM messages, rather than ASCET messages, are used for inter-process communication. It is true if OSEK COM messages are used and false otherwise. If the value is `true`, then the generated OIL file shall include message definitions.

AppMode

The AppMode object defines an OSEK-like application mode.

Property	Type	Description
name	string	Name of the application mode.
initTask	string	Name of the init task to activate when the OS is started in this application mode.
timeTable	string	Name of time table to start when the OS is activated in this application mode. This is ERCOS^{EK} specific.

Task

A Task object defines the properties of an OS task defined in the ASCET project.

Property	Type	Description
name	string	Name of the task.
id	string	ASCET-SE internal identifier for the task.
prio	integer	Priority of current task. Higher integers are higher priorities.
prioERCOSEK	integer	Priority of current task following the ERCOS^{EK} priority scheme.

Property	Type	Description
schedule	NON / FULL	Defines whether the task can be pre-empted by other tasks or not. Equivalent to the OSEK OIL property SCHEDULE.
activation	integer	Defines the maximum number of queued activation requests for the task.
autostart	TRUE / FALSE	Defines if the task shall be autostarted.
autostartAppModes	list	List of application mode names in which the task shall be autostarted.
usedResources	list	List of resource names representing the resources used by the task.
usedMessages	list	List of OSEK COM message names used by the task.
usesFPU	TRUE / FALSE	Specifies whether the task uses floating point registers which will need to be saved and restored during an OS context switch. The value is TRUE if a floating point context save is required and FALSE otherwise.
usedProcesses	list	List of ASCET processes that shall be called by the task.
hook	MONITORING / NONE	The (non-OSEK) hooks used by the task.
deadlineMicroSeconds	integer	The maximum allowed time in micro-seconds between task activation and completion.
usesTerminateTask	TRUE / FALSE	Defines whether the task uses OSEK TeminateTask() API.

InitTask

Property	Type	Description
name	string	Name of the init task.
id	string	ASCET-SE internal identifier for the init task.
autostartAppModes	list	List of application mode names in which the task shall be autostarted.
usedProcesses	list	List of ASCET-SE processes that are called by the task.

ISR

Property	Type	Description
name	string	Name of current ISR.
prio	integer	Priority of current ISR. Priorities are target-independent and take values in the range 1 to `OS.numHWlev-els`. Priority 1 is the lowest priority.
prioERCOSEK	integer	Priority of current ISR following the ERCOS^{EK} priority scheme.
autostartAppModes	list	List of application mode names for which the ISR shall be autostarted. Not used in OSEK.
usedResources	list	List of resource names used by the ISR.
usedMessages	list	List of OSEK COM message names used within this ISR.
usesFPU	TRUE / FALSE	Specifies whether the ISR uses floating point registers which will need to be saved and restored during an OS context switch. The value is `TRUE` if a floating point context save is required and `FALSE` otherwise.
usedProcesses	list	List of ASCET processes called by the ISR.
category	1 / 2	The OSEK interrupt category. ASCET-SE V6.2 only uses Category 2 ISRs.
source	string	The symbolic name of the ISR as shown in the ASCET-SE V6.2 OS editor. Symbolic names use the same convention as RTA-OSEK.

Property	Type	Description
vectorAddress	string	The interrupt vector address. The address is target dependent and will be an absolute address for non-relocatable vector tables, or a vector location for relocatable vector tables. Addresses use the same convention as RTA-OSEK.
hook	MONITORING / NONE	The (non-OSEK) hooks used by the ISR.
minPeriodMicroSeconds	integer	The minimum inter-arrival time between two subsequent instances of this ISR in microseconds. This is ERCOS^{EK} specific.

Alarm

Property	Type	Description
name	string	Name of the alarm.
taskToActivate	string	The name of the task to be activated when the alarm expires.
autostart	TRUE / FALSE	Defines whether or not the alarm shall be autostarted.
autostartAppModes	list	List of application mode names in which the alarm shall be autostarted.
delay	integer	The number of ticks that must elapsed before the alarm expires for the first time.
period	integer	The period of the alarm in ticks.
delayMicroSeconds	integer	The value of the delay property in microseconds instead of ticks.
periodMicroSeconds	integer	The value of the period property in microseconds instead of ticks.

Resource

Property	Type	Description
name	string	Name of the resource.
property	STANDARD / LINKED / INTERNAL	The type of the resource. ASCET-SE generates only STANDARD resources.
ceilingPrio	TRUE / FALSE	The ceiling priority of this resource.

Message

Property	Type	Description
name	string	Name of the message.
CDATAType	string	C-type used for message definition.

UsedMessage

Property	Type	Description
name	string	Name of the message.
sentAccessor	string	Accessor name used by the task to send this message.
recvAccessor	string	Accessor name used by the task to receive this message.

Process

Property	Type	Description
name	string	Name of the process.
usedRessources	list	List of resource names used by the process.
usedFunctions	list	List of function names called from the process.
usedMessages	list	List of OSEK COM messages used by the process.
usesFPU	TRUE / FALSE	Specifies whether the process uses floating point registers which will need to be saved and restored during an OS context switch. The value is TRUE if a floating point context save is required and FALSE otherwise.

Function

Property	Type	Description
name	string	Name of the function.
usedRessources	list	List of resource names used by the function.
usedFunctions	list	List of function names called from this function (i.e. functions that are nested inside the current function).
usesFPU	TRUE / FALSE	Specifies whether the function uses floating point registers which will need to be saved and restored during an OS context switch. The value is `TRUE` if a floating point context save is required and `FALSE` otherwise.

8   Measurement and Calibration with ASAM-MCD-2MC

ASCET provides support for measurement and calibration by generating ASAM-MCD-2MC (A2L) files. Generated files rely on a set of statically defined configuration files that are supplied with ASCET. This chapter describes the content of the static files and then the generation of the ASAM-MCD-2MC data.

> **Note**
>
> *The alignment definitions in ASAM-MCD-2MC are determined automatically by ASCET-SE. The formerly necessary* `align.a2l` *file is obsolete.*

8.1   Project Definitions in ASAM-MCD-2MC (`prj_def.a2l` File)

The `MOD_PAR` section of the ASAM-MCD-2MC file (see ASAM-MCD-2MC specification) can be defined by the user in the `prj_def.a2l` configuration file, which is located in the directory of the ASCET-SE installation (`.\target\trg_<targetname>`). At delivery of ASCET-SE the file contents are as follows:

```
VERSION "000"
ADDR_EPK 0x0
EPK ""
SUPPLIER "xxx"
CUSTOMER "xxx"
CUSTOMER_NO "000"
USER "xxx"
PHONE_NO "000"
ECU "NO_ECU"
CPU_TYPE ""
```

Edit the file to suit your requirements.

8.2   Memory Layout in ASAM-MCD-2MC (`mem_lay.a2l` File)

The data file `mem_lay.a2l` is used to define the memory layout of the controller in ASAM-MCD-2MC format (i.e. `MEMORY_LAYOUT`, compare with the ASAM-MCD-2MC standard for syntax and semantics). Its content is inserted unchanged in the generated ASAM-MCD-2MC data file. This file is located in the target directory (`.\target\trg_<targetname>`); it modified to match the controller hardware and the memory layout defined in the locator invocation file.

> **Note**
>
> *This file is provided as an example only. You **must** edit the file and adjust it to your target system.*

8.3   ETK Driver Configuration in ASAM-MCD-2MC (`aml_template.a2l` and `if_data_template.a2l`)

The file `aml_template.a2l` contains type descriptions of global configuration BLOBs—e.g., `IF_DATA`, `TP_BLOB`—for the ETK.

The file `if_data_template.a2l` contains global configuration BLOBs for the ETK (TP and QP BLOB) in ASAM-MCD-2MC format.

Both files are located in the target directory (`.\target\trg_<target-name>`). The syntax is taken from the description of ASAM-MCD-2MC standards, the semantics from the documentation of the respective application system.

Both files are copied into the generated ASAM-MCD-2MC file. To generate a useful result, you must make sure that the `IF_DATA` configuration in the `if_data_template.a2l` file matches the type descriptions in `aml_template.a2l`. For that purpose, you can either update the content of the files in the target directory or replace the content with a reference (containing complete path and file name) to a suitable file stored elsewhere.

> **Note**
>
> *The files* `aml_template.a2l` *and* `if_data_template.a2l` *contain only examples. To adopt the description to your application hardware you* **have to** *edit or replace the file content.*

8.4   Generation of an ASAM-MCD-2MC Description File

ASCET-SE provides the possibility to generate project-specific ASAM-MCD-2MC description files that can be used for calibration using an appropriate calibration tool (e.g., INCA). For this purpose, a so-called *Virtual Address Table (VAT)* is generated by ASCET-SE on demand as a part of the project-specific C file.

To generate a Virtual Address Table:

To generate a Virtual Address Table as a prerequisite for ASAM-MCD-2MC generation, proceed as follows.

- In the project editor, click the **Project Properties** button.

  The "Project Properties" window opens.

- In the "Production Code" node, activate the **Generate Map File** option.

- Click **OK** to close the "Project Properties" window.

- In the project editor, select **Build** → **Build** or **Build** → **Rebuild All** to generate code including the VAT.

> **Note**
>
> *For the Additional Programmer use case, it is important to ensure that all code is consistent and free of VATs. To grant this, you can use the* `addressTable` *option in the* `codegen_*.ini` *file to override the* **Generate Map File** *option.*

The VAT consists of various C structures. It mainly contains the names of all quantities of the generated code that are part of the ASAM-MCD-2MC description as well as pointers to these quantities.

After compiling and linking a project containing a VAT, the resulting hex-file (`temp_vat.*`, the extension depends on target controller and compiler), as well as all other result files, contains all address information needed for ASAM-MCD-2MC generation.

By means of a special hex-file reader, this address information is extracted from the hex file. Additional information about element sizes, alignment, byte order, etc. is read from the Virtual Address Table as well. An intermediate file called `etas.map` is generated that contains the names and the memory addresses of all elements as `ASCII` text.

As the VAT is not intended to be part of the program running on the ECU, another hex file (`temp.*`) and the respective result files containing no VAT are linked.

To generate an ASAM-MCD-2MC file:

- In the project editor, select **Tools → ASAM-2MC → Write** to generate the ASAM-MCD-2MC file.

  The "Write ASAM-2MC To:" window opens.

- In that window, enter the desired file name and select the specific storage directory.

> **Note**
>
> *If the ASAM-MCD-2MC file is to be stored, be careful when placing in the directory `.\cgen\`. The files in this directory may be deleted upon exiting ASCET, depending on the settings in the Options window (see the ASCET online help).*

- Click **Save**.

  The ASAM-MCD-2MC file is saved to the selected directory, with the name you entered.

The diagram below shows the code generation process with and without ASAM-MCD-2MC generation.
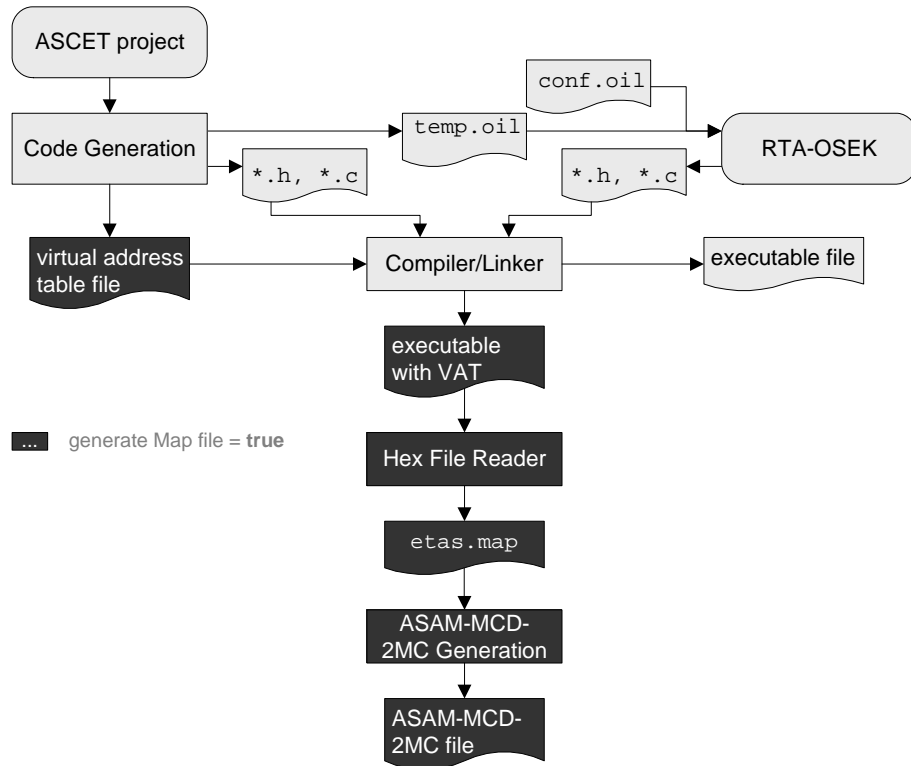
Fig. 8-1   Code generation with and without ASAM-MCD-2MC and VAT generation

You must ensure that the Virtual Address Table is mapped to a memory section that is **not** part of the ECU's physical memory. For details, please refer to section 3.3.5 "Memory Class Configuration". If the VAT is located in the ECU's physical memory then addresses in the ASAP2-MCD-2MC file may not be correct (and the mapped section of memory will be wasted).

Note

*In order not to waste ECU memory, it is recommended that the Virtual Address Table is located **outside** the physical ECU memory.*

8.5 Suppressing Exported Elements and Parameters

ASCET allows the generation of ASAP2-MCD-2MC information for elements and parameters whose scope is "Exported" to be suppressed. This allows you to provide the definitions of these elements outside of ASCET (for example, with 3rd party tooling). This is configured in the Project Properties.
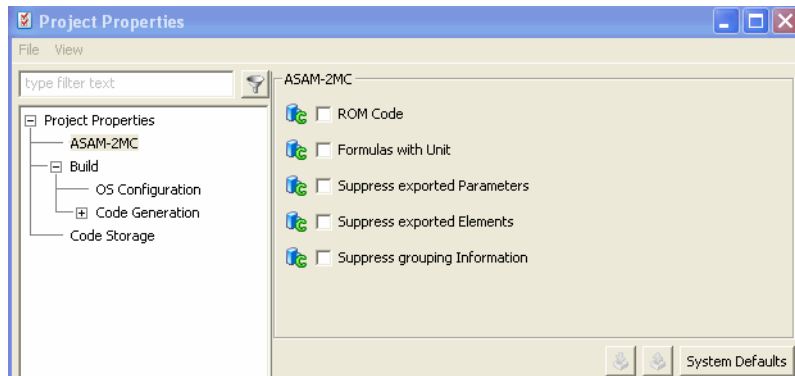
The behavior of suppression differs between ASCET objects (modules, classes and prototype classes) as shown in the following table. A plus (+) indicates that the element or parameter is generated in the the the A2L file. A minus (-) indicates that the element or parameter is **not** generated in the A2L file.

Suppress exported		Modules		Classes		Prototype Classes	
Parameters	Elements	Exported Elements	Exported Parameters	Exported Elements	Exported Parameters	Exported Elements	Exported Parameters
Not set	Not set	+	+	+	+	-	-
Not set	Set	+	+	-	+	-	-
Set	Not set	+	-	+	-	-	-
Set	Set	+	-	-	-	-	-

8.6   Working with SERAP

SERAP is a serial calibration concept that uses 2 pages of calibration parameters as follows:

1. A reference page that is located in non-writable memory (ROM, FLASH) as usual for parameters. This page holds the default values configured by the application

2. A working page that is located in writeable memory (RAM). This page is used at calibration time to modify values.

By switching back and forth between the reference and the working page, the impact of parameter modifications can be easily observed. This document assumes that you know how to use and deploy SERAP-based calibration in an application. Additional information on the correct use of SERAP is outside the scope of this document.

ASCET-SE can generate the data structures and code required to support SERAP calibration by:

1. Generating parameter data as two tables, one for the reference page and one for the working page. The tables have identical structure and values.

2. Modifying all parameter access in the generated code to include an indirection that allows selection of the reference or working page as appropriate at runtime.

Each ASCET method that needs to where to find its associated parameters within the table (conceptually this is the offset into the table). ASCET provides two alternative models of how methods find the offset that allow you to make a space/time trade-off when using SERAP.

1. *Embedded SERAP*: ASCET embeds (hence "embedded SERAP") the pointer to the SERAP data structure in the component data structure.

2. *Non-Embedded SERAP*: the offset is passed as a parameter to each method that needs parameter access

SERAP is enabled by setting the `serap` option in `codegen_ecco.ini` to true:

```
serap = true
```

Embedded SERAP is generated by default. To use non-embedded SERAP, the `serapEmbedded` option in `codegen_ecco.ini` must be set to false:

```
serapEmbedded = false
```

The `serapEmbedded` option has no effect unless the `serap` option is set to true.

Additional information about enabling SERAP functionality during the ASCET build process is provided in `proj_def.h`.

9 Integration with External Code

ASCET-SE provides powerful features that allow the combination of ASCET-generated code with external C code (either written by hand or generated by third-party tools). There are two main use cases:

- ASCET as an integration platform, supporting the complete make process from the model to the executable file and the ASAM-MCD-2MC description.
- The use of ASCET-generated code in an external make tool chain provided by the user.

This chapter describes the features that ASCET and ASCET-SE offer to support these use cases, in particular, the following features:

- User defined C- and H-files can easily be included in the ASCET make tool chain.
- Global declarations of functions, variables, and parameters provided outside ASCET can be easily accessed from the ASCET model. For this purpose, a special "prototype" model element has been introduced, comparable with a C function prototype.
- The optimizations concerning messages and method interfaces (signatures) can be configured by the user to ensure a stable interface for external code.
- Special header files are provided by the code generation that can be used as interfaces between ASCET and the user defined files.

The following sections describe some of the possibilities available.

9.1 Calling C Functions from an ASCET Model

ASCET offers different possibilities to call external functions from an ASCET model, which are described in this chapter.

9.1.1 Use of Prototypes

ASCET-SE provides a special interface to use C code functions, parameters and variables that are defined outside the ASCET environment (e. g. externally provided software). For this purpose, the ASCET implementation editor for classes provides the user the option to generate a "Prototype". Like a C function prototype, an ASCET prototype implementation provides the interface description for external C code. Similar to the use of service routines, this option can be set in the implementation editor of a class. See section 4.2.3 "Prototype Implementations" for details on the usage of the feature and the properties of the generated code.

Only extern declarations are generated for a class implemented as prototype. The code generated for a prototype contains neither variable and parameter definitions nor method definitions. The environment of the prototype element modeled in ASCET, however, refers to the prototype by means of extern declarations, wherever methods or global variables and parameters of the prototype are used. This way, it is the user's task to provide the global variables and parameters expected by the ASCET model in the external C code.

The following example shows how to call a function using a global variable from an ASCET model. Assume a file with the following content:

```
#include ".\include\a_std_type.h"

sint16 i;

void my_calc(void)
{
   i++;
}
```

To call the function `my_calc` from ASCET, the user can provide a class in the ASCET model that defines the global variable `i` and a method definition `my_calc`. The following example shows a possible implementation.
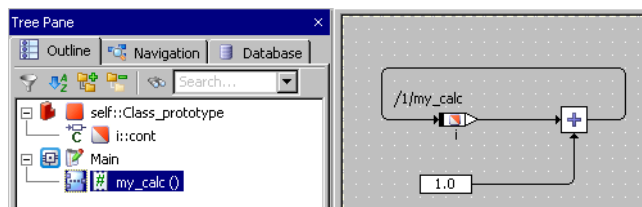
By setting the prototype flag in the implementation editor of the class, the user can specify that the actual content specified in the BDE shall *not* be used for code generation.
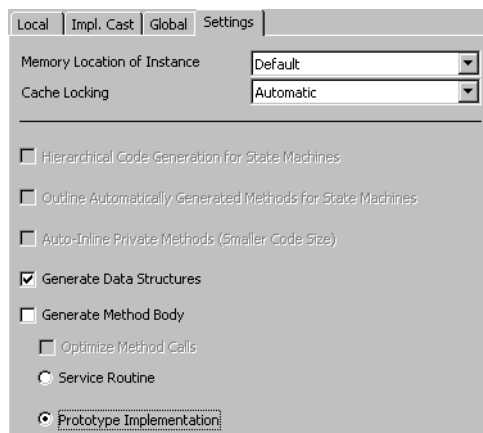
Instead, the code generated for the environment of the class in ASCET contains only the interfaces to the class, e. g.

```
#define _Class
#define _i i

#ifndef NO_DECLARE_i
extern sint16 i;
#endif

extern void CLASS_IMPL_my_calc (void);
```

```
   ...
   void MODULE_IMPL_process (void)
   {
      CLASS_IMPL_my_calc ();
   }
```

As the example shows, the names of the "prototype" methods are still generated according to the ASCET naming convention (e.g., `<Class>_<Impl>_<Methodname>`, see "Data Structures and Initialization for Complex (User-Defined) Objects" on page 172). To adapt the interfaces of the external code and the ASCET-generated code, an include file named `proj_def.h` is provided in the target directory of the ASCET-SE installation. This file is included in the ASCET generated code by default and offers the user the possibility to map the ASCET names to external code names using preprocessor directives ("`#define`"). In the example, the following adaptation of `proj_def.h` is suitable:

```
   #define CLASS_IMPL_my_calc() my_calc()
```

For prototypes, the extern declarations of global variables and parameters are enclosed by `#ifndef` preprocessor directives (see code example above). This allows you to provide your own `extern` declarations if required by `#define NO_DECLARE_<variablename>`.

For example, assume that the ASCET variable `i` needs to be mapped to your externally declared variable `i_usr`. The respective extern declaration could look as follows:

```
   #define NO_DECLARE_i
   #define i i_usr
   extern uint16 i_usr;
```

Again, this code can be provided in `proj_def.h`.

> **Note**
>
> **Warning:** *all of these changes modify ASCET code generation. You must provide adequate macro definitions for elements and methods or own declarations for exported elements. You assume full responsibility of the consequences for your external code as well as for the correct inter-operation with ASCET-generated code. Problems may arise with respect to the ASAM-MCD-2MC generation (see below) and similar. Note that the interfaces to ASCET-generated code may be changed in future product versions.*

ASCET does not generate A2L file entries for exported parameters or exported elements of prototype classes. If entries are required, then you must provide them externally and merge them with ASCET-generated A2L files outside of the ASCET development process.

9.1.2 Invocation by C Code Specified in ASCET

As well known from previous versions, of course ASCET V6.2 also offers the possibility to specify C code in internal or external editors. C functions specified outside ASCET can be called by this code using `extern` declarations.

9.1.3   Including C Source Files in the ASCET Make Process

To include C source files in the make process controlled by ASCET, ASCET-SE allows the definition of a list of file names in `project_settings.mk`. In addition, a list of path names can be defined to specify where ASCET-SE searches for the defined files.

See section 5.6 "Customizing the Build Process" for further details.

9.2   Calling ASCET-Generated Functions from External C Code

ASCET generates a `function_declarations.h` file, containing extern declarations of all functions of the ASCET model. This file can be included in the user software to easily access ASCET-defined methods or processes in external code.

For classes implemented as prototypes, these extern declarations can be disabled by means of the preprocessor switch. The switch is named `DECLARE_PROTOTYPE_METHODS`, as the following example (extract from `function_declarations.h`) shows:

```
#ifdef DECLARE_PROTOTYPE_METHODS
extern void CLASS_IMPL_my_calc (void);
#endif
```

9.3   Using External Global Variables/Parameters in ASCET Code

As described in section 9.1.1, global variables and parameters can be defined in external C code and accessed by ASCET-SE generated code model by means of a prototype implementation. The `proj_def.h` file, which is provided by the installation in the target-specific directory, can be used to map the external code name space to ASCET's symbolic names by means of preprocessor directives ("#define").

In addition, ASCET generates a `variable_declarations.h` file, containing extern declarations of all global variables of the ASCET model. This file can be included in the user software to easily access ASCET model elements from the external code.

For classes implemented as prototypes, the extern declarations are configurable by means of special preprocessor directives, as the subsequent example shows:

```
#ifdef DECLARE_PROTOTYPE_ELEMENTS
#ifndef NO_DECLARE_i
extern sint16 i;
/* min=-32768.0, max=32767.0, ident, limit=yes */
#endif
#endif
```

The switch `DECLARE_PROTOTYPE_ELEMENTS` can be used to globally disable the extern declarations of all prototype elements in the file `variable_declarations.h`. Individual switches are provided for single variables and parameters exported by prototypes, as described in chapter 9.1.1 "Use of Prototypes".

9.4   Generating Code for Use with External Data Structures

By default, ASCET-SE generates all data structures it needs so that a project is always internally consistent. However, if you have many projects that use the same logical model and differ only in the data values used, then it is desirable to generate the code in ASCET and supply the data sources externally (usually with a 3rd party tool).

Such a workflow can offer processes benefits, for example the code can be verified once and re-used without the risk of it being "touched" with each minor data change.

ASCET-SE provides support for this workflow by allowing the generation of ASCET data structures to be disabled.

> **Note**
>
> *It is expected that user's working with externally generated data structures are also building their systems outside of ASCET (i.e. you are not using ASCET as an integration platform).*

Data structure generation is configured in the "Project Properties" window, "Production Code" node. Three mode of operation are available:

1. Generate for every component.
2. Generate for no components.
3. Use component settings. By default, components are configured for data structure generation. Component settings are overridden by the other two options. This mode allows you to generate some data structures using ASCET and provide other by external code.

Fig. 9-1 shows a configuration where data structure generation has been disabled for all components.
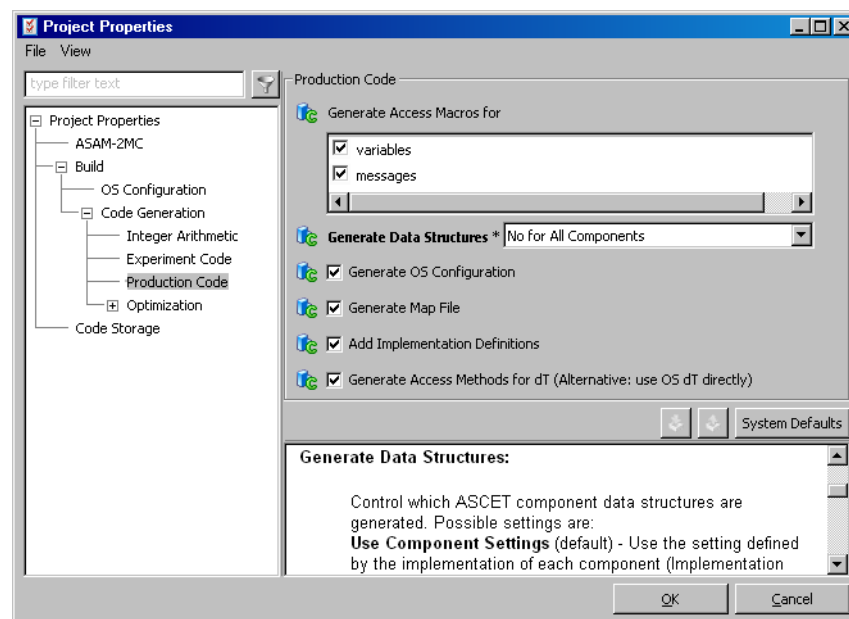
Fig. 9-1   Disabling data structure generation for all components

For the `Use Component Settings` mode, each component implementation can specify whether or not data structures are generate as shown in Fig. 9-2.
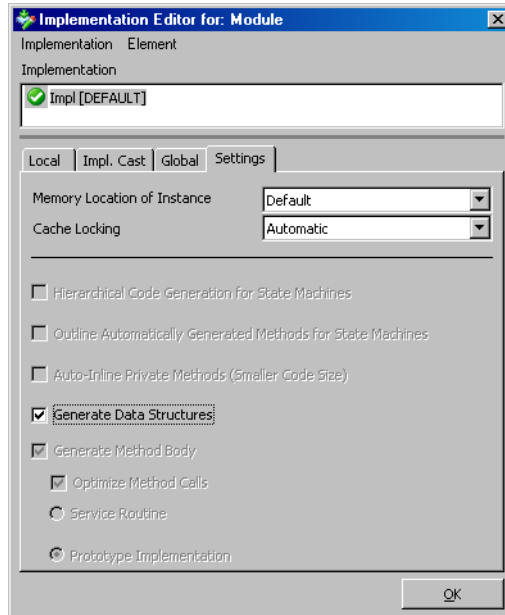
Fig. 9-2   Selecting data structure generation on a per component basis

9.5   Configuring the ASCET Optimization Features

When using ASCET with external code it is important that the interface remains stable. ASCET's default optimization strategies are designed to produce the smallest and fastest code and, consequently, may result in changes to the external interface when changes are made to the model.

The default optimizations that can have this side-effect can be deactivated to guarantee a stable interface.

9.5.1   Configuring Method Calls

For methods of classes which can be multiply instantiated, ASCET passes a pointer to the instance's data structures as the first element of the method argument list. This is called the *self*-pointer in ASCET (and is analogous to the self point in C++) (see Section 13.3.3).

For methods of classes that are only instantiated once, this pointer is not needed as there is only one data instance and that can be accessed directly without ambiguity. Optimizing away the self pointer increases the run-time performance and reduces the stack space requirements on ASCET-SE generated code. This optimization is done by default during code generation.

However, combining ASCET-generated code with external C code requires a software interface that is widely invariant to changes of the ASCET model. The optimization of single instance classes can therefore be switched off to avoid

unexpected changes of calling conventions for methods due to model modifications. The single method optimization can be deactivated in the "Settings" tab of the class implementation editor.
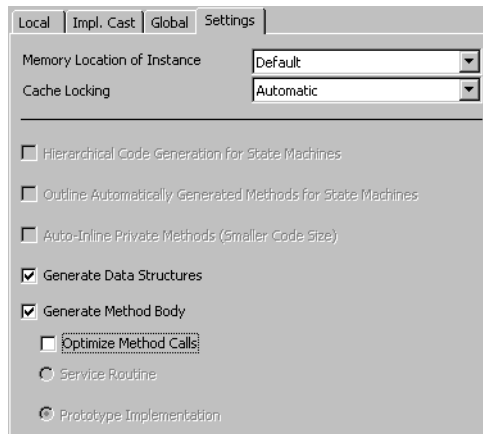
In this case, the self pointer will always be generated, no matter if the class is multiply instantiated or not.

> **Note**
>
> *When calling ASCET-generated methods or using ASCET-generated variable and parameter definitions from external C code, you must observe the data type definitions generated by ASCET carefully. It is not recommended to use types other than those generated by ASCET. This is especially true for the self-pointer.*
> *The function interfaces provided by the ASCET- generated code might change in successor versions of the tool.*

If you are certain that a class will only be single instantiated in a model, then generation of a method interface without the self-pointer can be re-enabled by re-activating the **Optimize method calls** option.

9.5.2   Configuring Message Copies

ASCET uses the configured OS task types and priorities to generate message copies only where needed to ensure data consistency (see section 13.4.3 on page 179). However, this optimization relies on ASCET knowing about **all** data accesses at code generation time.

ASCET cannot know about any data access of scheduling issues that are defined outside of the ASCET model. To prevent data consistency problems when using external OS configuration or external C code, ASCET-SE allows the generation and the use of message copies to be defined. See section 13.4.3 "Messages" for details.

9.6   Working with Variant Parameters

When parameters are configured in ASCET, it is possible to set the **Variants** attribute in the Properties editor of a parameter to control whether access to multiple variants of the parameter is available.

When the option is enabled, ASCET assumes that all parameters with the variant attribute set are grouped into a single memory section. This set of parameters defines a "variant". Furthermore, ASCET assumes that multiple sets of parameters, each set representing a specific variant, exist and generates code to access parameters using an indirection (through an externally defined offset).

This feature is EXPERIMENTAL in ASCET. Please contact ETAS for further details on its use.

10   Modeling Hints

This section provides some general guidelines for structuring models and specifying implementations with an emphasis on efficient and numerically correct implementation code.

The requirements to the model are often contradictory. An optimization of the memory requirement can be achieved at the expense of execution time and accuracy. If execution time is optimized, increased memory requirement and a worse readability of the code may be the consequences. Finally, high accuracy is connected with increased memory requirement as well.

10.1   Implementations

The different requirements have to be considered during implementation. The implementation of single entities thus depends on

- the physically possible value range,
- the required accuracy,
- the properties of hardware and sensors.

10.1.1   Definition of Conversion Formulas

Offset: Conversion formulas should have an offset of zero. A nonzero offset has little advantage, and results in additional code for mathematical operations. Possible exceptions include:

- Entities which already have an offset represented in the system, e.g., results from sensors.
- Arrays, matrices, distributions, or characteristic curves and maps, where a more compact representation (i.e. with smaller word length) is enabled with an offset, to save memory space.

For example: Assume a temperature from -50 to +150° C and a resolution of 1° C. Without an offset, a word length of 16 bits is required; with an offset, 8 bits suffice. One byte per quantity (e.g., an array element) is saved. Here, one should weigh between memory requirements and run-time/code overhead.

Usually, using an offset for a single value to save memory space is not justified.

Scale values: The approximate range of a scale value depends on the physics of the overall system. Such numerical requirements must be determined theoretically or experimentally. However, within the given order of magnitude, one has many possibilities when choosing the actual scale value.

- Scale values should be simple, rational numbers. For example, fractions should have simple coefficients that are small numbers, powers of two or ten, and not larger prime numbers, e.g., 8/3, 256/100, 50. In general, fractions (e.g., 3/16) should be preferred over decimals (e.g., 0.1875) when entering a scale value. The following rules should be observed:
- Scale values of the form $2^K/n$ are best suitable for unsigned results, and $2^{K-1}/n$ for signed results. K is the corresponding word length in bits, and n is a suitable number slightly greater than the maximum representable value. This assures usage of nearly the entire value range.
- Simple coefficients should have priority over using the entire available range of values.

Example: The given range of values is `[0,9.1]`. To implement in `8` bits, a simple scale value of $2^8/10 = 25.6$ should be used. The resulting quantization and interval are `0.039` and `[0,9.96]`, respectively.

If, in this example, the aim would be the highest possible precision (for 8 bits), the scale value would be `255/9.1=28.02=2550/91`. This has only an insignificantly higher resolution of `0.036`, and hence no visible numerical improvement in the control algorithm. On the other hand, considerable run-time and *loss* of precision are probable if users must convert between this complex scale value and a different one in the generated code. For instance, if a conversion of this unfavorable scale value into the above-mentioned simple scale value is necessary, the unfavorable rational rescaling factor `(256/10)/(5824/6375)=2550/91` emerges, which causes numerical inaccuracies and requires a 32-bit intermediate result.

To view a formula:

- In the project editor, you can view the conversion formulas by clicking on the "Formulas" tab.
- View a formula by double-clicking on it.

The advantage of using scale values that are a power of two has already been demonstrated in several examples. Re-scale operations are simply reduced to bit shifts. Therefore, these should be used whenever possible.

10.1.2   Definition of the Value Intervals

When specifying value intervals, their use by the code generator to transform mathematical expressions must be considered. Thus, two goals are important when creating the value interval:

- Avoid overflow protection (i.e. right shifts) which results in the unnecessary loss of numerical precision.
- Avoid clippings which result in additional overhead in the code.

Hence, only the range of values that are physically relevant should be selected for an implementation.

Example:

`{ A ∈ [0.. 40] } + {B ∈ [ 0,10 ]} = C`

If the same value interval is chosen for A and for B, i.e. A_{phys}`[0,40]`, B_{phys}`[0,40]`, a scale `S = 0.25` for all quantities will result in the following implementations:

`A`$_{uint8}$`[0, 160], B`$_{uint8}$`[0, 160], C`$_{uint16}$` [0, 320]`

The result uses the double bit length as the two addends.

If, however, the interval B_{phys}`[0, 10]` is chosen, the same bit length is sufficient for all three quantities:

`A`$_{uint8}$`[0, 160], B`$_{uint8}$`[0, 40], C`$_{uint8}$` [0, 200]`

Therefore, the common practise of using the default value range for a given implementation type (e.g. `[-128, 127]` for `int8`), is never recommended, especially if this default exceeds the relevant range by a factor of 2 or more.

Example definition of the formula and interval for a throttle position measurement:

- Regard the following example.

  The throttle position measurement is converted from voltage to degrees using a characteristic curve.

  For the Interpolation node distribution ("X Distribution" tab), the implementation editor of `Meas_v2deg` looks as follows:
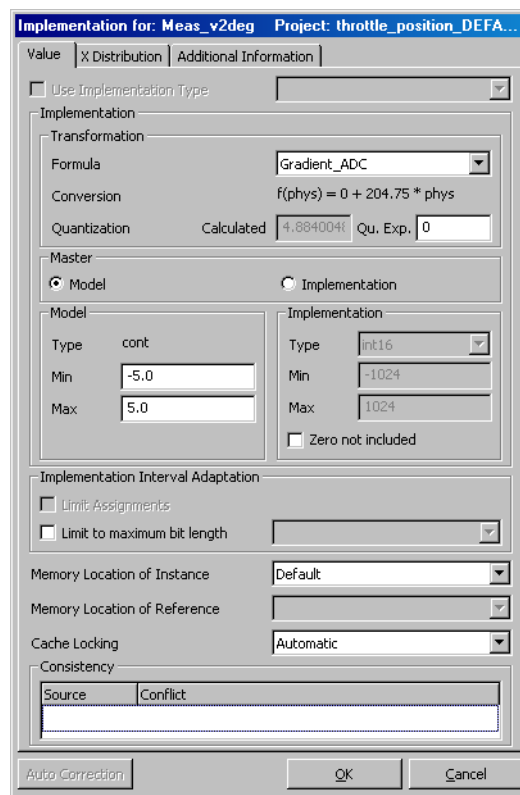
Here, the throttle position measurement is the difference of two signals that are both 0 – 5 volts. Each signal is converted using a 10-bit A/D converter. As a result, the finest resolution of this signal is $5\ V/2^{10}\ bits$, giving the scale value of `1024/5`. The interval, `[-5,5]`, results from the subtraction of the two signals.

10.1.3   Defining Implementations for Related Variables

Conversion formulas and implementation types for variables (or method arguments) which are assigned to each other or connected mathematically should, if possible, be chosen to match each other. The following are examples of this concept:

- Choose offset 0 if possible.

- For *addition* or *subtraction*, variables should be assigned the same, or at least similar, scale values.
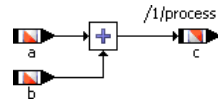
$$S_a = S_b = S_c$$
(S: scale factor)

Scale values are called "similar" if their quotient is a power of two, a small integer number, or a simple fraction. The first case is preferred for efficiency, whereas the simple fraction is the least favorable solution.

- For *multiplications* and *divisions* the scale should ideally be the product or the quotient of the operands, respectively. The result type has to be extended if necessary.
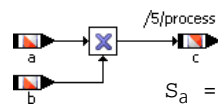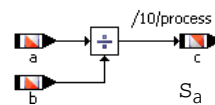
$$S_a = S_b * S_c \qquad S_a = S_b / S_c$$

- For more *complex classes*, the following scales are recommended:
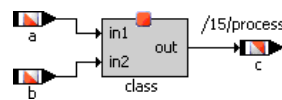
$$S_a = S_{in1}; \quad S_b = S_{in2}; \quad S_c = S_{out}$$
$$S_{out} = f(S_{in1}, S_{in2}, S_{p\_int})$$
(p_int: internal quantities)

The input arguments and the quantities assigned to them have the same scales, as well as the return value and the value it is assigned to. The scale of the return value depends on the scales of the arguments and the internal elements of the class.

- *Assignments*:
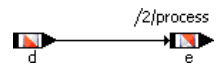
  – Re-scaling should be avoided in the model, as it involves additional multiplications and divisions. These result in additional run time and memory consumption.
    For the generated code in the above example, e.g., the generated code for different scales shows the following differences:

$$S_d = S_e \qquad \Rightarrow (e = d);$$
$$S_d = 1/5, \ S_e = 1/3 \qquad \Rightarrow (e = ((d*3)/5));$$

  – Quantizations with a fix base (mantissa) allow re-scaling by means of one single multiplication or division.

$$S_d = 10^{-1}, \ S_e = 10^{-2} \qquad \Rightarrow (e = (d*10));$$

  – Quantizations with a base of 2 allow re-scaling by shifts:

$$S_d = 2^{-2}, \ S_e = 2^{-1} \qquad \Rightarrow (e = (d>>1));$$

- By using dependent parameters,
  - re-scaling can be avoided, e. g. for comparators or concatenated cal-
    culations with parameters;

$S_{in1}=1/10$ $S_{in1}=1/10$

$S_{Parameter}=1/16$ $S_{Parameter\_dep}=1/10$
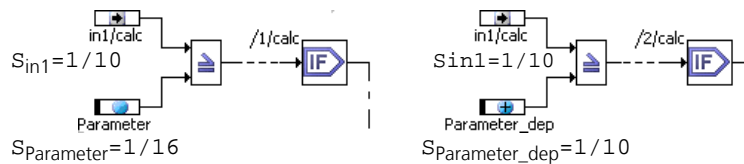
  - "odd scale factors" can be cancelled, e. g. when converting different
    units;
  - run time and code can be optimized. By using *virtual* parameters,
    memory can be saved.

  Disadvantageous is, however, the use of an additional parameter.

- Internal intermediate memories in which results are accumulated (i.e., in
  integrators, filters, low-passes, etc.) should be represented with at least
  twice the word size of the accumulated result to assure precision.

10.1.4 Multiplication of Large Results

If two quantities with large intervals are multiplied, numerical precision may be
lost. This happens when the code generator avoids a possible overflow via right
shifts.

Example 1: Compute `X*Y`, where `X` and `Y` both have implementation type
`uint32` and use the full 32-bit range. To avoid overflows, the following code is
generated:

```
(X>>16)*(Y>>16)
```

This may be numerically inaccurate; if, e.g., `X` or `Y<65536`, the result is 0.

The problem is particularly critical when several multiplication operations are exe-
cuted in a sequence.

Example 2: Consider an integrator that computes `X*K*DT`, where `X` (input), `K`
(integration constant) and `DT` (time difference) have type `uint16` and use the
full 16-bit range. Assuming the intermediate result is stored in a 32-bit memory,
a total of 16 right shifts are needed. This leads, e.g., to the following:

```
((X>>5)*(K>>5))*(DT>>6)
```

However, a small value for any of the three variables will yield zero, causing the
integrator to stay at zero. This is entirely a result of the automatic overflow pro-
tection.

> **Note**
>
> *Of course these effects are not special problems caused by the code generator,
> but common problems occurring with quantized arithmetic with limited word
> size. The effects occur in the same way for manual coding.*

To avoid such problems, the following rules should be adhered to during the
modeling stage:

- Do not represent operands for multiplication more precisely than
  required, i.e. with smallest possible word size.

- Reduce the operand's value range to that which is physically relevant only. For example, the time difference, `DT`, in the integrator can be represented in 16 bits with a quantization of 10 µs. This gives a range to 655 ms, which should suffice for a typical vehicle application.

- If several multiplication operations must be performed in sequence, the quantizations and the interval have to be carefully selected using the above criteria. This portion of the model should be tested in detail. If floating point arithmetic is possible for the target, it should be considered.

- For integrators, low-pass and similar filters, expressions of the following type occur:

```
in * k * dT
```

  If this computation runs in a static time frame, the variable `dT` should be replaced with a fixed value which is included (with the aid of the conversion formula) in the constant `k`, i.e.

```
in * (k * dTfix) = in * kfix
```

  In doing so, the multiplication sequence and the possible inaccuracy arising from the sequence are avoided.

To study the effects of dT in the PID derivative term calculation:

- Look at the derivative term calculation in the `PIDT1` controller (see section 12.3.9 on page 149).

  To study the effects of `dT`, we will focus on the calculation of `Temp2`.

The calculation of `Temp2` consists of `dT*t3`, where `t3` is the expression assigned to `D_term` also discussed in "To optimize the derivative term calculation:" on page 151. The implementations are `dT=2`^{14}`*dt` $\in[0,0.1]$, and for the intermediate result `t3`, a scale of 2^{13} and interval `[-42000,42000]` (see example on page 151).

- The multiplication `dT*t3` results in an overflow of 9 bits (i.e., 7 right-shifts for `t3` and 2 for `dT`).

- Since this calculation occurs in a static time frame, `dT` can be represented with a literal or parameter. With a parameter, a much smaller interval can be specified to reduce the overflow.

  - Replace `dT` with the parameter `delta` as shown below. Assign to it a value of `0.001`, a scale value of 2^{14}, and an interval of `[0,0.001]`.
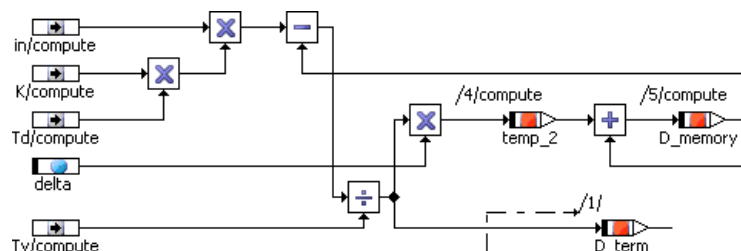
  - Generate new code for the example and examine the changes.

Because of the smaller interval, `dT*t3` results in an overflow of only 2 bits, even though the time step is represented with the same precision. Using a literal in this case also produces better results than using `dT`, but not as good as those obtained when using a parameter. The reason comes from the accuracy criteria for literals (see section 12.3.7 on page 148). This criterion produces the representation of a literal with a relative error of less than 0.1%. For `0.001`, this requires a scale value of 2^{17}, and therefore an overflow of 5 bits occurs.

10.2   Model Structure

This section contains considerations of the optimal design of ASCET models with respect to efficient code generation.

10.2.1   Division

Division leads to many numerical problems which have already been described elsewhere, and should be avoided, if possible. This can be achieved by, e.g.,

- introducing dependent parameters with the reciprocal value,
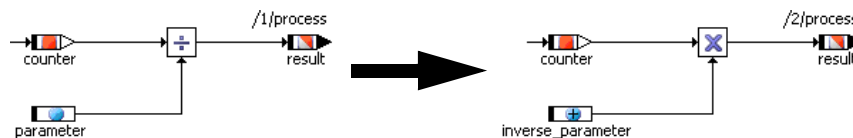
- temporarily storing the result of a division and reusing it.
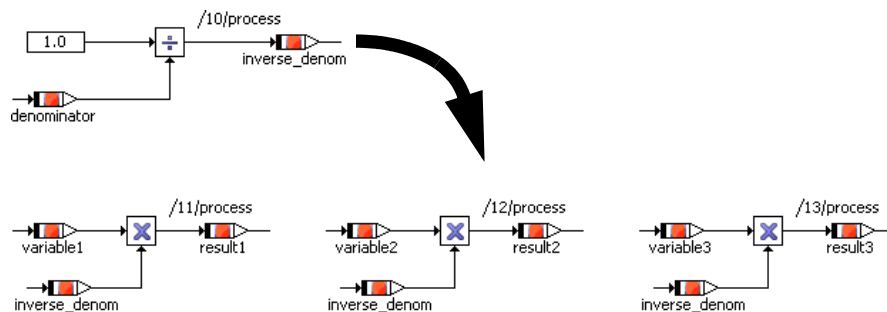
The following rules concerning division should be adhered to:

- Divisions within mathematical expressions should be performed as late as possible.

- In integer representation, the numerator should always be considerably larger than the denominator (double word size if possible).

- The denominator should not use the highest valid word size. For example, if a word length of 32 bits is valid, the denominator should have no more than 16 bits.

- The denominator interval must be restricted from 0.

10.2.2   Multiple Calculations, Concatenated Calculations, Logical Operators

Multiple Calculations

Multiple calculations like the ones shown below should be avoided, where possible. They require additional runtime and can cause wrong results, e.g. when used in timers or integrators.
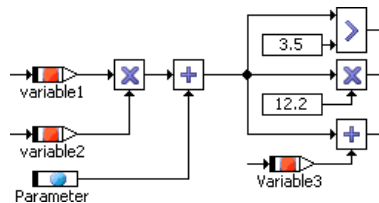
There are several possibilities to avoid multiple calculations:

1. By inserting temporary variables.
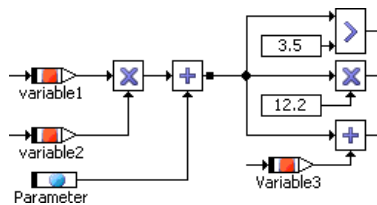
> On the one hand, this realization allows quick access to the intermediate result without additional memory consumption. On the other hand, the temporary variable can neither be implemented nor measured with a calibration system. It cannot be used in another context and the sequencing cannot be influenced. Stack management becomes more expensive.

2. By inserting process-/method-local variables.
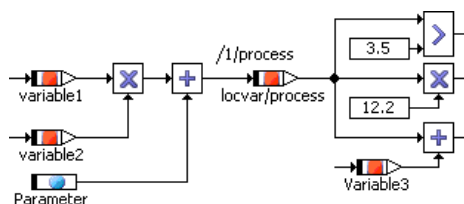
> This way the intermediate result can be accessed quickly. The method-/process-local variable can be implemented and multiply used in different contexts, and the sequencing can be specified. Like the temporary variable, the method-local variable can neither be measured nor be assigned a memory class. Additional expenses for stack management are necessary.

3. By inserting variables.
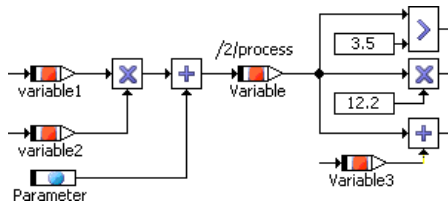
A variable can be implemented and measured. It has a unique memory location in the ECU and can thus be assigned a memory class. It can be multiply used, and is simultaneously available in different methods or processes. The sequencing information can be explicitly specified. On the other hand, introducing a variable causes additional permanent use of RAM.

4. If a send message is used as an intermediate result, it can be changed to a send&receive message.
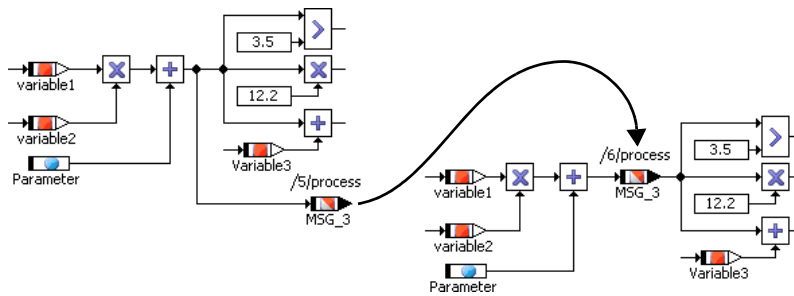
This does not cause additional RAM consumption. Only the RAM amount for the already existing message is needed. The element can be implemented and measured, it has a unique address in the ECU and can be assigned a memory class. It is simultaneously available in different processes. However, this approach is restricted to a limited number of cases, the more so since the sequencing has to be kept in mind for the whole model.

Concatenated Calculations

Intermediate variables (method-/process-local variables) should be inserted into long concatenated calculations. Otherwise, the overflow handling (i.e. right shifts) for the temporary intermediate results generated by the code generation can cause a loss of precision.
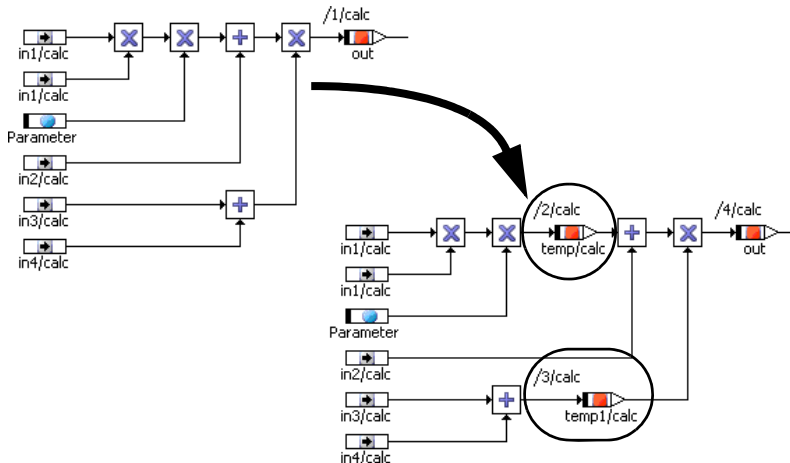
Introducing intermediate variables allows the specification of the desired precision for partial results.

Logical Operators

The code generator maps the inputs of a logical operator in descending order to a catenation from the left to the right.


```
results_log =
    ((Express_1)&&(Express_2)&&(Express_3))
```
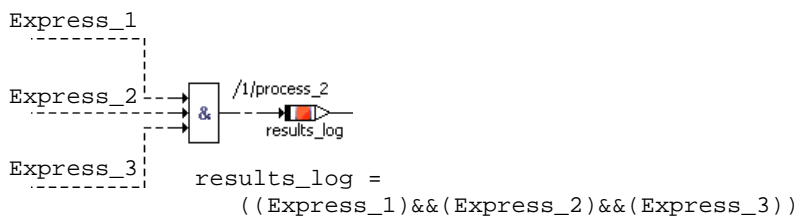
During runtime, the code is processed from left to the right as well; if the result can be determined before the calculation is complete (e.g. `Express_1 = false`), the evaluation is interrupted. It is thus recommended to arrange the inputs of logical operators top down in the order of calculation time and propability. For the AND-operator,

- expressions with short calculation time,
- unlikely expressions;

for the OR-operator,

- expressions with short calculation time,
- likely expressions

are specially recommended for the upper inputs of the operator.

Example:
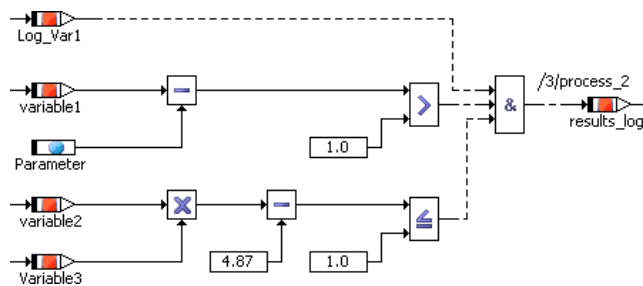
10.2.3   Classes and Modules

When using classes, keep the following in mind:

- A dead beat response ($z-1$) can be replaced by a single variable (mind the sequencing!).

- Unnecessary nesting of classes causes nested function calls and additional consumption of stack and run time. It should be avoided.

- If multiple instances of a class are used, all instances use the same program code, but each instance has its own data sets. This saves code space (ROM) but requires an extra indirection for each data element access.

- Classes should be decoupled, i.e. the return value should be separated from the calculation by means of separate return methods or direct access. Direct access methods should be preferred.

  Where applicable, the code generation option *optimize direct access methods* (a description is given in the ASCET online help) can be activated. Thus, no special function call is necessary for return.

  With this approach, the class is calculated only once, even if the return value is used several times; this means runtime saving. The calculation of the internal algorithms and the return values do not have to take place at the same rate. Both the old and the new return value can be accessed. The downside is the use of an additional variable, which is needed as intermediate memory for the results of the calculation.

- When inlining of methods is used, the method program code is written directly into the module program code by the compiler; no function call is needed. Runtime is optimized thereby, but additional memory is required when the method is used more than once.

- ASCET creates separate program code for each implementation of a class. If an implementation is used repeatedly, the memory requirement is reduced; however, the usability of this approach is restricted.

When using methods in modules, keep the following in mind:

- You can access messages and resources in a method in a module. However, only the message optimizations _OPT_COPY and _NO_COPY are supported during code generation for messages in modules. If you use another variant (_NON_OPT_COPY, _OSEK_COM, or _OSEK_COM_STACK_BUFFER), code generation produces an error message.

- If a method in a module uses a message, this method may be called from one task only; a static assignment is required between task priority and the place in the code where the message is accessed.
Calls from other tasks are forbidden; they produce an error message.

10.2.4   State Machines

You can optimize a state machine under three aspects:

- Response time
- Runtime
- Code size

The various optimization options are described in detail in the ASCET online help.

11   Migrating an Existing Project to a New Target

ASCET-SE allows a project that was originally developed for one target to be migrated to a new target by copying the C code and OS settings from the old target, experiment type or implementation to the new target.

To copy the C code for an entire project:

To copy the C code for all classes and modules of a project from another target, experiment type, or implementation, proceed as follows.

- In the project editor, select the appropriate target and code generation options for your controller.

- Select **Extras → Copy C-Code From.**

  The "Selection Required" window opens.
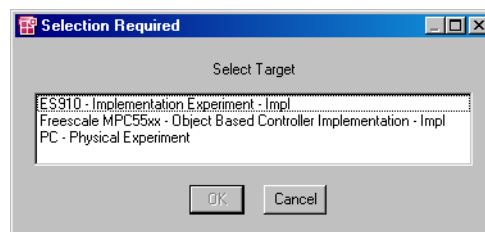
- Select the target you want to copy the code from, and click **OK**.

To copy C code for single classes or modules:

To copy existing C code of a single class or module to another target, experiment type, or implementation, use one of the two possibilities described here.

1. Use the menu item **Tools → Code Variants → Copy To**.

- Open the module/class in the C code editor.

- In the "Target" combo box, select the target the C code was written for.

- In the "Arithmetic" combo box, select the experiment type the C code was written for.

- In the "Implementation" combo box, select the implementation the C code was written for.

• Select **Tools → Code Variants → Copy To**.

The "Copy C-Code To:" selection window opens.
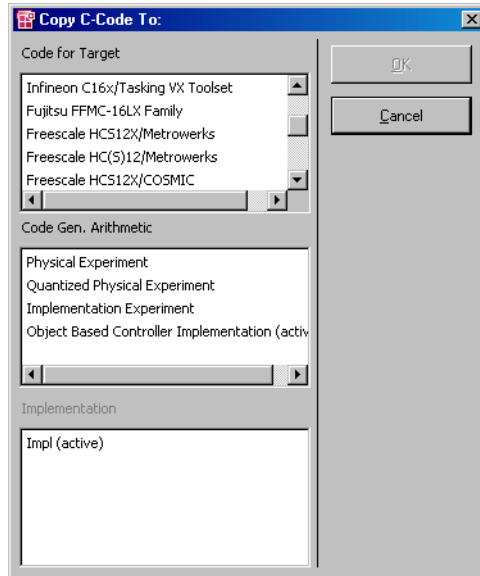
• In the "Code for Target" field, select the target you are using.

• In the "Code Gen. Arithmetic" field, select the appropriate experiment type.

• In the "Implementation" field, select the desired implementation.

Once you have completed the selection, the **OK** button is activated.

• Click **OK** to close the window.

2. Use the menu item **Tools → Code Variants → Copy From**.

• In the C code editor, use the "Target", "Arith-metic", and "Implementation" combo boxes to set up the target you want to use with the appropriate experiment type and implementation.

• Now select **Tools → Code Variants → Copy From**.

The "Selection Required" window opens.
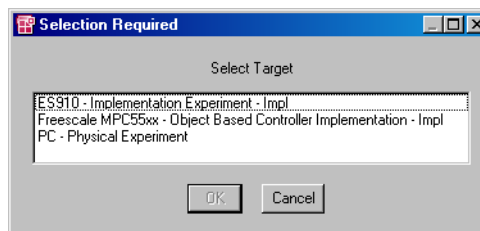
- Choose the target, experiment type, and implementation you want to copy the code from, and click **OK**.

To copy the operating system settings:

- In the project editor, select the "OS" tab.
- Select **Operating System → Copy From Target**.
- In the "Selection Required" window, choose the target you want to copy the OS configuration from.
- Click **OK** to close the window.

  The operating system settings are copied to the current target.

Further possibilities of target-specific adaptation of code generation are provided in chapter 5 "Configuring ASCET for Code Generation".

12   Understanding Quantized Arithmetic

This chapter provides a detailed description of how the code generator produces code for algorithms specified in ASCET. The rules of this transformation are described in more detail in later sections. Examples are used to illustrate how the base operations are first transformed and how the mathematical expressions are then optimized using the implementation specifications. One section is devoted to an overview of the numerical aspects of integer arithmetic.

The most essential task of implementation code generation is the automatic transformation of the arithmetic in the physical model into the quantized arithmetic for the target implementation. Necessary conversions and correction factors are generated and overflows avoided or corrected automatically. In the traditional manual coding process, this step has proven to be unreliable. Thus, a reliable automatic generation improves software quality.

The generated integer arithmetic could further be optimized.

Logical (Boolean) operations, control structures, and method calls are converted the same way in both the implementation and physical code generations. The main difference between the two is that implementation code generation produces integer arithmetic, while physical generation does not.

The main goal of the implementation code generation is the semantically correct transformation of the physical specification while considering the implementations given by the user. Numerical errors are inevitable due to quantization and integer division. However, these errors are minimized. The generated code is robust, e.g. no overflows occur at run-time.

12.1   Degrees of Freedom and Optimization

The variable/parameter implementations defined by the user are mandatory for the code generator. However, even in a mathematical expression containing several of these "fixed" implementations, there usually exist some degrees of freedom. The degrees of freedom are the choices of implementations for intermediate results. These can be defined by the code generator. However, restrictions for the target must be taken into account, particularly the maximum available bit length for integer quantities.

The degrees of freedom are used by the code generator for optimization based on the following criteria:

- Minimizing numerical errors.

- Avoiding or correcting overflows.

- Minimizing run-time and memory requirement, i.e. code size, RAM and stack space.

These optimization goals partially contradict each other. A complete optimization program cannot be created with acceptable overhead. The code generator, therefore, uses a heuristic procedure that has two essential components:

- Local rules for good transformation of the individual base operations.

- Global control strategy with which the local transformations are coordinated for more optimal mathematical expressions.

This procedure may produce unsatisfactory results in individual cases. In these cases, the user must intervene manually and reduce the degrees of freedom allowed to the generator. This is done by introducing temporary variables with defined implementations at strategic points in the mathematical expressions.

Further potential for optimization exists by selecting special fixed point code generation options (see the description of the "Integer Arithmetic" node in the ASCET online help).

12.2 Numerical Aspects of Integer Arithmetic

When physical arithmetic is transformed to integer arithmetic, numerical errors arise. Two different sources for these errors exist: quantization and integer division.

12.2.1 Quantization Errors

When a real quantity is mapped to a quantized representation, an error arises which is, at most, half the quantization.

This representation error cannot be avoided. It can, theoretically, be made arbitrarily small by choosing a finer quantization. However, the smaller the quantization chosen, the larger the corresponding integer results become. Of course, in practice only a restricted range of values (i.e., 32-bit numbers) is available for the quantized representations of both the quantities and the computations performed on them (i.e. the intermediate results).

Therefore, the achievable precision depends on the selection of those quantized representations (i.e. value range and quantization). While choosing the quantizations, a compromise must be found between numerical precision and memory space requirements. In addition, available word sizes for the target must be taken into account.

12.2.2 Errors from Integer Division

In integer arithmetic, addition, subtraction and multiplication are, in principle, calculated exactly – provided no overflows occur. But for integer division, errors occur because the fractional remainder is truncated. For example, 2/3 equals 0 and 9/5 equals 1. Principally, the result could be rounded-up, thus reducing the error (max. by half). Division results in particularly unfavorable behavior with respect to error propagation.

As to not impair the numerical precision unnecessarily, obey the following rules when using integer division:

- Completely avoid division if possible.
- The numerator should be noticeably larger than the denominator, e.g., 32 bits/16 bits. Numerators should be typically twice the word size and use these additional bits.
- In mathematical chain operations, perform the division as late as possible. For example, $(x*y)/z$ usually allows higher precision than $(x/z)*y$ provided that $x*y$ may be calculated without overflow.

12.2.3   Error Propagation

Quantization and division errors will be propagated through mathematical operations. They can grow quickly. This also applies to operations like addition, which is normally calculated correctly in integer arithmetic if the input quantities do not contain errors.

During the practical realization of embedded control software, investigate whether or not the resulting numerical precision will suffice after choosing the quantizations. If not, use the following possibilities:

- Select finer quantizations, if possible in the context of available word sizes.
- Select "strategic" quantizations to avoid automatically generated divisions during the re-scaling operations for expressions.
- Convert/simplify/approximate the mathematical expressions to reduce divisions or error propagation from multiplications.
- Modify algorithms altogether to make them numerically more stable.

An example for reducing error propagation:

- Consider the calculation of `I_term` as shown below.
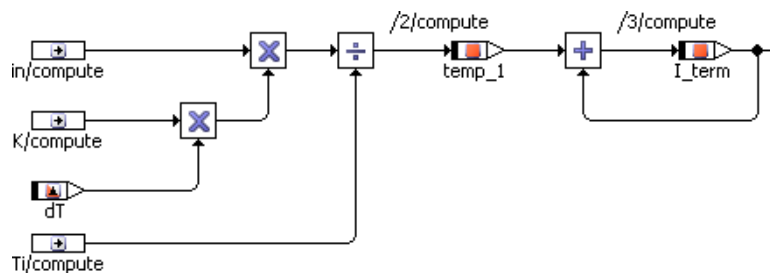
In a PID controller, the expression for the integral term is commonly written as:

$$I\_term = f_{integral}(\ in*(K/Ti)*dT\ )$$

where the function `in` and the factor `K/Ti` are computed before taking the integral. Doing so in the above expression causes numerical errors not only due to dividing first (which are then magnified by a multiplication), but also from overflow protection (i.e., due to the left shift of `K` before the division – this is explained later). Thus, the algorithm shown in the block diagram provides much better precision than the usual mathematical representation.

An even better solution is to remove the division completely by placing the inverse of `Ti` in the characteristic map in the `PIDT1_MOD` module. In doing this, however, the direct relation to the usual parameters gets lost.

12.3   Rules of Integer Code Generation

This section describes the *local rules* by which the code generator maps basic operations specified physically in the model to quantized integer arithmetic for the target. It also discusses the optimization of complex mathematical expressions.

The following principles are used for the transformation of base operations:

- **Keep numerical precision:** Numerical precision is sacrificed only if required due to overflows.

- **No overflows in intermediate results:** A priority of the code generator is to prevent overflows in the intermediate results. When required, a coarser quantization is selected automatically, even at the expense of numerical precision.

- **Minimize the number of additional operations:** When customizing quantizations for intermediate results, the number of added operations must be minimized.

- **Compliance of specified value ranges:** The code generator guarantees compliance to the value ranges specified by the user. When required, an explicit limit is generated.

The rules for transformation of base operations are derived from these principles.

12.3.1 Assignments

How is an assignment of physical quantities, e.g. `y := x`, transformed to C code with the corresponding quantized representation? To illustrate this, let us assign a quantized source value `X` to a target value `Y` with, perhaps, a different quantization:

```
assignment (phys.): y := x

source:             X  = ax+b

target:             Y  = cy+d
```

If source and target have the same conversion formula, the implementation value can be assigned directly.

```
Y := X
```

The source must otherwise be transformed into the conversion formula of the target before the assignment.

```
Y := fx,y(X)
```

One of the substantial advantages of the code generator is the automatic production of this transformation. In the first step, the source is re-scaled to match the target by multiplying with the correct conversion factor, i.e. the quotient of the target and source scales.

```
X1 := X*(c/a)
```

The offset is then adapted in a way suitable for the target.

```
Y = X2 := X1 + d – b*(c/a)
```

Re-scaling, i.e. multiplication by a rational but generally not integer conversion factor, is problematic. This multiplication can, in principle, be converted into integer arithmetic in different ways. For the following alternatives, the factor `c/a` is assumed to be a simple fraction.

- Multiply first: `(X*c)/a`

  This is the most correct variant and should always be chosen if the intermediate result is calculable without overflow.

- Divide first: `(X/a)*c`

  This possibility causes very large numerical errors, because the division error is inflated by the following multiplication.

- Approximate: `(X*c')/a'`

  Here, `c´/a´` should be a "simple" rational approximation of `c/a`, i.e., with smaller coefficients. It is generally quite difficult to design such an approximation with an algorithm. The attempt used by the code generation is the so-called continued fraction expansion.

The approach is clarified now with an example:

Suppose that `X*(20/13)` is to be calculated, with `x` bound by the interval `[0,80]`, only numbers with `8 bits (0-255)` are allowed, and the current value of `x` is `73.`

Calculation in floating point yields `112.31.`

In integer arithmetic, the following emerges:

- Multiplying first to get `(73*20)/13=112` is not feasible because the intermediate result `73*20=1460` is far too large.
- Dividing first yields too imprecise a result, namely `(73/13)*20=5*20=100.`

On the other hand, if the user chooses the approximation `3/2=1.5` for the needed division `20/13=1.538`, this becomes `(73*3)/2=19/2=109.` This result is reasonably precise, and no overflow occurs in the intermediate result.

The code generator tries to reach the highest possible numerical precision in the context of available word size. Therefore, the following algorithm is used for re-scalings:

1. The scales of the individual quantities are generally approximated by simple quotients. In doing so, it is assured that the re-scaling factor of `c/a` does not have any large coefficients.

2. If the intermediate result is representable in the available word size, then the multiplication comes first:

   `(X*c)/a`

3. Otherwise, a check is made for the amount of overflow (in bits) in the intermediate result. Then, the more numerically correct approach of the two following possibilities is selected for each individual case:

   - Divide first, then multiply:

     `(X/a)*c`

   - Right-shift by `s` places, then proceed as in step 2 above:

     `(((X>>s)*c))/a)<<s)`

     This variant is mainly used if the scale can be specified as a multiple of a power of two. The final shift operation is then dropped.

To summarize the overall process, assignments are generated in the following steps:

1. Re-scale the source to the target scale.
2. Adjust the offset.
3. Limit the value interval of the result, if necessary.
4. Assign the converted implementation value to the variable.

The assignments between actual and formal arguments for method calls are treated the same.

An assignment example:

- Consider the calculation of `P_term` shown below.
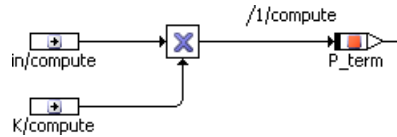
Here, the intermediate result, `in*K`, is assigned to `P_term`. The implementations are:

```
in      = 2048*in  ∈[-2,2],
K       = 64*k     ∈[0,50],
P_term  = 256*pterm ∈[-100,100]
```

Therefore, the intermediate result has a scale of `2048*64` and a range of `[-100,100]`. Assigning this to `P_term` requires a re-scaling of

$$1/512 = 256/2048/64 \text{ (i.e. } 2^{-9} = 2^{8-11-6})$$

Since all scale values are powers of two, this is simply done with a right-shift. No limits are required, and the resulting code is:

```
P_term = ((in * K) >> 9);
```

12.3.2   Addition and Subtraction

Since addition and subtraction are treated analogously, only the addition is described here.

When adding two quantities, the quantizations must be brought to the same scale value first. The offset is added thereafter. For example, you can not add two lengths in meter and kilometer without re-scaling one or the other first.

The code generation for addition is carried out in the following steps:

Re-scaling: Both operands are brought to the same scale. To avoid unnecessary loss in precision, the scale with the finer quantization is selected. If this is not possible, the less accurate representation is used. This may be the case if the coarser quantized value is not representable in the finer quantization using the available bit length.

Addition: The re-scaled operands are added including the offsets, if present.

Overflow Handling: If a possible overflow because of the specified value ranges is detected, then one or both operands are right-shifted before the addition. This reduces resolution but eliminates the overflow.

For example: Compute `x+y,` given

X = 3*x and
Y = 5*y,

both within the interval `[0,100]`. Assume only 8-bit results are valid.

- First, `X` is re-scaled to the finer scale of `Y` (5). Division is done first (loss of precision) because the intermediate result `X*5` does not fit in a byte:

    ```
    X':=(X/3)*5
    ```

- The intermediate result, `X'` has the value range `[0,165]`. The addition of `X'+Y` results in an overflow. Both operands are therefore down-scaled using a right shift before they are added.

- The generated code for the complete addition operation looks like this:

    ```
    ((X/3)*5)>>1)+(Y>>1)
    ```

> **Note**
>
> *Addition is usually seen as a commutative operation with mutually inter-changeable inputs. This is not true for the target code generation due to the application of different shift operations. The user should consider the specific situation, especially in complex arithmetic expressions.*

12.3.3   Multiplication

Unlike addition, multiplication of two quantities with different quantizations is possible. For example, multiplying `X=ax+b` and `Y=cy+d` gives

```
X*Y = acxy + adx + bcy + bd
```

However, the integer result is simplified if both operands are represented with offsets `b=d=0`. Then, the integer result is simply a linear scale of the physical result.

```
X*Y = (a*c)*(x*y)
```

As a result, the code is generated for multiplication in the following steps:

Offset brought to zero: Both operands are brought to an offset of 0.

Multiplication: The results are then multiplied.

Overflow Handling: If a possible overflow due to the specified value ranges is detected, both operands are right-shifted until the multiplication is possible without overflow. This necessary loss of resolution is divided up proportionally based on the number of significant bits in each operand.

For example: Compute `x*y`, given

```
X = 50x+3 ∈ [3,203] and
Y = 4y ∈ [0,10],
```

with only 8-bit arithmetic possible.

- First, `X` is shifted to offset 0, i.e.
    ```
    X'= X-3 ∈ [0,200].
    ```
- The multiplication `X'*Y` would result in an overflow, i.e. new interval ∈ `[0,2000]`. In order to stay within 8 bits, a right shift of three positions is necessary. The larger value X', is shifted two positions, while Y is shifted by one.
- The generated code for the multiplication is:
    ```
    ((X-3)>>2)*(Y>>1)
    ```
- The result has a scale value of 200/8=25 and an offset of 0.

> **Note**
>
> *The avoidance of overflows by performing right-shifts reduces resolution and can easily result in unsatisfactory numerical precision, especially with sequences of several multiplications using large values. However, this is not an error caused by the code generator, but an inherent problem of the limited available word length. Chains of multiplication should, therefore, only be used with caution. If required, intermediate results must be forced to a given scale value with the help of inserted variables.*

12.3.4   Division

As with multiplication, operands of different scales may be divided. Here as well, the operands must first be brought to an offset of 0. The result of the division is then scaled with the quotient of the two scale values:

```
X=ax, Y=cy and X/Y=(a/c)*(x/y)
```

Unlike multiplication, no overflow can occur here. The denominator can never become 0 at run-time. This is guaranteed with a check of the value range by the code generator. If the denominator's interval contains 0, an error message is given.

Integer division can result in considerable numerical errors, as already discussed. To reduce these, the code generator uses the following rules:

- The numerator must, as far as possible, have twice the word size of the denominator (for example, for 8-bit denominators, the numerator must be represented using 16 bits). This corresponds to the usual assembler instructions used for division in microcontroller targets.

- The numerator must make full usage of the word size.

If, at first, this is not the case, the numerator is increased with a left shift.

The code for division, correspondingly specified physically, is generated in the following steps.

Offset brought to zero: Both operands are brought to an offset of 0.

Test for zero in the denominator: If the range of values for the denominator contains 0, the code generator stops with an error message.

Increase numerator: Through some suitable left shifts, the numerator is increased so that it has twice the word size of the denominator, if possible, and makes full use of this word size.

Division: The division is finally performed.

For example: Compute `x/y`, given

X = 3x ∈[0,255] and
Y = y ∈[2,10].

- `x` is left shifted eight positions to fully use its 16-bit word size.

- Next, the division of 16-bit by 8-bit is performed. The generated code looks like this:

    ```
    (X<<8)/Y
    ```

- The result has a scale of `3*256`, an offset of `0`, and the value range `[0, 32640]`.

To calculate the integral term in the PID controller:

- Consider the calculation of `I_term` shown below. It combines assignment, addition, multiplication, and division operations.
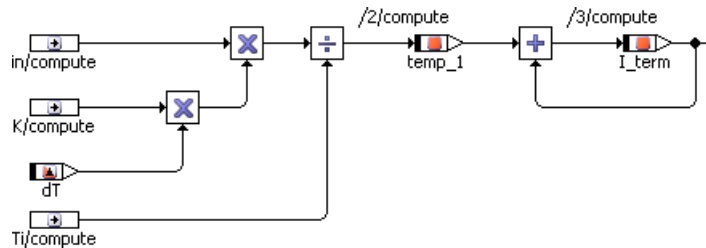
The implementations are:

```
in      = 2048*in  ∈[-2,2],
K       = 64*k  ∈[0,50],
dT      = 214*dt  ∈[0,0.1],
Ti      = 1024*ti  ∈[0.005,2],
temp_1  = 1024*temp1  ∈[-2,2],
I_term  = 256*iterm  ∈[-100,100]
```

Intermediate results may be 32 bits long. Since there is an additional variable, `Temp1`, the expression is calculated in two parts:

- The first multiplication, `K*dT`, has the scale value $64*2^{14}$ and the interval `[0,5]`. This result has no overflow as only 23 bits are needed.

- The next multiplication, `K*dT*in`, has the scale value $2^6*2^{14}*2^{11}=2^{31}$ and the interval `[-10,10]`, which creates an overflow of 4 bits. Therefore, the right-shift is divided proportionally based on the number of significant bits between `in` (12 bits) and `K*dT` (24 bits, signed).

- Next, the result is divided by `Ti`. The numerator is already using the full word size so no left-shift is required. Assigning the result to `temp_1` requires a re-scaling of $1/128=2^{-7}=2^{10+10-6-14-11+4}$. The generated code looks like this (note that clipping is required but not shown):

  ```
  temp_1 = ( ( (in>>1)*((K*dT)>>3) )/T_i )>>7;
  ```

- The second part is the addition of `temp_1 + I_term`. Normally, the finer quantization (that of `temp_1`) would be used to re-scale, but since the result must be assigned back to `I_term`, a re-scaling of $1/4=2^{-2}=2^{8-10}$ is used. This saves one re-scale operation – see more on this in section 12.3.9 on page 149. The generated code looks like this (again, clipping is not shown):

  ```
  I_term = ( (temp_1>>2) + I_term );
  ```

  - Re-examine the generated code to verify the above expressions. Note how the limiters are implemented.

12.3.5 Comparisons

Inputs for comparison operators must be transformed to a common conversion formula. Customizing of the conversion formulas occurs in two steps. First the scaling is adapted, and then the offset is adjusted.

Normally, the comparison is executed using the finer of the two quantizations, to avoid unnecessary loss of precision. If this is not possible because the re-scaled representation exceeds the available word size, the coarser quantization is used.

12.3.6 Switches and Multiplexers

As in comparisons, all inputs for switches and multiplexers must also be transformed to a common conversion formula. This is carried out analogously to the comparison operators. The selection is then executed via the usual control structures in C, i.e. `if/else, case, (a?b:c).`

12.3.7 Literals

Literals cannot have an implementation specified in ASCET. The code generator transforms literals automatically using a conversion formula matching the respective context.

For example: The computation of `x+1.0` in a model is transformed to `x+10` if `x` is scaled with `10`.

The automatically adapting quantization of the environment can result in unsatisfactory results if, through this, the literal is represented too coarsely. This can occur particularly if literals are multiplied or divided in the midst of mathematical expressions with intermediate results. For example, consider the expression

```
y := x*1.049,
```

where `x` and `y` are quantized with `0.1`. Depending on the value range for `x`, the literal `1.049` could get approximated by the integer `10` (i.e., physical value `1.0`). If this is the case, it vanishes from the expression completely:

```
y := (x*10)/10 = x
```

In order to suppress this effect, the literal gets a refined scale value. The goal is to keep the relative error lower than `0.1%`. In the example above, the literal `1.049*10=10.49` is represented as `671/64=10.484.` Hence the expression from above reads:

```
y := (x*671)/640,
```

and the factor is reasonably approximated.

> **Note**
>
> *The precision threshold of 0.001 is hard-coded and cannot be adjusted by the user. The quantization of the automatically refined literal can, therefore, become too inaccurate in rare cases. In such cases, the literal can be replaced by a constant in the model. In this case, the user can provide a conversion formula.*

12.3.8 Treatment of Operators With Multiple Inputs

Mathematical operators with multiple inputs are dissolved into sequences of binary operations for which code is then generated in succession.

Subsequently, the result in the picture below is always `True`.
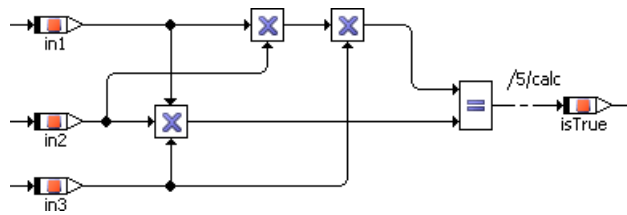
> **Note**
>
> *Usually, addition is seen as a commutative operation with mutually inter-changeable inputs. This is not true for the target code generation due to the application of different shift operations. The user should consider the specific situation, especially in complex arithmetic expressions.*

12.3.9 Optimization of Mathematical Expressions

The code generator uses a heuristic control strategy for optimizing mathematical expressions. The control strategy works in two phases. Optimization data is collected during the bottom-up semantic analysis for each intermediate result in a mathematical expression. Then, a target-scaling is defined for each result in the top-down generation phase from this data. The available degrees of freedom (see section "Degrees of Freedom and Optimization" on page 139) allow the selection of optimal scales for the overall mathematical expression. The goal is to minimize the number of additional calculations used during re-scaling.

The optimal scale values are determined using a normalized scale, i.e., the factor in the total scale value that is not a power of 2.

For example, a normalized scale of 3 indicates that the intermediate result can be scaled with $3*2^{N-1}$, i.e. with `3/2, 3, 6, 12` etc. This is important for the following reasons:

- The range of the scale must be variable, so that customizing the numerical precision to avoid overflows is possible.
- Such customizations are executed by shifts.
- The basis for this is the assumption that shift operations are more efficient than multiplications or divisions. This is true for most targets.

A simple example is presented to illustrate this approach.

Example: Compute the addition of four variables `v, w, x, y` and assign the result to `z`.

```
z := ((v+w)+x)+y
```

Assume the variables are scaled as follows:

Variable	Scale Value	Normalized
v	4	1
w	3	3
x	8	1
y	5	5
z	10	5

First, during the bottom-up semantic analysis, a set of optimal scale values is collected using the normalized scale values (i.e. excluding the power-of-two factor) for every intermediate result.

Then, in the generation phase, this local data is used to select the best scale value for each result by downward back tracing (top-down) through the entire expression.

Intermediate Result	Optimum Scale Value	Comment
`v+w`	1 or 3	Either scale value works equally well because only one re-scaling is required. In any other case, both inputs would have to be re-scaled.
`(v+w)+x`	1	Since x has the scale value of 1, it is best to scale v+w also with 1. This saves one additional re-scaling. Hence, 1 is better than 3 here.
`((v+w)+x)+y`	1 or 5	Here again both choices work equally well. At least one side needs to be re-scaled.
`z:= ((v+w)+x)+y`	5	The entire expression should be generated with the scale value of 5, because then a re-scaling is not necessary before the assignment.

These scale values are then inserted according to the necessary re-scalings and shift operations. Under the assumption that no overflow can occur for the intermediate results, the code represented below is compiled for the expression.

Intermediate Result	Scale value*
`t1 := v+((w>>2)/3)`	4
`t2 := (t1<<1)+x`	8
`z := ((t2*5)>>2)+(y<<1)`	10

*: not normalized

For clarity, the intermediate results are shown separately. During the code generation, one lengthy mathematical expression is produced for the C code. The generation of the individual operations is performed locally, according to the control strategy. No global optimization is carried out for these operations.

So far all examples from the PID controller have used scale values that are a power of two. Therefore, all re-scaling has been performed with shift operations. This makes it less evident when optimization does occur. For example, in the calculation of the integral term (see above), the final addition `temp1+I_term` was performed with the less refined scale value in order to save one shift operation in the final assignment to `I_term`.

In the next example, a scale value that is not entirely a power of two is introduced.

To optimize the derivative term calculation:

- Consider the derivative term calculation in the example of a PID-controller shown below:
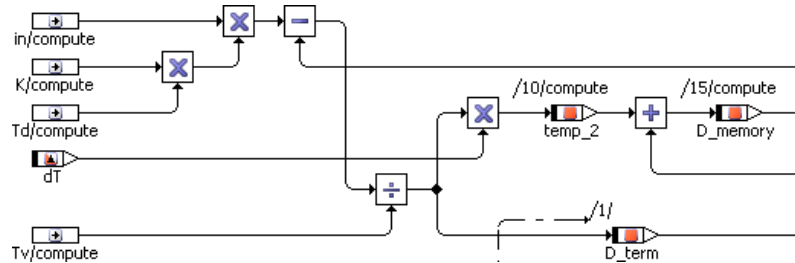
This example will focus on the calculation of `D_term` only.

- Verify the implementations of quantities in the expression. They are summarized below.

The implementations are:

```
in         = 2048*in  ∈[-2,2],
K          = 64*k   ∈[0,50],
Td         = 640*td  ∈[0,2],
Tv         = 640*tv  ∈[0.005,2],
D_memory   = 1024*dmem ∈[-10,10],
D_term     = 256*dterm ∈[-100,100]
```

Again, intermediate results may be 32 bits. The calculation of `D_term` occurs as follows:

- As in the integral term calculation, the first two multiplications, `K*Td*in`, result in an overflow of 3 bits (i.e. 3 right-shifts). The result has a scale $2^6*640*2^{11}*2^{-3} = 5*2^{21}$ and interval $[-200,200]$.

- Next, `D_memory` is subtracted from the result, but first the operands must be brought to the same scale. Here is where the optimization occurs. The following scale values must be considered:

Operand	Scale Value	Normalized
`K*Td*in`	$5*2^{21}$	5
`D_memory`	2^{10}	1
`Tv`	$5*2^{7}$	5
`D_term`	2^{8}	1

- Either normalized scale, 5 or 1, could be used for the subtraction result, `K*Td*in-D_memory`. If 1 is used, the next step of dividing by `Tv` re-introduces the scale value of 5. This result would have to be re-scaled for the final assignment to `D_term`, requiring a total of three re-scalings.

  Thus, the normalized scale of 5 is the better choice. The division by `Tv` then cancels the 5-scale out so that no rescaling is needed for the final assignment. Only one rescaling (i.e. `Dmem` to scale value 5) is then required.

- Subsequently, for the subtraction, `Dmem` is rescaled to $5*2^{21}$ (not normalized). However, both operands must be right-shifted to avoid an overflow, resulting in an actual scale value of $5*2^{20}$.
- Finally, this result is divided by `Tv`. The numerator is already using the full 32 bits, so no left-shift is required. Assigning the result to `Dterm` requires a re-scaling (i.e. right-shift) of $5*2^{7}*2^{8}/(5*2^{20})=2^{-5}$.

The intermediate results are summarized below:

Intermediate Result	Scale Value
`t1 := (in>>1)*((K*Td)>>2)`	$5*2^{21}$
`t2 := (t1>>1) - D_memory*5120`	$5*2^{20}$
`D_term := (t2/Tv)>>5`	2^{8}

Again, the intermediate results are shown separately for clarity. One lengthy expression is generated in the actual C code.

- Re-examine the generated code for the example to verify the above expressions. Note how the limiters are implemented.

13   Understanding Generated Code

This chapter describes the properties of the code generated by ASCET-SE. The basic rules of converting the ASCET model contents and structures into C code are described to help you understand what is generated and to ease a code inspection of formal review if required by your development process.

13.1   Modularity

The code generation of ASCET-SE is modular. C code and header files are created separately for each individual complex ASCET element (project, module, or class). One nested data structure is generated for each ASCET module and its element hierarchy. Knowledge of the entire system is not required for this purpose. However, the code for a module and its hierarchy can be created correctly only if, for all dependent modules, the public interface (exported variables, public methods) is known. Thus, the code generator creates an internal structure, referred to as a *class interface*, for the project and every class or module. The code generation of an element only needs the class interfaces of all referenced elements. This is analogous to the strategy frequently used for manual programming: using a header file with prototype declarations in C.

13.2   Distribution of Generated Code to Files

For each element (class, module, or project) the generated C code is divided into several files. The file names are automatically generated using Windows file format allowing a maximum of 255 characters. File names can optionally be generated in MS-DOS-compatible 8.3 format.

The following rules apply:

- A C source code (`*.c`) file is generated for the project, each module and each class and implementation. C header (`*.h`) files are generated according to the setting of the "Header Structure" configuration option in the "Build" node of the "Project Properties" window.

    - `Component` (default): a header file is generated for the project, each module and each class.

    - `Module`: a header file is generated for the project and each module.

    - `Project`: a header file is generated for the project only.

- For elements with external C code, two additional files (`*E.c` and `*E.h`) are generated that contain the external code.

- All generated header files of a project are included by the code generation via the `FILES_HEADER_PROJ` variable (see Chapter 5.4.3).

- A `function_declarations.h` file is generated, containing extern declarations of all functions of the ASCET model.

- A `variable_declarations.h` file is generated, containing extern declarations of all variables and parameters of the ASCET model.

13.2.1   Include Hierarchy

The include hierarchy of the generated code depends upon the setting of the "Header Structure" configuration option in the "Build" node of the "Project Properties" window.

The following figures (Fig. 13-1, Fig. 13-2, Fig. 13-3) show the differences between the three options and use the same key:
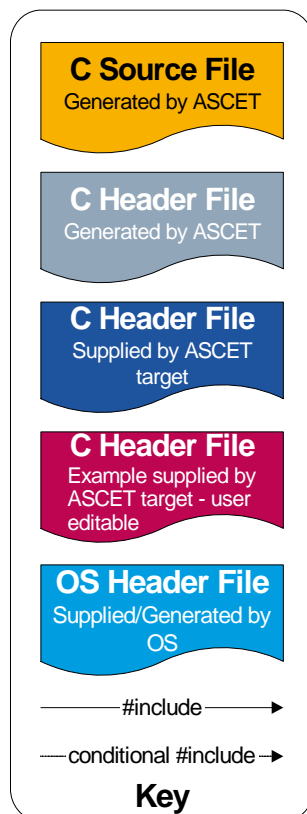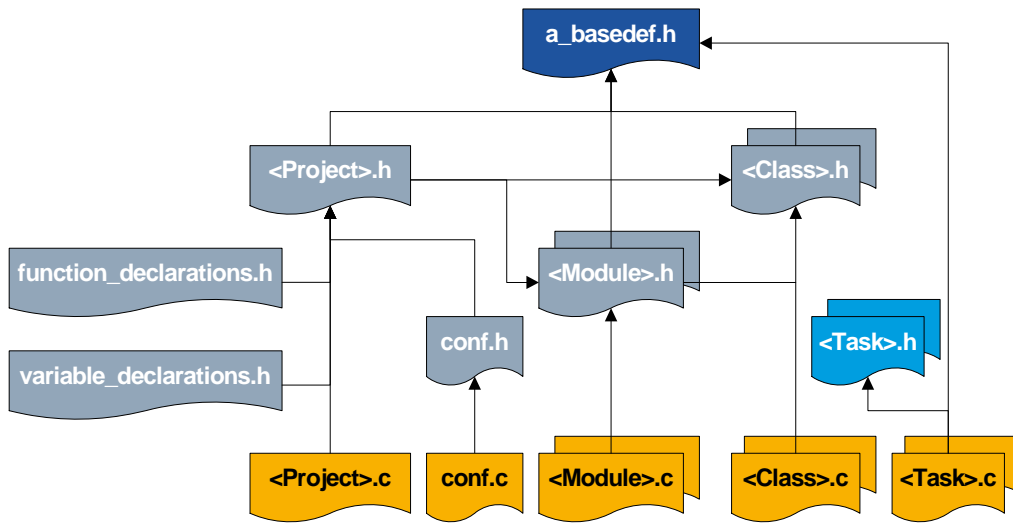
Fig. 13-1   Include Hierarchy: Component Headers
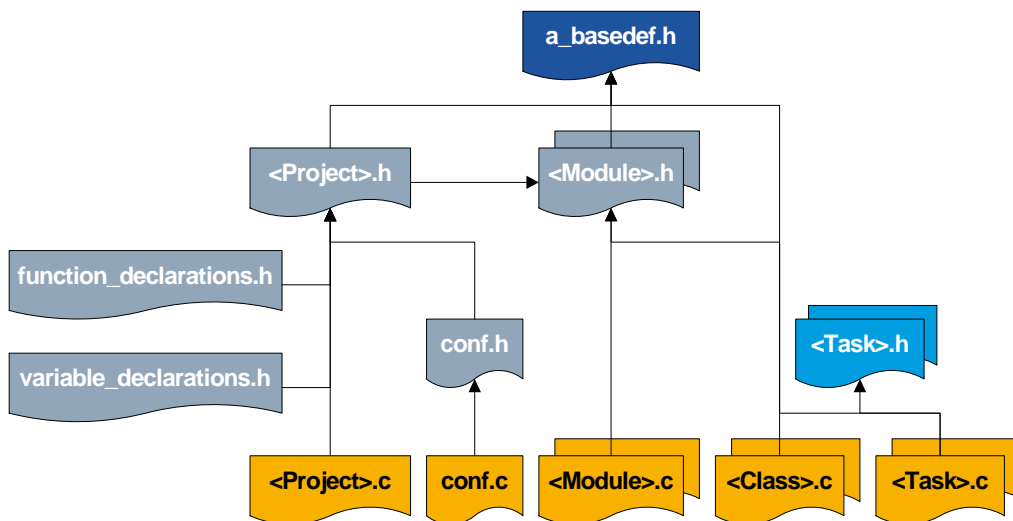
Fig. 13-2   Include Hierarchy: Module Headers
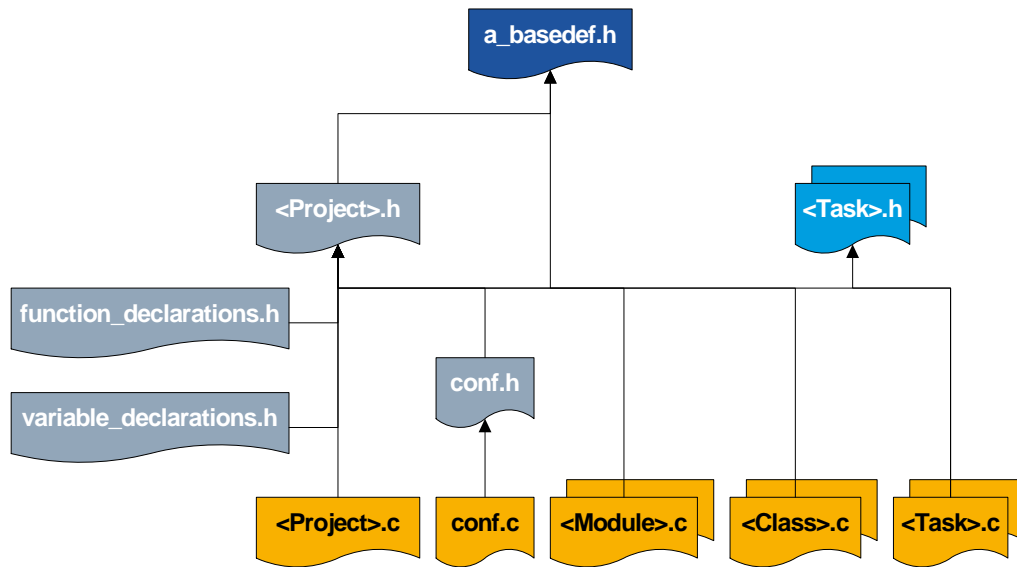
Fig. 13-3   Include Hierarchy: Project Headers

The include hierarchy of `a_basedef.h` itself is identical for all variants and is shown in Fig. 13-4.
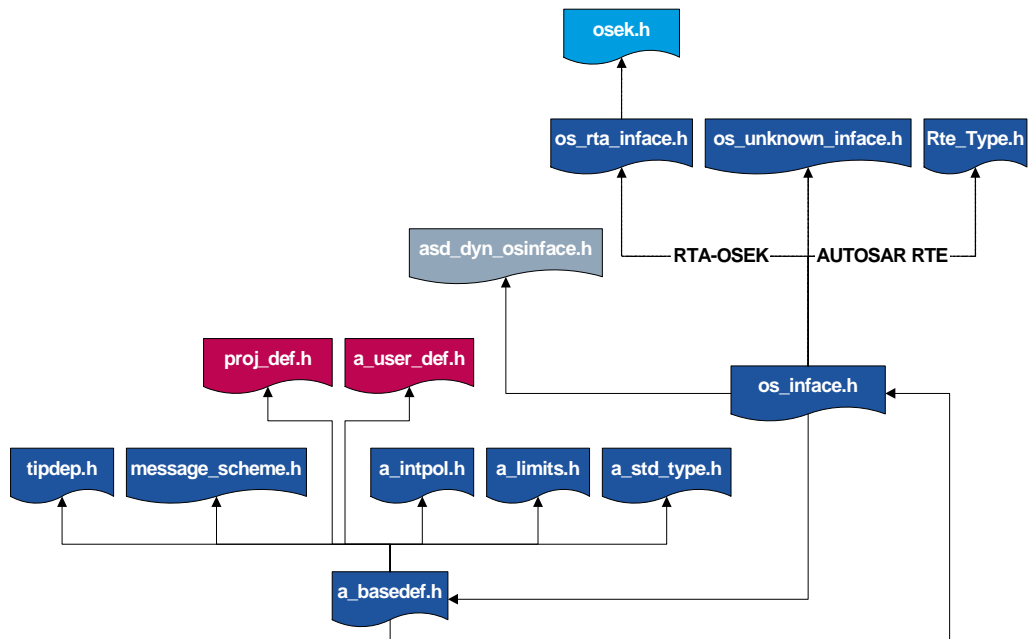
Fig. 13-4   Include Structure of `a_basedef.h`

13.3   Software Architecture

We consider *software architecture* to mean all the basic rules by which the ASCET model data and function structures are converted into C code. This includes, among other things, naming conventions, supported storage systems, and the conversion of data structures. A common *Base Software Architecture* is used for all ASCET SE targets. Its essential parts will be described in this section.

The major *design criteria* of this software architecture are the following:

- The *instantiation* of data, and thus the reservation of memory, in the controller is completely *static*. The use of dynamic allocation is not allowed. For example, memory and run-time overhead for variables caused by pointer management and malloc calls are intolerable.

- The chosen data structure must allow a static *multiple instantiation* of classes, whereby the same code is to be used for all instances with the same implementation. It would waste memory to duplicate the same code.

- Optimization occurs throughout the system.

- Data storage in user-defined memory classes is supported.

- All static data, such as parameters, must also be *initialized statically.*

The following *design decisions* were made based on the above criteria:

- Exported parameters are statically created and initialized as global C variables in the exporting C file; they are declared as external in the importing files. Parameters are assigned to a ROM area.

- Exported and imported variables are treated similarly, but created and initialized statically in a RAM area. Variables specified as non-volatile are not initialized statically. If this is required, then you must write the initialization code yourself.

- The local elements of classes and modules are stored in specific C structures. If they pertain to different memory classes, C structures are added for each memory class. They can be accessed by using C pointers. Based on the model structure, for each module a so-called instance tree is created by nesting (modules contain classes that may contain instances of other classes as elements). Besides embedding instances into a structure, access by pointer is also possible if the "as reference" option has been selected in the model. This is necessary in cases where two objects are to mutually reference each other (e.g., the wheels of a vehicle axle).

- A pointer to the memory area of the receiving instance is passed in each method call allowing the same code of the methods to be used also for the instances of all classes having the same implementation (The so called *self*-pointer. This applies only to multiple instance generation or if explicitly configured in the element's implementation).

- For each memory area, the elements of a component are grouped in a structure. For each component (provided it contains data), a structure exists from which the memory class structures are referenced.

- All implicit initializations are static.

- Only one fixed storage system (record layout) for characteristic curves and maps is supported.

13.3.1   Naming Conventions

The C name for an ASCET component (i.e., class, module, or project) is built according to the following convention:

```
<name of component>_<name of implementation>
```

The addition of the implementation name is required for classes because several instances of a class can occur in the model along with different implementations. This name is called the `classIdentifier` in the following sections.

Modules and projects have a single instance, so the addition of the implementation name could be avoided for these components. For consistency, however, the above convention is followed for these components as well.

In contrast to code generation for experimental targets, this naming convention produces the *restriction that class, module, and project names have to be unique within the project*. Otherwise, malfunctions or compiler/linker errors could occur. The uniqueness of the name is, therefore, checked in the make mechanism at the start of the code generation.

The user can partially modify the rules for producing class and variable names in the expander configuration file `codegen.ini` (see chapter 5.1 "The codegen[_*].ini Files" for more details).

13.3.2   Storage Systems, Data Structures, Initialization of Primitive Objects

A generic object structure, which allows the recording and changing of data at arbitrary locations during simulation, is used for supporting experimental targets. The dynamic memory allocation associated with it would have memory and run-time requirements which are too high for use in the controller. The supplementary data used in the simulation are not needed in the controller. They are replaced by condensed structures which are *preset* by this base software architecture and cannot be modified by users.

The generic definitions for implementation types (e.g., `uint16`, etc.) are also used in the controller and are defined in a global system header file.

> **Note**
>
> *In the following examples,* global *elements are shown for clarity because their data structures are created isolated (i.e. not embedded in the instance tree). Thus, the generated data structures for declaration and initialization can be documented. For local elements, declaration and initialization are generated accordingly, but embedded in the instance tree.*

Scalar and Logical Values

Global scalar and logical values are directly realized by a C variable of specified implementation type:

```
uint16 scalarVariable;
```

The initialization of global scalar parameters and variables occurs statically in the definition:

```
const uint16 scalarParameter = 123;
```

Local values are defined and initialized as parts of data structures. Non-volatile variables are not initialized, no matter if they are local or exported.

> **Note**
>
> *Variables specified as non-volatile are not initialized at all.*

Arrays and Matrices

Arrays and Matrices are directly realized as C arrays of the specified implementation type.

Arrays and matrices can be created as variables or parameters. They *cannot* be created as constants or system constants because these are generated as `#define`. In case of a migration from older ASCET versions, possibly existing system constants *have* to be switched to parameters manually.

By default, the array size is fixed and, therefore, cannot be modified at execution time. It corresponds to the size in the model. Matrices are generated as one-dimensional arrays in the C code.

```
sint32 array[size];
uint16 matrix[size];
```

To be able to modify array/matrix sizes at runtime, you must use system constants for size definition (see the ASCET online help for details) and set the `ResolveSystemConstants` option in `codegen.ini` to `CompileTime`[1]. If an array or matrix uses one or two system constants as size definition, the declaration includes the system constants:

```
sint32 array[SysConst_x];
uint16 matrix[SysConst_x][SysConst_y];
```

It is also possible that a matrix uses a system constant for one dimension, and a fixed value for the other dimension:

```
uint16 matrix[SysConst_x][3];
```

Arrays/matrices specified as explicit references are generated as pointers:

```
sint32 * array;
uint16 * matrix;
```

Note

For multiple instances, the size of an array or matrix must be the same in all instances since the same object definition is used for all data records. The size is not stored with it. It is not required because the array size cannot be accessed in the model.

Arrays are initialized in the generated C code, and stored in the memory, in order of increasing index. Matrices are initialized and stored in *column-major-order*.

Example – normal matrix: The following normal matrix,

$$
\texttt{matrixNormal} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}
$$

would be stored as

```
1 4 7 2 5 8 3 6 9
```

---

[1] If you set `ResolveSystemConstants=GenerationTime`, the init values of the system constants are used.
`ResolveSystemConstants=RunTime` produces an error.

and initialized as

```
{
    1, 4, 7,
    2, 5, 8,
    3, 6, 9
}
```

Example – matrix with system constants: The matrix `matrixVariant` is specified with a numerical max size of 3 x 3, and the system constants `SC_x` and `SC_y` are assigned as X and Y variant size. 3 x 3 values can be entered; they are set as follows.:

$$
matrixVariant = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}
$$

The matrix initialization is generated as follows:

```
{
  {
    1
    #if SC_y >= 2
    , 4
    #endif
    #if SC_y >= 3
    , 7
    #endif
  }
  #if SC_x >= 2
  ,
  {
    2
    #if SC_y >= 2
    , 5
    #endif
    #if SC_y >= 3
    , 8
    #endif
  }
    #endif
  #if SC_x >= 3
  ,
  {
```

```
        3
        #if SC_y >= 2
        , 6
        #endif
        #if SC_y >= 3
        , 9
        #endif
    }
    #endif
}
```

Exported array/matrix parameter: Initialization of a global (i.e. exported) parameter array occurs statically in the C code definition:

- array without system constants

```
const uint16 arrayParam[4] =
    { 10,1,4,9 };
```

- array with system constant `SC_x`

```
const sint32 array_paramSC[SC_x] = {
    10
    #if SC_INDEX4 >= 2
    , 1
    #endif
    #if SC_INDEX4 >= 3
    , 4
    #endif
    #if SC_INDEX4 >= 4
    , 9
    #endif
};
```

Local arrays/matrices: Local values are defined and initialized as parts of nested data structures. Non-volatile variables are not initialized, no matter if they are local or exported.

Enumerations
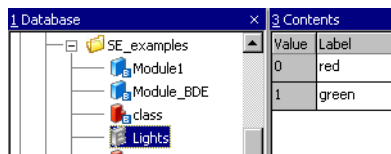
Enumerations are mapped onto a primitive integer data type in the C code. In contrast to the C type `enum`, usually less than machine word is necessary in this way to represent an enumeration.

```
uint8 Lights;
```

The symbolic names (`red` and `green` in the example) are mapped onto integer values. In the ASAM-MCD-2MC description file generated by ASCET, the respective symbolic name is assigned again to each integer value so that these names are visible in the application system:

```
/begin COMPU_VTAB enum_Lights_tab_ref
""
TAB_VERB
2
0 "red"   1 "green"
DEFAULT_VALUE "Error"
/end COMPU_VTAB
```

Characteristic Curves

The following simple storage system is used for characteristic curves:

Value Stored	Description	Number of Bytes
n (start address)	No. of interpolation nodes	1 or 2 bytes (see below)
X1		
X2		
…	interpolation nodes	n*x bytes, increasing index
Xn		
W1		
…	characteristic values	n*w bytes, increasing index
Wn		

Tab. 13-1   Storage system - characteristic curve

The number of nodes is stored in one byte if both the nodes and the characteristic values (`X` or `W`) are represented in one byte. Otherwise, two bytes are used.

For such a storage system, no generic structure definition can be used in C because the number of nodes and the implementation types of nodes and values can vary. A separate structure definition must therefore be produced by code generation for *every individual characteristic curve*. This definition must be named and entered into the C header code because of the separate generation of module and initialization code.

Characteristic curve – example:
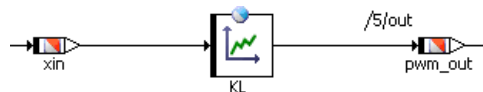
`KL` has three nodes. The input and output data types are both `sint16`.

In the C code, the structure is defined as follows (component header file `<component>.h`):

```
struct PIDT1_MOD_IMPL_KL_TYPE {
   uint16 xSize;
   sint16 xDist [3];
   sint16 values [3];
};
```

The static initialization of the global element `KL` occurs in the declaration:

```
const struct PIDT1_MOD_IMPL_KL_TYPE KL =
{
   3,
   {
     -2, 1, 4
   },
   {
     5, 6, 7
   }
};/*** KL ***/
```

Local characteristic curves are defined and initialized as parts of nested data structures.

The storage system makes no distinction between the current and maximum number of nodes. An *adjustment of the number of nodes during calibration is not planned.* The dimensions of the vectors in `struct` correspond to the current number of nodes set at generation time.

Access occurs with the help of access routines. There are two possibilities of accessing characteristics: "linear", i.e. by means of interpolation routines, or "rounded", i.e. using the characteristic as a look-up table. Both kinds of access routines are shipped with ASCET-SE. For the above example, linear access looks like this:

```
pwm_out = CharTable1_getAt_s16s16
         ((void *)&KL, xin);
```

Code for rounded access is generated as follows:

```
pwm_out = CharTable1_getAtR_s16s16
         ((void *)&KL, xin);
```

Note

When creating access routines, be aware that the storage of the structure elements in the memory ("Alignment") is defined by the compiler.

In the data editor of a characteristic, the user can specify whether to use linear or rounded access.

Characteristic Maps

The storage system for characteristic maps is illustrated in the following table. It is similar to characteristic curves:

Value Stored	Description:	Number of Bytes
n (start address)	No. of X interpolation nodes	1 or 2 bytes (see below)
m	No. of Y interpolation nodes	1 or 2 bytes (see below)
X1		
…	X interpolation nodes	n*x bytes, increasing index
Xn		
Y1		
…	Y interpolation nodes	m*y bytes, increasing index
Ym		
W1,1		
W1,2		
…	characteristic values	(n*m)*w bytes, column-major ordering (Y index m increases faster than X index n)
Wn,m-1		
Wn,m		

Tab. 13-2   Storage system – characteristic map

Here, the number of X and Y nodes (n and m) are both stored in one byte if all of the nodes and characteristic values (X, Y, or W) are represented in one byte. Otherwise, two bytes are used.

As in case of characteristic curves, code generation produces a *separate* `struct` definition for *every individual characteristic map*.

Characteristic map – example:
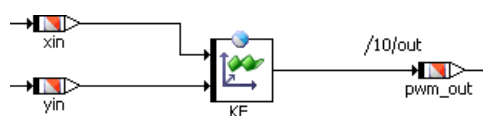
KF has three nodes on the x axis and four on the y axis. Both input data types are `sint16`, and the outputs are `uint16`.

In the C code, the structure is defined as follows (component header file `<component>.h`):

```
struct PIDT1_MOD_IMPL_KF_TYPE {
   uint16 xSize;
   uint16 ySize;
   sint16 xDist [3];
   sint16 yDist [4];
   sint16 values [3 * 4];
};
```

The static initialization of the global element `KF` occurs again in the declaration:

```
const struct PIDT1_MOD_IMPL_KL_TYPE KF =
{
   3,
   4,
   { 1, 3, 5 },
   { 0, 1, 8, 15 },
   { -5, -3, 0, 1,
     0, 1, 4, 6,
     8, 5, 4, 4 }
};/*** KF ***/
```

Local characteristic maps are defined and initialized as parts of nested data structures.

Nodes and values are stored by increasing index; the respective storage space is reserved for the number of nodes currently set at generation time. The storage of the value matrix is column-by-column. Everything else is the same as for characteristic curves. Access takes place in an analog way, too, as the following example for linear access (i.e. an interpolation routine call) shows:

```
pwm_out
   = CharTable2_getAt_s16s16s16(
     (void *)&KF, xin, yin);
```

Also the rounded access (i.e. look-up functionality) is similar to curves:

```
pwm_out
   = CharTable2_getAtR_s16s16s16(
     (void *)&KF, xin, yin);
```

The user can specify in the data editor of a characteristic whether to use linear or rounded access.

Interpolation Node Distributions, Group Characteristic Curves and Maps

For group characteristic curves and maps, *only the values* are stored *as an array* with increasing index.

Value Stored	Number of Bytes
`W1` (start address)	
...	`n*w` bytes, increasing index
`Wn`	

Tab. 13-3   Storage system - group characteristic curve

The respective interpolation nodes are saved in separate objects, the interpolation node distributions.

Value Stored	Description	Number of Bytes
n (start address)	number of interpolation nodes	2 Byte
X1		
X2		
…	interpolation nodes	n*x bytes, increasing index
Xn		

Tab. 13-4   Storage system interpolation node distribution

An interpolation node distribution can thus be used for several group characteristic curves or maps.

Example for interpolation node distributions and group characteristic curves:
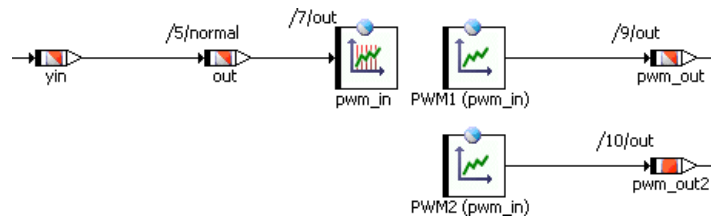
PWM1 and PWM2 have six interpolation nodes each, as defined in pwm_in. pwm_in has an input data type of uint16. Both curves have an output data type of uint16.

The static definitions in the C code have the following form (component header file <component>.h):

```
struct PIDT1_MOD_IMPL_pwm_in_TYPE {
  uint16 size;
  uint16 dist [6];
};
struct PIDT1_MOD_IMPL_PWM1_TYPE {
  sint16 values [6];
};
struct PIDT1_MOD_IMPL_PWM2_TYPE {
  sint16 values [6];
};
```

Additionally, three variables are generated for each interpolation node distribution, as intermediate memory for the interpolation results. They are then used to access the group characteristic curve.

```
uint16 pwm_in_index;
uint16 pwm_in_offset;
uint16 pwm_in_distance;
```

Because these elements are exported in the example, the initialization of the data structures is again performed in separate structures. The intermediate variables are not initialized separately.

```
const struct PIDT1_MOD_IMPL_pwm_in_TYPE pwm_in =
{
  6,
  {
    0, 4, 8, 10, 12, 13
  }
};/*** pwm_in ***/
const struct PIDT1_MOD_IMPL_PWM1_TYPE PWM1 =
{
  {
    1584, 16, 16, 0, 0, 0
  }
};/*** PWM1 ***/
const struct PIDT1_MOD_IMPL_PWM2_TYPE PWM2 =
{
  {
    16, 16, 1584, 0, 0, 0
  }
};/*** PWM2 ***/
```

Local distributions and group characteristic curves are defined and initialized as parts of nested data structures.

Access occurs in two steps, analog to the model. First, a search for the interpolation nodes is performed.

```
Distribution_search_u16(
    (void*)&pwm_in.dist,
    (uint16)pwm_in.size,
    (uint16)out,
    (void *)&pwm_in_index,
    (void *)&pwm_in_offset,
    (void *)&pwm_in_distance);
```

The results of the search for interpolation nodes are stored in the intermediate variables pwm_in_index, pwm_in_offset and pwm_in_distance. After that, these results can be accessed with the help of special interpolation routines. Thus, several different characteristic curves and maps can be evaluated based on one search for interpolation nodes.

```
pwm_out
    = GroupTable1_getAt_u16s16(
      (void*)&PWM1,
      pwm_in_index,
      pwm_in_offset,
      pwm_in_distance);

pwm_out
    = GroupTable1_getAt_u16s16(
      (void*)&PWM2,
      pwm_in_index,
      pwm_in_offset,
      pwm_in_distance);
```

Example for interpolation node distributions and group characteristic map:
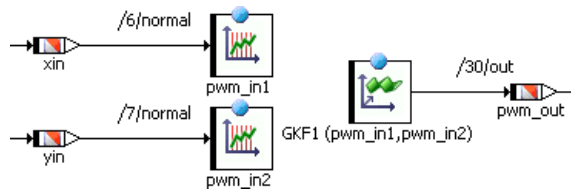
GKF1 has four X interpolation nodes (defined in pwm_in1) and three Y interpolation nodes (defined in pwm_in2). pwm_in1 and pwm_in2 have an input data type of uint16, the characteristic map has the output data type sint16.

The static definitions in the C code have the following form (component header file <component>.h):

```
struct PIDT1_MOD_IMPL_pwm_in1_TYPE {
   uint16 size;
   uint16 dist [4];
};

struct PIDT1_MOD_IMPL_pwm_in2_TYPE {
   uint16 size;
   uint16 dist [3];
};

struct PIDT1_MOD_IMPL_GKF1_TYPE {
   sint16 values [4 * 3];
};
```

Again, the three intermediate variables are generated for each interpolation node distribution.

```
uint16 pwm_in1_index;
uint16 pwm_in1_offset;
uint16 pwm_in1_distance;

uint16 pwm_in2_index;
uint16 pwm_in2_offset;
uint16 pwm_in2_distance;
```

Initialization of data structures:

```
struct PIDT1_MOD_IMPL_pwm_in1_TYPE pwm_in1 =
{
   4,
   {
      0, 4, 8, 12
   }
};/*** pwm_in1 ***/

struct PIDT1_MOD_IMPL_pwm_in2_TYPE pwm_in2 =
{
   3,
   {
      1, 2, 3
   }
};/*** pwm_in2 ***/
```

```
struct PIDT1_MOD_IMPL_GKF1_TYPE GKF1 =
{
  {
    -5, -3, 0,
    0, 1, 4,
    8, 5, 4,
    19, 7, 0
  }
};/*** GKF1 ***/
```

Local group characteristic maps are defined and initialized as parts of nested data structures.

The search for interpolation nodes is done separately for each interpolation node distribution:

```
Distribution_search_u16(
    (void *)&pwm_in1.dist,
    (uint16)pwm_in1.size,
    (uint16)xin,
    (void *)&pwm_in1_index,
    (void *)&pwm_in1_offset,
    (void *)&pwm_in1_distance);

Distribution_search_u16(
    (void *)&pwm_in2.dist,
    (uint16)pwm_in2.size,
    (uint16)yin,
    (void *)&pwm_in2_index,
    (void *)&pwm_in2_offset,
    (void *)&pwm_in2_distance);
```

The results of the search for interpolation nodes are stored in the intermediate variables. After that, these results can be accessed with the help of special interpolation routines.

```
pwm_out
  = GroupTable2_getAt_u16u16s16(
    (void *)&GKF1,
    pwm_in1_index,
    pwm_in1_offset,
    pwm_in1_distance,
    (uint16)pwm_in1.size,
    pwm_in2_index,
    pwm_in2_offset,
    pwm_in2_distance,
    (uint16)pwm_in2.size);
```

Fixed Characteristic Curves and Maps

Fixed characteristic curves and maps have equidistant axis points, so there is no need to store the axis points extensionally in a distribution array. Instead, the data structure can store the intensional description based on the number of axis points, the offset to the first point and the distance between points.

Value Stored	Description	Number of Bytes
n (start address)	No. of interpolation nodes	2 Byte
Xoff	offset of the first interpolation node	2 Byte
Xdist	distance between interpolation nodes	2 Byte
W1 … Wn	characteristic values	n*w bytes, increasing index

Tab. 13-5 Storage system fixed characteristic curve

Value Stored	Description	Number of Bytes
n (start address)	No. of X interpolation nodes	2 Byte
m	No. of Y interpolation nodes	2 Byte
Xoff	offset of the first X interpolation node	2 Byte
Xdist	distance between X interpolation nodes	2 Byte
Yoff	offset of the first Y interpolation node	2 Byte
Ydist	distance between Y interpolation nodes	2 Byte
W1,1 … Wn,m	characteristic values	(n*m)*w bytes, column-major ordering (Y index m increases faster than X index n)

Tab. 13-6 Storage system - fixed characteristic map

Fixed characteristic curve - Example:

The fixed characteristic curve FKL1 has five interpolation nodes with the distance 2. The offset of the first interpolation node is 0.

In the C code, the declaration for this exported characteristic curve has the following form (component header file `<component>.h`):

```
struct PIDT1_MOD_IMPL_FKL1_TYPE {

   uint16 xSize;

   sint16 xOffset;

   uint16 xDistance;

   sint16 values [5];

};
```

The definition and static initialization of the fixed characteristic curve look like this:

```
const struct PIDT1_MOD_IMPL_FKL1_TYPE FKL1 =

{

   5,
   0,
   2,
   {
      0, 1, 2, 3, 4
   }
};/*** FKL1 ***/
```

Local fixed characteristic curves are defined and initialized as parts of nested data structures.

Fixed characteristic curves and maps can be evaluated by direct calculations of indices, without special subroutines (search routines), because they have constant and equidistant interpolation nodes. In the example, the C code has the following form:

```
pwm_out = CharTableFixed1_getAt_s16s16(&FKL1,xin);
```

Fixed characteristic map - Example:
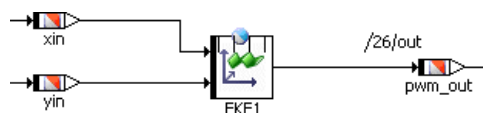
The fixed characteristic map `FKF1` has four interpolation nodes on the x-axis and five on the y-axis. The X interpolation nodes have an offset of 2 and a distance of 2, The Y interpolation nodes have an offset of -3 and a distance of 3.

In the C code, the declaration for this exported characteristic map has the following form (component header file `<component>.h`):

```
struct PIDT1_MOD_IMPL_FKF1_TYPE {

   uint16 xSize;
   uint16 ySize;
   sint16 xOffset;
   sint16 yOffset;
   uint16 xDistance;
   uint16 yDistance;
   sint16 values [4 * 5];
};
```

The definition and static initialization of this global fixed characteristic map look like this:

```
const struct PIDT1_MOD_IMPL_FKF1_TYPE FKF1 =
{
   4,
   5,
   2,
   -3,
   2,
   3,
   {
      23, 23, 24, 25, 26,
      23, 15, 16, 17, 18,
      23, 7, 8, 9, 10,
      23, -1, 0, 1, 2
   }
};/*** FKF1 ***/
```

Local fixed characteristic maps are defined and initialized as parts of nested data structures.

The call in the C code has the following form:

```
pwm_out =
CharTableFixed2_getAt_s16s16s16(&FKL1,xin,yin);
```

13.3.3 Data Structures and Initialization for Complex (User-Defined) Objects

Classes

A C structure is defined for each user-defined class. It contains the instance variables of the classes, ordered in terms of memory classes. The name of the structure is the C name of the class (class + implementation name; see also section 13.3.1 "Naming Conventions"). For each memory class, an individual structure is generated and referenced. All instance variables can be accessed directly via this structure. There are *no exceptions*. From the PID controller example, the structure definition for the class `PIDT1` is:

```
struct PIDT1_IMPL_RAM_SUBSTRUCT {
   sint16 temp_1;
   sint16 temp_2;
};

struct PIDT1_IMPL {
   struct PIDT1_IMPL_RAM_SUBSTRUCT *PIDT1_IMPL_RAM;
   sint16 memory_D_term;
   sint16 D_term;
   sint16 P_term;
   sint16 I_term;
};
```

An instance of a user-defined class is created in the C code by creating a structure with the type of the class `PIDT1_IMPL`.

To access class instance variables in methods, you can usually directly access the values stored in the structure. However, this is not so when multiple instances of the same class are allowed. In this case, an additional receiver argument (*self*

pointer) is used. This way, the same code for the method can be used for all instances of the class. Again using the `PIDT1` class as an example, the call for the `compute` method looks like the following:

```
void PIDT1_IMPL_compute (const struct PIDT1_IMPL
   *self, sint16 in, uint16 K,
   uint16 Tv, uint16 Ti, uint16 Td) {
   sint32 _t1sint32;
   sint16 _t1sint16;

   ...(the rest of code for method "compute")

};
```

Note

The receiver is omitted if only one instance is used per class. The respective components are determined in the global analysis.
The optimization of the self pointer can be switched off in the class implementation editor.

Prototype Classes

- encapsulation of extern declarations with `define`
- no function bodies
- no local data structures

Service Routines

- no function bodies
- local data structures
- special naming convention

Modules

Modules are treated like classes by the code generator. In addition, each module contains the root for its so-called instance tree, the nested data structure for all local elements located in the module's hierarchical element structure.

Only one instance can be defined for each module. It is therefore possible to directly access all instance variables and parameters of the module. Different from classes, a *self* pointer is not required. Processes are implemented as void-void functions. The `normal` process in `PIDT1_MOD` looks like this (with most of the code left out):

```
void PIDT1_MOD_IMPL_normal (void) {
   ...
   PIDT1_MOD_IMPL_TP_cmd_d =
      CharTable1_getAt_s16u16((CharTable1*)&
      (PIDT1_MOD_IMPL_Cmd_pct2deg),
      (sint16)_t1sint16);

   ...(the rest of code for process "normal")

};
```

Boolean Tables

Boolean tables are treated like classes during code generation. They are special only in so far that they may not include parameters.

The logical dependencies defined in the table are converted into sequences of logical operators, as shown in the following example:

```
sint8 CLASS_BOOLTAB_Y1
        (struct CLASS_BOOLTAB_Obj *self)
{
   return ( (sint8) ( (
        ((!_X1) && _X2)
        || (_X1 && (!_X2)) )
        || ((_X1 && _X2)
          && _X3) ) );
}
```

Conditional Tables

Conditional tables are transformed into ESDL classes internally and processed by the code generation accordingly. See the ASCET online help for a description of their functionality.

13.3.4   Local Variables and Parameters

Local elements are realized in the code as parts of data structures (see section 13.3.2 on page 158). In the generated code, these elements are accessed via the path name provided by their respective data structure. To increase the readability of the generated code, the complex hierarchical names are mapped to simple names via preprocessor definitions.

Example:

```
#define _a ModuleA_IRAM.Class.a

#define _b ModuleA_IRAM.Class.b

...

void CLASS_IMPL_calc (void)
{
   _a = _b;
}
```

13.3.5   Exported and Imported Variables

Exported variables and messages are implemented as global C variables defined in the code of the exporting module.

In ASCET, exported variables are commonly referred to as *class variables* in the sense that they only exist once for all instances of a class.

An imported variable is accomplished by using its global C-variable name directly in the importing module's generated code. To do so, the variable is declared as `external` in the header of the importing module. The code for the importing

module thus has a direct reference into the exporting module code, and is therefore not completely modular at this point. A pointer assignment for the linkage, as in the simulation code, does not exist.

> **Note**
>
> *After changes, such as renaming or converting of exported variables, the user needs to explicitly regenerate the entire model in the export/import structure. This is achieved by choosing* **Build** → **Touch** → **Recursive** *prior to code generation.*

Also in this case, the element names are mapped via preprocessor definitions.

13.3.6   Method Declarations and Calls

A method's C name results from concatenating the class identifier and method name with an underscore in between:

```
classIdentifier_methodName()
```

The C name of a method's formal argument agrees with the model name:

```
returnType classIdentifier_methodName(argType1
    argName1, argType2 argName2)
```

The passing of parameters, such as arguments and return values, depends on whether the type is a value or a pointer.

- *Scalar and Boolean parameters* are passed directly as *value* of the corresponding implementation type.
- *Characteristic curves and maps* pass a *pointer* to the structure of the characteristic curve/map.
- *Arrays and Matrices* pass a *pointer* to the first element.
- *Complex objects* pass a *pointer* to the corresponding class structure.

This corresponds with the semantics which is generally defined in ASCET, and which also holds in the physical experiment: scalar and Boolean parameters are passed by value, all other types by reference.

To handle multiple instances correctly, an additional parameter with the C name `self` is inserted into the first location of the parameter list. A pointer to the receiver of the method call or its instance variable structure is passed in this parameter. This parameter is eliminated in the following cases where it is not needed:

- Processes of modules because they can have only one instance.
- Methods of classes without instance variables because in this case the receiver is irrelevant.

However, the generation of this parameter can be forced by means of the respective setting in the implementation editor of a class.

As an example, the `out` method in the `PIDT1` class has the form:

```
sint16 PIDT1_IMPL_out
    (const struct PIDT1_IMPL *self);
```

13.3.7   Constants and Literals

Literals are represented as such, namely literals, in the C code. They are transformed depending on the implementation context when needed. The same holds true for constants. Both cannot be implemented. In addition, constants are created in the C code using `#define`.

Example:
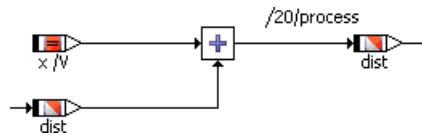
The constant used in the example is represented in the generated C code as follows:

```
/**** constants defined by module MOD_IMPL ****/
#ifndef MOD_IMPL_X
#define MOD_IMPL_X 2.0
#endif
```

The following code is generated for the example:

```
void MOD_IMPL_process (void) {
   dist = ((dist + (sint16)2));
   /* min=-10, max=10, hex=1phys+0 */
   /* end of process MOD_IMPL_process */
```

Constants created as global elements are generated without the appended project and implementation names, according to the naming convention for other global elements.

Example:
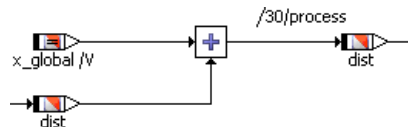

```
/**** exported constant ****/
```

```
#ifndef X_GLOBAL
#define X_GLOBAL 5.0
#endif
```

The generated code is equivalent to that for a local constant:

```
void MOD_IMPL_process (void) {
   output = ((dist  +  (sint16)5));
   /* min=-20, max=20, hex=1phys+0 */
   /* end of process MOD_IMPL_process */
```

13.3.8   System Constants

System constants are created in the C code via `#define`, and used symbolically. They can be implemented. In the following example, the system constant was created with a quantization of 1/2.
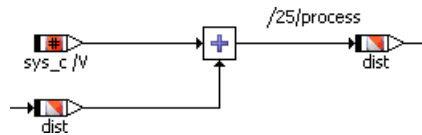
The following code is generated for the definition of the system constant:

```
#ifndef MOD_IMPL_SYS_C
#define MOD_IMPL_SYS_C 2
#endif
```

The system constant is used symbolically in the function definition:

```
void MOD_IMPL_process (void) {
    dist = ((((sint16)MOD_IMPL_SYS_C << 1) + dist));
    /* min=-10, max=10, hex=1phys+0 */
    /* end of process MOD_IMPL_process */
```

System constants created as global elements are generated without the appended project and implementation names, like global constants (see page 176). The names are generated in capital letters in each case. ASCET model names are adapted, if necessary.

13.3.9   Virtual Parameters

Virtual parameters are parameters that do not exist physically in the control unit memory. Instead, they can be used to define real (i.e., non-virtual) dependent parameters. In combination with a calibration system supporting this mechanism (e.g., INCA), it is then sufficient to calibrate a virtual parameter in order to affect several real parameters at the same time.

Example: Suppose the radius of a wheel is defined as a virtual parameter. Therefore, it cannot be used in the ASCET model directly. The diameter and circumference of the wheel are defined as parameters dependent on the radius, and they are used in various locations of the model.

> **Note**
>
> *Virtual parameters cannot be used directly in the ASCET model, because they do not physically exist in the control unit.*

For the use of virtual parameters, a separate memory class `VIRT_PARAM` is defined in the `memorySections.xml` file. All parameters defined as virtual are assigned to this memory class.

When creating the C structure for a class or module, a separate substructure for virtual parameters is created for the memory class, the same as for all other memory classes (see section "Classes" on page 172). Unlike the substructures for normal memory classes, this substructure is not referenced in the main structure (`MOD_IMPL`).

```
struct MOD_IMPL_IRAM_SUBSTRUCT {
  uint16 cont;
};

struct MOD_IMPL_VIRTPAR_SUBSTRUCT {
  uint16 radius;
};


struct MOD_IMPL {
  struct MOD_IMPL_IRAM_SUBSTRUCT *MOD_IMPL_IRAM;
  uint16 diameter;
};
```

The reason for this special treatment is because the memory area for virtual parameters is allocated physically outside of the control unit memory. Consequently, it may not be referenced by the code. To achieve this, the memory configuration simply specifies a memory area that does not exist in the control unit (see also chapter 3.3.5 "Memory Class Configuration").

13.3.10 Dependent Parameters

Regarding the generated code, dependent parameters do not differ from normal parameters. However, their initialization value is not specified directly by the user, but determined indirectly by the code generator due to the defined dependency. Beyond that, the dependency is not reflected in any other way in the code. It is included in the ASAM-MCD-2MC file where it is used by the calibration system.

13.4 Real-Time Constructs

13.4.1 Tasks

Task are ordered collections of processes that can be activated by the application or the operating system. The activation of a task does not imply its immediate execution. The start of the task, i.e. the beginning of its execution, is scheduled by the operating system. Attributes of a task are e.g. its operating modes, its activation trigger, its priority and the mode of scheduling. On activation the processes of a task are executed in the given order.

For OSEK operating systems, tasks are marked in C source code using the TASK() macro. The expansion of this macro is OS-vendor dependant. It ensures that the task body can be called in the correct way by your OS.

ASCET and ASCET-SE support the following task scheduling modes:

- *Alarm tasks*
- *Interrupt tasks*
- *Software tasks*
- *Init tasks*

Only one Init task may exist for each application mode.

13.4.2 Processes

Processes are concurrently executable pieces of functionality. Processes are mapped into tasks, i.e. a task can call a sequence of processes.

Processes have no arguments or return value. For all targets (including ANSI C), processes are generated as `void`/`void` functions, as the following simple example shows:

```
void MOD_IMPL_process (void) {
    CL_IMPL_calc();
}
```

The only purpose of this example process is to call method `calc` from the class `CL`.

13.4.3   Messages

Messages should be used to ensure data consistency at any time during the program execution under real-time conditions. The use of "normal" global variables bears the risk of data inconsistency if, for example, a variable may be changed during its use in a process because another process with higher priority accesses the same entity.

When using messages, message copies are generated in all required cases as a result of the global analysis. This does not require any user intervention.

The user should ensure, however, that each message is sent by one process only. If different processes write to the same message in a real time environment, there is no deterministic way to define from which sender a receiver will receive the message.

> **Note**
>
> *The optimization of message copies is based on the priority scheme of an OSEK operating system. Therefore, it must be ensured that ASCET knows all tasks used on your ECU, and their priorities.*
> *If this cannot be ensured—because, e.g., the operating system you use is not OSEK compliant, or messages are accessed from outside (hand-coded sources)—, it cannot be ensured that the optimization of message copies is performed an appropriate way. This may even endanger the safety of the generated code. It is highly recommended to switch message optimization off in these cases.*

As the default optimization of message copies is not suited for all applications, the message handling can be configured extensively by the user. Four different variants exist.

Selection of Message Copy Variants at Compilation Time

The `codegen[_*].ini` files can be configured so that all supported message copy variants are generated in C code at once (`modularMessageUse=true`). Each variant is separated in generated code by pre-compiler directives `#if ... #endif`.

This allows you to choose the message copy variant at compilation time (rather than at code generation time).

The choice of message copy variant is made by defining the C macro `__MESSAGES` that can be included in the user-defined header file `message_scheme.h` or defined in the compiler options (see make variable `PROJECT_DEFINES` in `project_settings.mk`).

The following options are available:

- Optimize message copies (default):

  Messages copies are optimized by exploiting knowledge about the operating system's priority scheme. This variant is enabled by the C macro definition:

  ```
  #define __MESSAGES __OPT_COPY
  ```

  Prerequisite: For this message copy variant, it is essential that ASCET knows the priorities of every Task and ISR in the OS that uses messages. If this information is not complete then the generated code for message copies can be erroneous and there is a risk of data corruption at runtime.

- No message copies:

  Messages are used like global variables in this case. No copies are generated. This variant is enabled by the C macro definition:

  ```
  #define __MESSAGES __NO_COPY
  ```

Note

*If messages are accessed in methods in modules, **only** __OPT_COPY and __NO_COPY are available. Other optimizations are not yet supported.*

- No message optimization (always copy the message):
  Messages are always copied. This variant is enabled by the C macro definition:

  ```
  #define __MESSAGES __NON_OPT_COPY
  ```

  In this case, no optimization takes place. This variant is most flexible and can be used even if ASCET does not know the whole OS configuration ("Additional Programmer" use case).

- Use OSEK COM:
  Use OSEK COM for message communication. This is only possible if the the operating system supports OSEK-COM messaging. This variant is enabled by the C macro definition:

  ```
  #define __MESSAGES __OSEK_COM
  ```

  OSEK_COM assumes that all messages and their copies are defined by the underlying OSEK operating system. The OSEK-COM[1] API calls `Receive-Message()` and `SendMessage()` are used to access current values of messages before and after each process respectively.

Selection of Message Copy Variant at Generation Time

If only one specific message copy variant shall be generated, the `code-gen[_*].ini` option `modularMessageUse` must be set to `false`. Additionally, the option `messageUsageVariant` must be defined to specify the required message copy variant (see descriptions in `codegen_ecco.ini` for more information). In this case, C code will be generated only for the specified message copy variant, so there is no need to define the compiler macro `__MESSAGES`.

---

[1] OSEK Communication Specification, see http://www.osek-vdx.org/

13.4.4   Resources

The resources are protected by the OSEK operating system mechanisms `GetRe-`
`source` and `ReleaseResource`. The code is suited for the use with other
operating systems or in combination with handcoded sources without restric-
tions.

RTA-OSEK supports the OSEK resource `RES_SCHEDULER` (see OSEK specifica-
tion). The ceiling priority of this resource corresponds with the OS scheduler pri-
ority. In ASCET, this resource can be used only in C code. To do so, you first have
to define the resource in the C code module by clicking on the button **Resource**
( ) and name the resource e.g., `RES_SCHEDULER`).

You can then access the resource in the C code editor via the corresponding
ASCET macros, e.g.,

```
ASD_RESERVE(RES_SCHEDULER);
   /* user code */
   ...
ASD_RELEASE(RES_SCHEDULER)
```

The code generated by ASCET will then look like this:

```
...
DeclareResource(RES_SCHEDULER);
...
void process(void)
   {
   ...
   GetResource(RES_SCHEDULER);
   /* user code */
   ...
   ReleaseResource(RES_SCHEDULER);
   ...
}
```

13.4.5   Application Modes

Application modes are designed to support different runtime configurations of
the whole system at different times. This allows an easy and flexible design and
a management of system states with completely different function. Examples of
such modes are *Startup, Normal Operating Mode, Shutdown, Diagnosis* and
EEPROM Programming. Each application mode can be defined with its individual
tasks, priorities, timer configuration etc.

ASCET supports OSEK OS's application mode concept. The application mode
required is passed as a parameter to the OS's `StartOS()` API call. Control of
modes and mode switching is outside the scope of ASCET.

When integrating ASCET with RTA-OSEK V5.x is possible to re-start the OS in a
different application mode. However, such functionality is not part of the OSEK
OS standard and may not be supported by other implementations of OSEK OS.

14   Inside ASCET-SE

This chapter provides an overview of the key parts of the ASCET-SE code generator. It describes the process by which an ASCET model is converted to an executable program when the **Build → Compile** / **Build All** / **Rebuild All** menu options are selected.

This is background material for the interested reader. It is not necessary to read this chapter in order to work successfully with ASCET-SE.
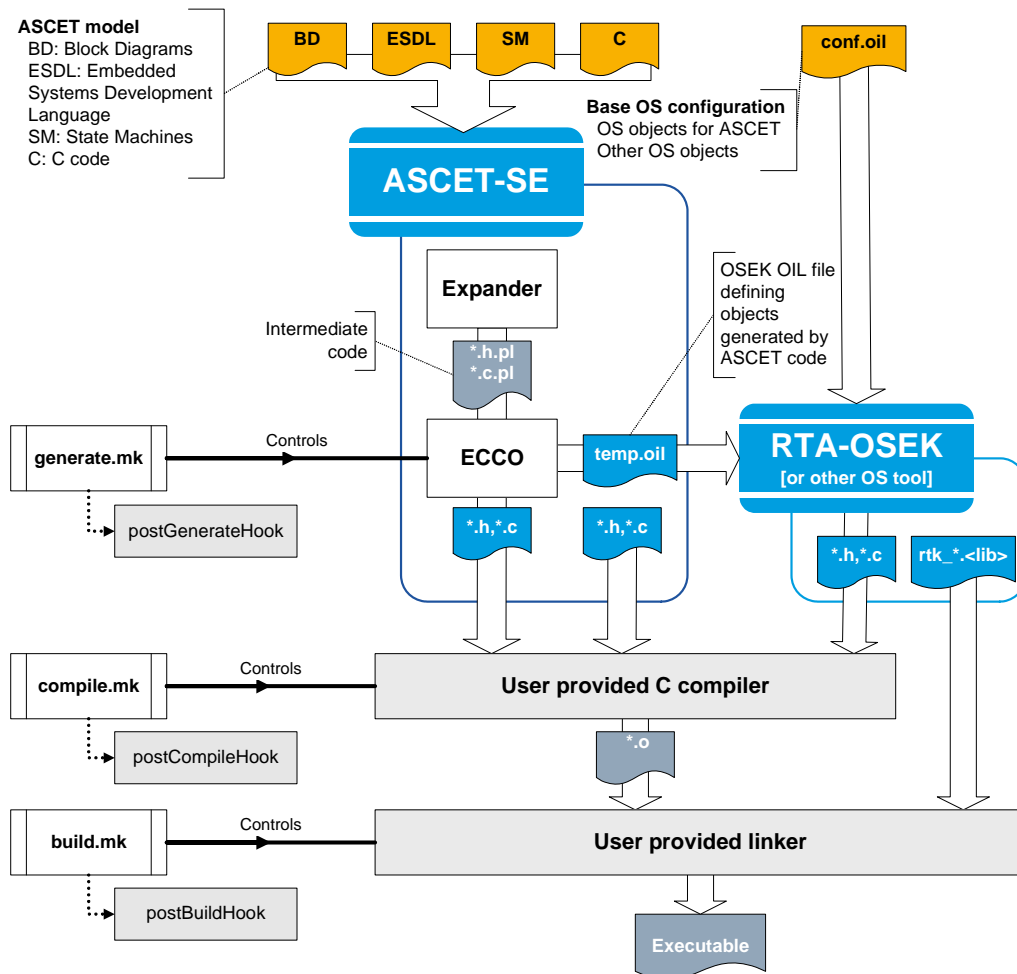
Fig. 14-1   Structure of the program generation process for ASCET-SE

Code generation in ASCET-SE is similar to compilation and has two phases.

1. Expander

In the first phase, a "front-end" called the expander converts the ASCET model specified in the block diagrams, ESDL and C code editors into intermediate code. During this phase, the physical model is transformed into the quantized model. Each module and each class are treated separately, and optimizations are done locally.

The expander writes the ASCET data model into the `CGen` directory on the hard disk. Each module specified in ASCET is expanded into three files: the database with the extension `*.db`, a header file with the extension `*.h.pl`, and the C file with the extension `*.c.pl`.

2. ECCO

In the second phase, a "back end" called ECCO (Embedded Code Creator and Optimizer) uses its global view of the ASCET project to do extensive global optimization and then converts the intermediate code into C code, adding any target compiler intrinsics (e.g. pragmas to place code into memory sections) required for the target microcontroller. ECCO uses a set of code production rules (CPRs) to do the conversion. These CPRs can be modified, within certain restrictions, to adapt the code generation to changing requirements.

The generation process is controlled by the `generate.mk` and `make-file` files. The latter is generated automatically by ASCET for the individual steps of code generation.

An OSEK OIL file, `temp.oil`, is created and RTA-OSEK is invoked on a basic OS configuration called `confVx.y.oil` to generate the OS data stuctures.

Building the executable from the generated code needs two additional phases that are managed by ASCET-SE:

3. Compile

The C source and header files generated by ECCO and RTA-OSEK are compiled by the target-specific compiler. This process is controlled by ASCET using several make files. ASCET makefiles have a `.mk` extension, e.g. `project_settings.mk` and `target_settings.mk`. The `makefile` file itself is generated by ASCET and contains all paths the user has entered via the user interface, as well as an include command for the `compile.mk` file. The following is an excerpt from the `makefile` file, using the MPC56x with RTA-OSEK as example:

```
# path definitions
P_TGROOT = C:\etas\ascet6.2\target
P_TARGET = c:\etas\ascet6.2\target\trg_mpc56x
...
P_CCROOT = c:\compiler\diab\5.0.3
...
# phase definition
include $(P_TARGET)\compile.mk
```

As a consequence of the "Smart-Compile" optimization, many different files are generated and used during the compile phase. As a result, a set of object files is created.

4. Build

The compiled files are now linked to an executable program. This process is controlled by the `build.mk` file and the specific `makefile`, as well as `project_settings.mk` and `target_settings.mk`. As a result, the user receives an executable program.

14.1   Structure of the Code Generator

The code generation subsystem has a layered structure. The tasks of the individual layers are discussed briefly in the following sections.

14.1.1   Front-End Transformation

A respective *front-end* conversion exists for each different type of specification (i.e., block diagram, state machine, ESDL). Here, the specification is analyzed syntactically. For example, a check is made to determine whether or not all necessary ports on a block were connected during graphical input. For ESDL, a parser is used. If the specification is syntactically correct, the front-end converts these files into the so-called *MDL format*.

C code modules, in which the user works directly on the implementation layer, form an exception in the specification. C code, in this case, is entered manually for the respective target. Because of this special position, C code modules are not important for the code generation. These are discussed later in this document.

14.1.2   MDL and MDL Builder

The *MDL (method definition language)* is an intermediate format, invisible to the user, which is used internally to represent all the specification types uniformly. MDL offers an object-based view. Classes and methods can be declared and defined. In addition, MDL has elements to represent real-time behavior, i.e. processes, messages, etc. Algorithms are still represented physically, without target dependence. User-specific quantizations (e.g., re-scaling with correction factors) occur later in the generation process. However, all elements (e.g., variables, method arguments) are detailed with the available implementation information in this format.

Semantic Analysis

In the MDL builder, also a general semantic analysis occurs. After this, a special analysis takes place for the implementation code generation. The mathematical expressions are analyzed semantically according to a stack-based mode of operation. The following additional checks are performed:

- Usage of non-linear conversion formulas? If yes: error message.

- Illegal mixture of floating point and integer entities? If yes: error message.

- Maximum bit width exceeded in an implementation specified by the user? If yes: error message.

- For division, does the physical interval of the denominator contain zero? If yes: error message.

- For assignments, does the physical interval of the assigned expression fit in the physical destination interval of the variables to which it is assigned? If no: warning. In this case, the generation of limiters is strongly recommended.

Collecting Optimization Data

After the semantic analysis, additional information (e.g. scaling factors, intervals) is calculated during the setup of the MDL tree. This data is used to optimize the transformation of the arithmetic and is stored with each node in the MDL tree.

Computation of physical intervals for intermediate results: The user speci-fies intervals for all variables, parameters, method arguments and return values. However, for the intermediate results found in mathematical expressions, the range of values must be computed using interval arithmetic.

Computation of optimization data: To balance the precision and efficiency of the generated code, a skillful choice of quantizations for the intermediate results is important. For each operation in a mathematical expression, a list of optimal scales is created which are based on minimizing the number of re-quantization operations. The optimization data serves as a decision base in the generation phase.

14.1.3   Code Generator

The code generator maps the object-oriented structure of the MDL to a function-oriented structure. This contains simpler language features that are more akin to C.

The code generator is still independent of the target. A distinction is made, how-ever, between experimental targets and electronic control unit targets because special optimizations are carried out for electronic control unit targets which are required even at this layer.

In ASCET, four different code generators are available for selection. They differ mostly in the method of arithmetic conversion. The four code generators corre-spond to the four phases of an integrated development. The first three phases are executed with experimental targets. The last phase corresponds to the work with a specific microcontroller target.

- *Physical experiment* produces physical entities and floating-point arith-metic (without quantizations). For this code generator, no implementation information is required.

- *Quantized physical experiment* produces a physical simulation with quan-tizations. Floating-point arithmetic is used, but value ranges and quantiza-tions can be indicated for any entity. Implementations may be partially specified and can be changed at run-time.

- *Implementation experiment* produces a simulation on the implementation layer. All implementations (e.g., data types, conversion formula, etc.) must be specified (as needed later in the *Controller Implementation*). Algo-rithms are transformed automatically into fixed-point arithmetic of the target system.

- The *object based controller implementation* performs additional optimiza-tions for the electronic control unit (e.g., imported entities are directly ref-erenced). Name conventions are converted differently. Here, names are used instead of data base IDs. The generation of fixed-point arithmetic is identical to that of the *implementation experiment*, which ensures the same behavior.

    All ASCET-SE targets are only capable of an object-based controller imple-mentation, i.e. the object structures selected in the model are mapped in the controller software.

For an ASCET module, code can be generated and simulated without project context only in the physical experiment. For the other code generators the module must be integrated into a project. This is the only way to access the implementation information. Without project context, the conversion formulas as well as all implementations of imported entities are missing.

Expander

The expander creates a target-independent intermediate code (`*.pl` files), which is used for the generation of the final, target-specific C code. It creates the desired software architecture. A substantial task of the expander is transforming the physical/mathematical expressions in the MDL into concrete calculations appearing later in the C code. It is directed by the code generator, using a standardized internal interface. The user can therefore select the expander independently of the code generator.

Unlike the MDL Builder, the expander is function oriented. The MDL tree is traversed from top to bottom recursively, in order to generate intermediate code for the individual operations that correspond to the nodes in the MDL tree. At first, code generation for individual operations is executed using basic principles in a local context, i.e. for that operation only. Then, using the value intervals and optimization data calculated during the semantic analysis, optimal code is generated for each entire mathematical expression.

The expander works on the implementation layer, i.e., it uses C data types instead of physical representations.

ECCO

Finally, the intermediate code generated by the expander is translated into executable C code by ECCO.

14.2 Code Administration

The *administration systems* described below are not directly part of the code generation subsystem. They aid the code generator and allow permanent, safe storage of automatically generated and handwritten code.

14.2.1 Make Mechanism

The Make mechanism performs the task of creating an up-to-date and consistent code version for a module. Due to modularity, the turn-around times are minimized after model changes, because code is regenerated and compiled for as few modules as possible. The Make mechanism creates a dependency network from the ASCET data model. The time stamps of each module in this network are analyzed to determine which modules must be regenerated.

Unfortunately, the time stamps are not always sufficient to decide whether regeneration is necessary. Regeneration is not required with every change in the time stamp, but this cannot be recognized automatically.

As a result, the Make mechanism is optimized for *physical experiment* code generation. Emphasis is given to achieving short turn-around times. In individual cases, too many modules or, in rare cases, too few modules get regenerated. Users should therefore select **Build → Touch → Recursive** after larger modifications to the model structure (e.g., creation/deletion of variables/methods, or changing exported/imported variables) before generating new code.

14.2.2   Code Manager

The code manager acts internally as the central interface for code generation and storage. Through this interface, all other subsystems communicate demands for code generation, the Make mechanism, and code storage. Some example functions controlled by this interface are:

- Generating source code for a component (by selecting **Build →
  Generate Code**).

- Generating the executable (by selecting **Build → Build** the code is generated, compiled, linked and stored in the ASCET database).

- Loading code into the target (e.g., by selecting **Build → Experiment**).

- Saving source code to files (by selecting **File → Export → Generated
  Code → ***). This option is only available, if the code has been stored to the database before.

- Executing a "Touch" (by selecting **Build → Touch → *** the time stamp is updated, specifically for the Make mechanism).

Code management ensures permanent, safe storage in the ASCET database of software-generated and handwritten code. For any ASCET component (i.e. module, state machine, class, etc.), several code variants may simultaneously be stored in the database as separate entities.

A code variant is essentially based on the target, code generator, and expander selection in the *code generation settings.*

> **Note**
>
> *As the target and expander are chosen in relation to each other, the target and code generator suffice to identify a code variant.*

Therefore, if one of these selections is changed at any time, a new variant will be created and stored separately. Conversely, any time one of the other Code Generation Options (e.g., protected division) is changed, the code of the existing variant is overwritten with the altered form.

When a code generator that does *not* allow different implementations is selected (e.g., "Physical Experiment"), system-generated and handwritten code is stored with the component.

> **Note**
>
> "*Physical Experiment*" *is currently the only code generator in ASCET for which this is the case.*

When a code generator that allows different implementations has been selected, system-generated and handwritten codes are stored in different locations. Generated code is stored with the project, since that is the only location where the necessary data (i.e., formulas, global variables, etc.) are available for generation with implementations. Handwritten code is, again, stored with the component of the respective implementation.

14.3   Directory Structure of the CPRs (Code Production Rules)

The *code production rules* (CPRs) are Perl programs that are stored in a directory with the following structure:
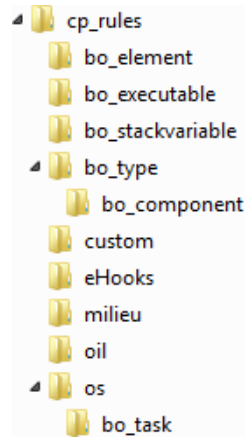
Fig. 14-2   Directory structure of the CPRs

CP Rules (Generation Base)	General Code Generation Rules
`bo_*`	Code generation adaptations for elements, types, components, and executables.
`custom` (user-specific)	User-defined rules for the code generation
`eHooks`	Rules for the eHooks package.
`milieu`	Adaptation of the code generated by ECCO to the target operating system configuration CPU frequency, prescaler
`oil`	OIL generation rules
`os`	OS generation rules

The technical prerequisite for a re-use on the CPR level is based on the following Perl feature:

For searching a function (Macro, CPR), Perl processes a list of directories that can be passed at start-up. The search ends either when the **first** function with a matching signature is found, or with an error message if no matching function is found.

If the CPRs of each component contained in ASCET-SE are stored in individual directories, and if the Perl interpreter, in the corresponding make file, is provided with a directory list that follows the order "from special CPR to general CPR", a superimposition of standard CPRs by user-specific CPRs, i.e. overwriting of the standard functionality, is possible.

15 ASCET-SE — Restrictions

This chapter describes what restrictions exist in the structure of the code generation and how these can be avoided. The known errors are also listed.

15.1 General Restrictions

15.1.1 Interval Arithmetic

The ranges for intermediate results in mathematical expressions are computed by interval arithmetic. Functional relationships cannot be recognized in this; intermediate results are always computed as if all intervals arose from input variables independent of each other. This can lead to unnecessarily large word lengths, incorrectly detected overflows, and the corresponding unnecessary loss of numerical precision. Another consequence can be the unnecessary generation of code for limiters in a later assignment.

Example: $x \in [9.0, 99.0]$. Then the expression `x/(x+1)` has the actual interval of `[0.9, 0.99]`, because `x` is in both the numerator and denominator. If, on the other hand, the interval algorithm first calculates `x+1`, `[10.0,100.0]`, the interval for `x/(x+1)` is obtained from this by dividing the intervals: `[9.0,99.0]/[10.0,100.0]=[0.09,9.9]`. This is two orders of magnitude larger than the actual interval.

Therefore, when part of a larger expression, e.g., `(x/(x+1))*y`, this excessive interval can cause an unnecessary right-shift in the immediate result (and possibly even in `y`) in order to prevent a supposedly possible overflow. This can, in turn, significantly worsen the numerical behavior of the system.

To avoid such effects, the range of the intermediate result can be explicitly specified using an additional variable, based on the known function dependence.

15.1.2 No Quantization for Literals

In rare cases, the automatic establishment of quantization of a literal according to its context can lead to unsatisfactory results if the literal is thereby represented too roughly. These cases are quite rare, however, and generally only occur for irrational literal values.

The use of a parameter or an implemented temporary variable helps to alleviate the problem.

15.1.3 ASCET Direct Access and Characteristic Maps

Direct access on a characteristic table in nested classes may lead to correct, but inefficient code.

It is expected that the expression which delivers a characteristic curve or map within a call of the interpolation routine

```
CharTable2_getAt_s8s8s8(ASD_CHTBL_PTR(Two_D),
(sint8)1,(sint8)1);
```

is a simple expression. If not, correct, but inefficient code is generated if the optimization options **Optimize Direct Access Methods *** are deactivated, e.g.,

```
ASD_INPL_CharTable2_getAt_s8s8s8(              ↵
    INNER_IMPL_getTwo_D((MIDDLE_IMPL_getInner(   ↵
    (struct MIDDLE_IMPL *)&self->Middle))).xSize, ↵
```

```
(const sint8 *)                                      ↵
(INNER_IMPL_getTwo_D((MIDDLE_IMPL_getInner(    ↵
(struct MIDDLE_IMPL *)&self->Middle))).xDist),  ↵
INNER_IMPL_getTwo_D((MIDDLE_IMPL_getInner(     ↵
(struct MIDDLE_IMPL *)&self->Middle))).ySize,  ↵
(const sint8 *)                                      ↵
(INNER_IMPL_getTwo_D((MIDDLE_IMPL_getInner(    ↵
(struct MIDDLE_IMPL *)&self->Middle))).yDist),  ↵
(const sint8 *)                                      ↵
(INNER_IMPL_getTwo_D((MIDDLE_IMPL_getInner(    ↵
(struct MIDDLE_IMPL *)&self->Middle))).values),  ↵
(sint8)1, (sint8)1 ;
```

Workaround: For performance optimization, it may be useful to use tempo-rary variables in a model if the `getAt` method of a characteristic curve or map reference shall be called, which was delivered via a method call:

```
res = Middle.Inner().Two_D().getAt(1,1);
```

In such a situation, a reference should be assigned to a temporary variable. Then, the `getAt` method of the temporary variable is called.

```
_Two_D_REF = Middle.Inner().Two_D();
res = _Two_D_REF.getAt(1,1);
```

This results in a more efficient generated code:

```
_Two_D_REF = INNER_IMPL_getTwo_D(              ↵
   (MIDDLE_IMPL_getInner((struct MIDDLE_IMPL *)  ↵
   &self->Middle)) );

ASD_INPL_CharTable2_getAt_s8s8s8(               ↵
   _Two_D_REF->xSize,(const sint8 *)            ↵
   _Two_D_REF->xDist,                            ↵
   _Two_D_REF->ySize,(const sint8 *)            ↵
   _Two_D_REF->yDist,(const sint8 *)            ↵
   _Two_D_REF->values,                          ↵
   (sint8)1, (sint8)1);
```

15.2   Restrictions in Using ASCET-SE

15.2.1   Inputs of Characteristic Curves and Maps

Restriction: Inputs of characteristic curves and maps must be static variables (stored in RAM). In the ASCET-SE software architecture, these are exported or imported class variables and also local instance variables of modules, *but not* method arguments, method local variables, or instance variables of classes.

Reason: Modern calibration systems and the ASAM-MCD-2MC format require (i.e., for display of operating point) the name and memory address of the input variable which must be stored in static RAM cell (i.e. not on the stack) for every

characteristic curve, etc. If an expression or a variable is used which is neither global nor visible, rather than in a RAM location, then the characteristic curve cannot be calibrated, or only with limitations.

Check: In the code generation, a warning indicates that the parameter is not calibratable, if applicable.

Workaround: If necessary, insert an appropriate static intermediate variable (RAM cell) in the model before the input of the characteristic curve.

15.2.2   No Separate Search for Interpolation Nodes and Interpolation

Restriction: Separate processes to search for interpolation nodes and the interpolation itself are not possible for normal (individual) characteristic curves and maps, i.e., the methods `search` and `interpolate` (extended interface of characteristic curves and maps) can not be used.

Reason: Characteristic curve objects are stored in static memory areas (ROM/FLASH) in the controller and therefore cannot contain storage spaces. To make a separate search for interpolation nodes possible, additional separate variables would always have to be created in RAM in order to store the result of the search. This is not performed for reasons of efficiency.

Check: A failure report is indicated during code generation when applicable.

15.2.3   No Choice for Interpolation Method

Restriction: Individual selection of different interpolation or extrapolation methods for characteristic curves and maps (rounded, linear) as in the simulation is not possible. Interpolation and extrapolation behaviors are determined globally by the interpolation routines used.

Reason: If the type of interpolation were to be individually selected for each characteristic curve, then it would be necessary either to provide separate routines for each type of interpolation (i.e., greater amount of code), or to use a generic routine to which the interpolation type is passed as a switch when it is called (i.e., greater amount of code and longer running time). Thus, this is not provided in the controller for reasons of efficiency.

Check: None. The interpolation routine provided (or supplied by the user) for the respective combination of characteristic curve type and data type is always called.

15.2.4   Uniqueness of Component Names

Restriction: The names of components must be unique within a project. Additionally, the project may not have the same name as a component contained within it.

Reason: The C names of functions and variables in the controller code must be readable and therefore contain the names of components. If two components in the project have the same name, then this could cause a name conflict in the code (compiler/linker error).

Check: In the Make mechanism, a failure report is indicated when applicable.

15.2.5   Make Mechanism for Controllers and Fixed-Point Arithmetic

Restriction: The make mechanism does not recognize *all* dependencies (e.g., changes of formulas, etc.) that, together with *Implementation Experiment* or *Controller Implementation*, require a regeneration of the entire project or individual project parts. If it did, the entire project would have to be analyzed, which would take about as much time as a complete regeneration.

Reason: The make mechanism for the *Object Based Controller Implementation* works the same way as for the physical simulation. Some global side effects from changes in the model are therefore not recognized.

Workaround: For changes with global effects, the user has to force a complete regeneration of the project by selecting **Component → Touch → Recursive**. Thus, the code consistency is put under the user's control.

15.3   Known Errors in the ASCET-SE Code Generation

The following errors are known for ASCET-SE. Only errors which are specially associated with controller code generation via ASCET-SE are listed here. General restrictions associated with ASCET are not given here.

15.3.1   Build Executable Code After Exiting ASCET

When you select **Build → Build All**, an executable program is generated in the temporary `..\ascet6.2\cgen` directory and stored into the ASCET database. When the **Keep files in Code Generation Directory** option is deactivated in the ASCET options (cf. ASCET online help), the content of the `.\cgen\` directory is deleted whenever you exit your ASCET session. Retrospectively activating the option has no effect for the running session.

The executable code is still in the database, but there is no way of reading it from there. The workaround is, upon re-entering ASCET, to force a new compilation of a component and relinking by selecting **Build → Touch → Flat** before rebuilding the executable.

16   **ETAS Contact Addresses**

ETAS HQ

ETAS GmbH

Borsigstraße 14	Phone:	+49 711 89661-0
70469 Stuttgart	Fax:	+49 711 89661-106
Germany	WWW:	www.etas.com

ETAS Subsidiaries and Technical Support

For details of your local sales office as well as your local technical support team and product hotlines, take a look at the ETAS website:

ETAS subsidiaries	WWW:	www.etas.com/en/contact.php
ETAS technical support	WWW:	www.etas.com/en/hotlines.php

Index